

# mXSS Attacks: Attacking well-secured Web-Applications by using innerHTML Mutations

Mario Heiderich   Tilman Frosch   Jörg Schwenk   Jonas Magazinius  
Edward Z. Yang

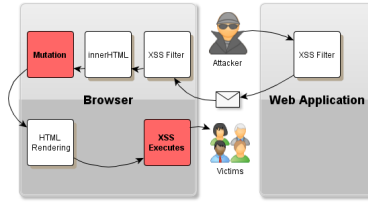
**Abstract.** Back in 2007, Hasegawa discovered a novel Cross-Site Scripting (XSS) vector based on the mistreatment of the back-tick character in a single browser implementation. This initially looked like an implementation error that could easily be fixed. Instead, as this paper shows, it was the first example of a new class of XSS vectors, the class of *mutation-based XSS (mXSS)* vectors, which may occur in `innerHTML` and related properties. mXSS affects all three major browser families: IE, Firefox, and Chrome.

We were able to place stored mXSS vectors in high-profile applications like Yahoo! Mail, Rediff Mail, OpenExchange, Zimbra, Roundcube, and several commercial products. m-XSS vectors bypassed widely deployed server-side XSS protection techniques (like HTML Purifier, kses, htmlLawed, Blueprint and Google Caja), client-side filters (XSS Auditor, IE XSS Filter), Web Application Firewall (WAF) systems, as well as Intrusion Detection and Intrusion Prevention Systems (IDS/IPS). We describe a scenario in which seemingly immune entities are being rendered prone to an attack based on the behavior of an involved party, in our case the browser. Moreover, it proves very difficult to mitigate these attacks: In browser implementations, mXSS is closely related to performance enhancements applied to the HTML code before rendering; in server side filters, strict filter rules would break many web applications since the mXSS vectors presented in this paper are harmless when sent to the browser.

This paper introduces and discusses a set of seven different subclasses of mXSS attacks, among which only one was previously known. The work evaluates the attack surface, showcases examples of vulnerable high-profile applications, and provides a set of practicable and low-overhead solutions to defend against these kinds of attacks.

## 1 Introduction

*Mutation-based Cross-Site-Scripting (mXSS)* Server- and client-side XSS filters share the assumption that their HTML output and the browser-rendered HTML content are mostly identical. In this paper, we show how



**Fig. 1.** Information flow in an mXSS attack.

this premise is *false* for important classes of web applications that use the `innerHTML` property to process user-contributed content. Instead, this very content is *mutated* by the browser, such that a harmless string that passes nearly all of the deployed XSS filters is subsequently transformed into an active XSS attack vector by the browser layout engine itself.

The information flow of an mXSS attack is shown in Figure 1: The attacker carefully prepares an HTML or XML formatted string and injects it into a web application. This string will be filtered or even rewritten in a server-side XSS filter, and will then be passed to the browser. If the browser contains a client-side XSS filter, the string will be checked again. At this point, the string is still harmless and cannot be used to execute an XSS attack.

However, as soon as this string is inserted into the browser’s DOM by using the `innerHTML` property, the browser will *mutate* the string. This mutation is highly unpredictable since it is not part of the specified `innerHTML` handling, but is a proprietary optimization of HTML code implemented differently in each of the major browser families. The mutated string now contains a valid XSS vector, and the attack will be executed on rendering of the new DOM element. Both server- and client side filters were unable to detect this attack because the string scanned in these filters did not contain any executable code.

Mutation-based XSS (mXSS) makes an impact on all three major browser families (IE, Firefox, Chrome). Table 1 gives an overview on the mXSS subclasses discovered so far, and points to their detailed description. A web application is vulnerable if it inserts user-contributed input with the help of `innerHTML` or related properties into the DOM of the browser. It is difficult to statistically evaluate the number of websites affected by the seven attack vectors covered in this paper, since automated testing fails to reliably detect all these attack prerequisites: If `innerHTML` is only used to insert trusted code from the web application itself into the DOM, it is not vulnerable. However, it can be stated that amongst the 10.000 most popular web pages, roughly one third uses the `innerHTML` property, and about 65% use JavaScript libraries like jQuery [7], who

| Description   | Section |
|---|---------|
| Backtick Characters breaking Attribute Delimiter Syntax         | 3.1     |
| XML Namespaces in Unknown Elements causing Structural Mutation  | 3.2     |
| Backslashes in CSS Escapes causing String-Boundary Violation    | 3.3     |
| Misfit Characters in Entity Representation breaking CSS Strings | 3.4     |
| CSS Escapes in Property Names violating entire HTML Structure   | 3.5     |
| Entity-Mutation in non-HTML Documents                           | 3.6     |
| Entity-Mutation in non-HTML context of HTML documents           | 3.7     |

**Table 1.** Overview on the mXSS vectors discussed in this paper

abet mXSS attacks by using the *innerHTML* property instead of the corresponding DOM methods.

However, it is possible to single out a large class of vulnerable applications (Webmailers) and name high-profile state-of-the-art XSS protection techniques that can be circumvented with mXSS. Thus the alarm we want to raise with this paper is that an important class of web applications is affected, and that nearly all XSS mitigation techniques fail.

*Webmail Clients* Webmail constitutes a class of web applications particularly affected by mutation-based XSS: nearly all of them (including e.g. Microsoft Hotmail, Yahoo! Mail, Rediff Mail, OpenExchange, Roundcube and other tools and providers) were vulnerable to the vectors described in this paper. These applications use the *innerHTML* property to display user-generated HTML email content. Before doing so, the content is thoroughly filtered by server-side anti-XSS libraries in recognition of the dangers of a stored XSS attack. The vectors described in this paper will pass through the filter because the HTML string contained in the email body does not form a valid XSS vector – but would require only a single *innerHTML* access to be turned into an attack by the browser itself.

Here the attacker may submit the attack vector within the HTML-formatted body of an email. Most webmail clients do not use *innerHTML* to display this email in the browser, but a simple click on the “Reply” button may trigger the attack: to attach the contents of the mail body to the reply being edited in the webmail client, mostly *innerHTML* access is used.

*HTML Sanitizers* We analysed a large variety of HTML sanitizers such as HTML Purifier, htmLawed, OWASP AntiSamy, jSoup, kses and various commercial providers. At the time of testing, all of them were (and many of them still are) vulnerable against mXSS attacks. Although some of the authors reacted with solutions, the major effort was to alert the browser vendors and trigger fixes for the *innerHTML*-transformations. In fact, several of our bug reports have led to subsequent changes in browser behavior. To protect users, we have decided to anonymise names of several formerly affected browsers and applications used as examples in our work.

This paper makes the following contributions:

1. We identify an undocumented but long-existing threat against web applications, which enables an attacker to conduct XSS attacks, even if strong server- and client-side filters are applied. This novel class of attack vectors utilize performance-enhancement peculiarities present in all major browsers, which *mutate* a given HTML string before it is rendered. We propose the term *mXSS* (for Mutation-based XSS) to describe this class of attacks to disambiguate and distinguish them from classic, reflected, persistent and DOM-based XSS attacks.
2. We discuss client- and server-side mitigation mechanisms. In particular, we propose and evaluate an in-browser protection script, entirely composed in JavaScript, which is practical, feasible and has low-overhead. With this script, a web application developer can implement a fix against mXSS attacks without relying on server-side changes or browser updates. The script overwrites the getter methods of the DOM properties we identified as vulnerable and changes the HTML handling into an XML-based processing, thereby effectively mitigating the attacks and stopping the mutation effects<sup>1</sup>.
3. We evaluated this attack in three ways: first, we analyzed the attack surface for mXSS and give a rough estimate the number of vulnerable applications on the Internet; second, we conducted a field study testing commonly used web applications such as Yahoo! Mail and other high profile websites, determining whether they could be subjected to mXSS attacks; third, we have examined common XSS filter software such as AntiSamy, HTML Purifier, Google Caja and Blueprint for mXSS vulnerabilities, subsequently reporting our findings back to the appropriate tool's author(s).

## 2 Problem Description

In the following sections, we describe the attack vectors which arise from the use of the *innerHTML* property in websites. We will outline the his-

<sup>1</sup> In result, one can purposefully choose XML-based processing for security-critical sites and HTML-based processing for performance-critical sites.

tory of findings and recount a realistic attack scenario. The problems we identify leave websites vulnerable against the novel kind of mXSS attacks, even if the utilized filter software fully protects against the dangers of the classic Cross-Site Scripting.

## 2.1 The innerHTML Property

Originally introduced to browsers by Microsoft with Internet Explorer 4, the property quickly gained popularity among web developers and was adopted by other browsers, despite being non-standard. The use of *innerHTML* and *outerHTML* is supported by each and every one of the commonly used browsers in the present landscape. Consequently, the W3C started a specification draft to unify *innerHTML* rendering behaviors across browser implementations [20].

An HTML element's *innerHTML* property deals with creating HTML content from arbitrarily formatted strings on write access on the one hand, and with serializing HTML DOM nodes into strings on read access on the other. Both directions are relevant in scope of our paper – the read access is necessary to trigger the mutation while the write access will attach the transformed malicious content to the DOM. The W3C working draft document, which is far from completion, describes this process as generation of an ordered set of nodes from a string valued attribute. Due to being attached to a certain *context node*, if this attribute is evaluated, all children of the context node are replaced by the (ordered) node-set generated from the string.

To use *innerHTML*, the DOM interface of `element` is enhanced with an `innerHTML` attribute/property. Setting of this attribute can occur via the `element.innerHTML=value` syntax, and in this case the attribute will be evaluated immediately. A typical usage example of `innerHTML` is shown in Listing 1.1: when the HTML document is first rendered, the `<p>` element contains the "First text" text node. When the anchor element is clicked, the content of the `<p>` element is replaced by the "New `<b>second</b>` text." HTML formatted string.

**Listing 1.1.** Example on `innerHTML` usage

```
<script type="text/javascript">
  var new = "New <b>second</b> text.";
  function Change () {
    document.all.myPar.innerHTML = new;
  }
</script>
<p id="myPar">First text.</p>
<a href="javascript:Change()">
  Change text above!
</a>
```

`outerHTML` displays similar behavior with single exception: unlike in the `innerHTML` case, the whole context (not only the *content* of the context node) will be replaced here. The *innerHTML-access changes the utilized markup though* for several reasons and in differing ways depending on the user agent. The following code listings show some (non security-related) examples of these performance optimizations:

**Listing 1.2.** Examples for internal HTML mutations to save CPU cycles

```
<!-- User Input -->
<s class="">hello&#x20;<b>goodbye</b>

<!-- Browser-transformed Output -->
<S>hello <B>goodbye</B></S>
```

The browser – in this case Internet Explorer 8 – mutates the input string in multiple ways before sending it to the layout engine: the empty *class* is removed, the tag names are set to upper-case, the markup is sanitized and the HTML entities are resolved. These transformations happen in several scenarios:

1. Access to the *innerHTML* or *outerHTML* properties of the affected or parent HTML element nodes;
2. Copy (and subsequent paste) interaction with the HTML data containing the affected nodes;
3. HTML editor access via the *contenteditable* attribute, the *designMode* property or other DOM method calls like *document.execCommand()*;
4. Rendering the document in a print preview container or similar intermediate views. Browsers tend to use the *outerHTML* property of the HTML container or the *innerHTML*.

For the sake of brevity, we will use the term *innerHTML-access* to refer to some or all of the items from the above list.

## 2.2 Problem History and Early Findings

In 2006, a non-security related bug report was filed by a user, noting an apparent flaw in the print preview system for HTML documents rendered by a popular web browser. Hasegawa's 2007 analysis [11] of this bug report showed that once the *innerHTML* property of an element's container node in an HTML tree was accessed, the attributes delimited by backticks or containing values starting with backticks were replaced with regular ASCII quote delimiters: the content had *mutated*. Often the regular quotes disappeared, leaving the backtick characters unquoted and therefore vulnerable to injections. As Hasegawa states, an attacker can craft input operational for bypassing XSS detection systems because of

its benign nature, yet having a future possibility of getting transformed by the browser into a code that executes arbitrary JavaScript code. An example vector is being discussed in Section 3.1. This behavior constitutes a fundamental basis for our research on the attacks and mitigations documented in this paper.

### 2.3 Mutation-based Cross-Site Scripting

Certain websites permit their users to submit inactive HTML aimed at visual and structural improvement of the content they wish to present. Typical examples are web-mailers (visualization of HTML-mail content provided by the *sender* of the e-mail) or collaborative editing of complex HTML-based documents (HTML content provided by all editors).

To protect these applications and their users from XSS attacks, website owners tend to call server-side HTML filters like e.g. the HTML Purifier, mentioned in Section 5.1, for assistance. These HTML filters are highly skilled and configurable tool-kits, capable of catching potentially harmful HTML and removing it from benign content. While it has become almost impossible to bypass those filters with regular HTML/Javascript strings, the mXSS problem has yet to be tackled by most libraries. The core issue is as follows: the HTML markup an attacker uses to initiate an mXSS attack is considered harmless and contains no active elements or potentially malicious attributes – the attack vector examples shown in Section 3 demonstrate that.

Only the browser will transform the markup internally (each browser family in a different manner), thereby unfolding the embedded attack vector and executing the malicious code. As previously mentioned, such attacks can be labeled mXSS – XSS attacks that are only successful because the attack vector is *mutated* by the browser, a result of behavioral mishaps introduced by the internal HTML processing of the user agents.

## 3 Exploits

The following sections describe a set of *innerHTML*-based attacks we discovered during our research on DOM mutation and string transformation. We present the code purposefully appearing as sane and inactive markup before the transformation occurs, while it then becomes an active XSS vector executing the example method `xss()` after that said transformation. This way server- and client-side XSS filters are being elegantly bypassed.

The code shown in Listing 1.3 provides one basic example of how to activate (Step 2 in the chain of events described in Section 4) each and any of the subsequently following exploits – it simply concatenates

an empty string to an existing *innerHTML* property. The exploits can further be triggered by the DOM operations mentioned in Section 2.2. Any *innerHTML*-access mentioned in the following sections signifies a reference to a general usage of the DOM operations framed by this work.

**Listing 1.3.** Code-snippet – illustrating the minimal amount of DOM-transaction necessary to cause and trigger mXSS attacks

```
<script>
window.onload = function(){
    document.body.innerHTML += '';
}
</script>
```

We created a test-suite to analyze the *innerHTML* transformations in a systematic way; this tool was later published on a related website dedicated to HTML and HTML5 security implications <sup>2</sup>. The important *innerHTML*-transformations are highlighted in the code examples to follow.

### 3.1 Backtick Characters breaking Attribute Delimiter Syntax

This DOM string-mutation and the resulting attack technique was first publicly documented in 2007, in connection with the original print-preview bug described in Section 2.2. Meanwhile, the attack can only be used in legacy browsers as their modern counterparts have deployed effective fixes against this problem. Nevertheless, the majority of tested web applications and XSS filter frameworks remain vulnerable against this kind of attack – albeit measurable existence of a legacy browser user-base. The code shown in Listing 1.4 demonstrates the initial attack vector and the resulting transformation performed by the browser engine during the processing of the *innerHTML* property.

**Listing 1.4.** *innerHTML*-access to an element with backtick attribute values causes JavaScript execution

```
<!-- Attacker Input -->


<!-- Browser Output -->
<IMG alt="`onload=xss()" src="test.jpg">
```

### 3.2 XML Namespaces in Unknown Elements causing Structural Mutation

A browser that does not yet support the HTML5 standard is likely to interpret elements such as *article*, *aside*, *menu* and others as *unknown elements*. A developer can decide how an unknown element is to be treated

<sup>2</sup> *innerHTML* Test-Suite, <http://html5sec.org/innerHTML>, 2012



by the browser: A common way to pass these instructions is to use the *xmlns* attribute, thus providing information on which XML namespace the element is supposed to reside on. Once the *xmlns* attribute is being filled with data, the visual effects often do not change when compared to none or empty namespace declarations. However, once the *innerHTML* property of one of the element's container nodes is being accessed, a very unusual behavior can be observed. The browser prefixes the unknown but namespaced element with the XML namespace that in itself contains unquoted input from the *xmlns* attribute. The code shown in Listing 1.5 demonstrates this case.

**Listing 1.5.** innerHTML-access to an unknown element causes mutation and unsolicited JavaScript execution

```
<!-- Attacker Input -->
<article xmlns="urn:img src=x onerror=xss()/">123

<!-- Browser Output -->
<img src=x onerror=xss()//:article xmlns="urn:img src=x on
error=xss()/">123</img src=x onerror=xss()//:article>
```

The result of this structural mutation and the pseudo-namespace allowing white-space is an injection point. It is through this point that an attacker can simply abuse the fact that an attribute value is being rendered despite its malformed nature, consequently smuggling arbitrary HTML into the DOM and executing JavaScript. This problem was reported and fixed in the modern browsers. A similar issue was discovered and published by Silin<sup>3</sup>.

### 3.3 Backslashes in CSS Escapes causing String-Boundary Violation

To properly escape syntactically relevant characters in CSS property values, the CSS1 and CSS2 specifications propose CSS escapes. These cover the Unicode range and allow to, for instance, use the single-quote character without risk. This is possible even inside a CSS string that is delimited by single quotes. Per specification, the correct usage for CSS escapes inside CSS string values would be: **property: 'v\61 lue'**. The escape sequence is representing the *a* character, based on its position in the ASCII table of characters. Unicode values can be represented by escaping sequences such as *\20AC* for the € glyph, to give one example.

<sup>3</sup> Silin, A., *XSS using "xmlns" attribute in custom tag when copying innerHTML*, <http://html5sec.org/?xmlns#97>, Dec. 2011

Several modern browsers nevertheless break the security promises indicated by the correct and standards-driven usage of CSS escapes. In particular, it takes place for the *innerHTML* property of a parent element being accessed. We observed a behavior that converted escapes to their canonical representation. The sequence `property: 'val\27ue'` would result in the *innerHTML* representation `PROPERTY: 'val'ue'`. An attacker can abuse this behavior by injecting arbitrary CSS code hidden inside a properly quoted and escaped CSS string. This way HTML filters checking for valid code that observes the standards can be bypassed, as depicted in Listing 1.6.

**Listing 1.6.** *innerHTML*-access to an element using CSS escapes in CSS strings causes JavaScript execution

```
<!-- Attacker Input -->
<p style="font-family: 'ar\27\3bx\3a
expression\28xss\28\29\29\3bial ' "></p>

<!-- Browser Output -->
<P style="FONT-FAMILY: 'ar';x:expression(xss());ial ' "></P>
```

Unlike the backtick-based attacks described in Section 3.1, this technique allows recursive mutation. This means that, for example, a double-escaped or double-encoded character will be double-decoded in case that *innerHTML*-access occurs twice. More specifically, the `\5c 5c` escape sequence will be broken down to the `\5c` sequence after first *innerHTML*-access, and consequently decoded to the `\` character after the second *innerHTML*-access.

During our attack surface’s evaluation, we discovered that some of the tested HTML filters could be bypassed with the use of `&#amp;x5c 5c 5c 5c` or alike sequences. Due to the backslashes’ presence allowed in CSS property values, the HTML entity representation combined with the recursive decoding feature had to be employed for code execution and attack payload delivery.

The attacks that become possible through this technique range from overlay attacks injecting otherwise unsolicited CSS properties (such as positioning instructions and negative margins), to arbitrary JavaScript execution, font injections (as described by Heiderich et al. [13]), and the DHTML behavior injections for leveraging XSS and ActiveX-based attacks.

### 3.4 Misfit Characters in Entity Representation breaking CSS Strings

Combining aforementioned exploit with enabling CSS-escape decoding behavior results in yet another interesting effect observable in several browsers. That is, when both CSS escape and the canonical representation for the double-quote character inside a CSS string are used, the render engine converts them into a single quote, regardless of those two

characters seeming unrelated. This means that the `\22, &quot;;, &#x22;` and `&#34;` character sequences will be converted to the `'` character upon *innerHTML*-access. Based on the fact that both characters have syntactic relevance in CSS, the severity of the problems arising from this behavior is grand. The code example displayed in Listing 1.7 shows a mutation-based XSS attack example. To sum up and underline once again, it is based on fully valid and inactive HTML and CSS markup that will unfold to active code once the *innerHTML*-access is involved.

**Listing 1.7.** *innerHTML*-access to an element using CSS strings containing misfit HTML entities causes JavaScript execution

```
<!-- Attacker Input -->
<p style="font-family: 'ar&quot;;x=expression(xss())/ *ial
'"></p>

<!-- Browser Output -->
<P style="FONT-FAMILY: 'ar';x=expression(xss())/ *ial
'"></P>
```

We can only speculate about the reasons for this surprising behavior. One potential explanation is that in case when the *innerHTML* transformation might lead the `\22, &quot;;, &#x22;` and `&#34;` sequences to be converted into the actual double-quote character (`"`), then – given that the attribute itself is being delimited with double-quotes – an improper handling could not only break the CSS string but even disrupt the syntactic validity of the surrounding HTML. An attacker could abuse that to terminate the attribute with a CSS escape or HTML entity, and, afterwards, inject crimson HTML to cause an XSS attack.

Our tests showed that it is not possible to break the HTML markup syntax with CSS escapes once used in a CSS string or any other CSS property value. The mutation effects only allow CSS strings to be terminated illegitimately and lead to an introduction of new CSS property-value pairs. Depending on the browser, this may very well lead to an XSS exploit executing arbitrary JavaScript code. Supporting this theory, the attack technique shown in Section 3.5 considers markup integrity but omits CSS string sanity considerations within the transformation algorithm of HTML entities and CSS escapes.

### 3.5 CSS Escapes in Property Names violating entire HTML Structure

As mentioned in Section 3.4, an attacker cannot abuse mutation-based attacks to break the markup structure of the document containing the style attribute hosting the CSS escapes and entities. Thus far, the CSS escapes and entities were used exclusively in CSS property values and not in the property names. Applying the formerly discussed techniques

to CSS property names instead of values forces some browsers into a completely different behavior, as demonstrated in Listing 1.8.

**Listing 1.8.** innerHTML-access to an element with invalid CSS property names causes JavaScript execution

```
<!-- Attacker Input -->


<!-- Browser Output -->
<IMG style="font-fa"onload=xss() mily:
    'arial'" src="test.jpg">
```

Creating a successful exploit, which is capable of executing arbitrary JavaScript, requires an attacker to first terminate the style attribute by using a CSS escape. Therefore, the injected code would trigger the exploit code while it still follows the CSS syntax rules. Otherwise, the browser would simply remove the property-value pair deemed invalid. This syntax constraint renders several characters useless for creating exploits. White-space characters, colon, equals, curly brackets and the semi colon are among them. To bypass the restriction, the attacker simply needs to escape those characters as well. We illustrate this in Listing 1.8. By escaping the entire attack payload, the adversary can abuse the mutation feature and deliver arbitrary CSS-escaped HTML code.

Note that the attack only works with the double-quote representation inside double-quoted attributes. Once a website uses single-quotes to delimit attributes, the technique can be no longer applied. The *innerHTML*-access will convert single quotes to double quotes. Then again, the \22 escape sequence can be used to break and terminate the attribute value. The code displayed in Listing 1.9 showcases this yet again surprising effect.

**Listing 1.9.** Example for automatic quote conversion on innerHTML-access

```
<!-- Example Attacker Input -->
<p style='fo\27\22o:bar'>

<!-- Example Browser Output -->
<P style="fo"o: bar"></P>
```

### 3.6 Entity-Mutation in non-HTML Documents

Once a document is being rendered in XHTML/XML mode, different rules apply to the handling of character entities, non-wellformed content including unquoted attributes, unclosed tags and elements, invalid elements nesting and other aspects of document structure. A web-server can instruct a browser to render a document in XHTML/XML by setting a

matching MIME type via Content-Type HTTP headers; in particular the MIME *text/xhtml*, *text/xml*, *application/xhtml+xml* and *application/xml* types can be employed for this task (more exotic MIME types like *image/svg+xml* and *application/vnd.wap.xhtml+xml* can also be used).

These specific and MIME-type dependent parser behaviors cause several browsers to show anomalies when, for instance, CSS strings in style elements are exercised in combination with (X)HTML entities. Several of these behaviors can be used in the context of mutation-based XSS attacks, as the code example in Listing 1.10 shows.

**Listing 1.10.** innerHTML-access to an element with encoded XHTML in CSS string values causes JavaScript execution

```
<!-- Attacker Input -->
<style>{*{font-family:'ar&lt;img src=&quot;test.jpg&quot;
onload=&quot;xss()&quot; /&gt;ial '}}</style>

<!-- Browser Output -->
<style>{*{font-family:'arial '}}</style>
```

Here-above, the browser automatically decodes the HTML entities hidden in the CSS string specifying the font family. By doing so, the parser must assume that the CSS string contains actual HTML. While in *text/html* neither a mutation nor any form or parser confusion leading to script execution would occur, in *text/xhtml* and various related MIME type rendering modes, a CSS style element is supposed to be capable of containing other markup elements. Thus, without leaving the context of the style element, the parser decides to equally consider the decoded *img* element hidden in the CSS string, evaluate it and thereby execute the JavaScript connected to the successful activation of the event handler. This problem is unique to the WebKit browser family, although similar issues were spotted in other browser engines. Beware that despite a very small distribution of sites using MIME types such as *text/xhtml*, *text/xml*, *application/xhtml+xml* and *application/xml* (0.0075% in the Alexa Top 1 Million website list), an attacker might abuse MIME sniffing, frame inheritance and other techniques to force a website into the necessary rendering mode, purposefully acting towards a successful exploit execution. The topic of security issues arising from MIME-sniffing has been covered by Barth et al., Gebre et al. and others [1,3,8].

### 3.7 Entity-Mutation in non-HTML context of HTML documents

In-line SVG support provided in older browsers could lead to XSS attacks originating in HTML entities that were embedded inside style and similar elements, which are by default evaluated in their canonic form upon

the occurrence of *innerHTML*-access. This problem has been reported and mitigated by the affected browser vendors and is listed here to further support our argument. The code example in Listing 1.11 showcases anatomy of this attack.

**Listing 1.11.** Misusing HTML entities in inline-SVG CSS-string properties to execute arbitrary JavaScript

```
<!-- Attacker Input -->
<p><svg><style>*<{font-family: '
    &lt;&sol;&style&gt;&lt;img/src=x&Tab;
    onerror=xss()&sol;&sol;' }</style></svg></p>

<!-- Browser Output -->
<p><svg><style>*<{font-family: ' </style></svg>' }</p>
```

This vulnerability was present in a popular open-source user agent and has been since fixed successfully, following a bug report.

### 3.8 Summary

In order to initiate the mutation, all of the exploits shown here require a single access to the *innerHTML* property of a surrounding container, while except for the attack vector discussed in Section 3.1, all other attacks can be upgraded to allow recursive mutation – making double-, triple- and further multiply-encoded escapes and entities useful in the attack scenario, immediately when multiple *innerHTML*-access to the same element takes place. The attacks were successfully tested against a large range of publicly available web applications and XSS filters – see Section 4.

## 4 Attack Surface

The attacks outlined in this paper target the client-side web application components, e.g. JavaScript code, that use the *innerHTML* property to perform dynamic updates to the content of the page. Rich text editors, web email clients, dynamic content management systems and components that pre-load resources constitute the examples of such features. In this section we detail the conditions under which a web application is vulnerable. Additionally, we attempt to estimate the prevalence of these conditions in web pages at present.

The basic conditions for a mutation event to occur are the serialization and deserialization of data. As mentioned in Section 2, mutation in the serialization of the DOM-tree occurs when the *innerHTML* property of a DOM-node is accessed. Subsequently, when the mutated content is parsed

back into a DOM-tree, e.g. when assigned to *innerHTML* or written to the document using `document.write`, the mutation is activated.

The instances in Listing 1.12 are far from being the exclusive methods for a mutation event to occur, but they exemplify vulnerable code patterns. In order for an attacker to exploit such a mutation event, it must take place on the attacker-supplied data. This condition makes it difficult to statistically estimate the number of vulnerable websites, however, the attack surface can be examined through an evaluation of the number of websites using such vulnerable code patterns.

**Listing 1.12.** Code snippets – vulnerable code patterns

```
// Native JavaScript / DOM code
a.innerHTML = b.innerHTML;
a.innerHTML += 'additional content';
a.insertAdjacentHTML('beforebegin', b.innerHTML);
document.write(a.innerHTML);

// Library code
$(element).html('additional content');
```

#### 4.1 InnerHTML Usage

Since an automated search for *innerHTML* does not determine the exploitability of its usage, it can only serve as an indication for the severity of the problem. To evaluate the prevalence of *innerHTML* usage on the web, we conducted a study of the Alexa top 10,000 most popular web sites. A large fraction of approximately one third of these web sites utilized vulnerable code patterns, like the ones in Listing 1.12, in their code for updating page content. Major websites like Google, Amazon, EBay and Microsoft could be identified among these. Again, this does not suggest that these web sites can be exploited. We found an overall of 74.5% of the Alexa Top 1000 websites to be using *innerHTML*-assignments. While the usage of *innerHTML* is very common, the circumstances under which it is vulnerable to exploitation are in fact hard to quantify. Note though that almost all applications applied with an editable HTML area are prone to being vulnerable.

Additionally, there are some notable examples of potentially vulnerable code patterns identifiable in multiple and commonly used JavaScript libraries, e.g. jQuery [7] and SWFObject [27]. Indeed, more than 65% of the top 10,000 most popular websites do employ one of these popular libraries (with 48.87% using jQuery), the code of which could be used to trigger actual attacks. Further studies have to be made as to whether or not web applications reliant on any of these libraries are affected, as it largely depends on how the libraries are used. In certain cases, a very specific set of actions needs to be performed if the vulnerable section of

the code is to be reached. Regardless, library's inclusion always puts a given website at risk of attacks.

Ultimately, we queried the Google Code Search Engine (GCSE) as well as the Github search tool to determine which libraries and public source files make use of potentially dangerous code patterns. The search query yielded an overall 184,000 positive samples using the GCSE and 1,196,000 positive samples using the Github search tool. While this does not provide us with an absolute number of vulnerable websites, it shows how widely the usage of *innerHTML* is distributed; any of these libraries using vulnerable code patterns in combination with user-generated content is likely to be vulnerable to mXSS attacks.

## 4.2 Web-Mailers

A class of web applications particularly vulnerable to m- XSS attacks are classic web-mailers – applications that facilitates receiving, reading and managing HTML mails in a browser. In this example, the fact that HTML Rich-Text Editors (RTE) are usually involved, forms the basis for the use of the *innerHTML* property, which is being triggered with almost any interaction with the mail content. This includes composing, replying, spell-checking and other common features of applications of this kind. A special case of attack vector is sending an mXSS string within the body of an HTML-formatted mail. We analyzed commonly used web-mail applications and spotted mXSS vulnerabilities in almost every single one of them, including e.g. Microsoft Hotmail, Yahoo! Mail, Rediff Mail, OpenExchange, Roundcube, and many other products – some of which cannot yet be named for the sake of user protection. The discovery was quickly followed with bug reports sent to the respective vendors, which were acknowledged.

## 4.3 Server-Side XSS Filters

The class of mXSS attacks poses a major challenge for server-side XSS filters. To completely mitigate these attacks, they would have to *simulate* the mutation effects of the three major browser families in hopes of determining whether a given string may be an mXSS vector. At the same time, they should not filter benign content, in order not to break the web application. The fixes applied to HTML sanitizers, as mentioned in the introduction, are new rules for *known* mutation effects. It can be seen as a challenging task to develop new filtering paradigms that may discover even unknown attack vectors.



## 5 Mitigation Techniques

The following sections will describe a set of mitigation techniques that can be applied by website owners, developers, or even users to protect against the cause and impact of mutation XSS attacks. We provide details on two approaches. The first one is based on a server-side filter, whereas the other focuses on client-side protection and employs an interception method in critical DOM properties access management.

### 5.1 Server-side mitigation

Avoiding outputting server content otherwise incorrectly converted by the browsers is the most direct mitigation strategy. In specific terms, the flawed content should be replaced with semantically equivalent content which is converted properly. Let us underline that the belief stating that “well-formed HTML is unambiguous” is false: only a browser-dependent subset of well-formed HTML will be preserved across *innerHTML*-access and -transactions.

A comprehensible and uncomplicated policy is to simply disallow any of the special characters for which browsers are known to have trouble with when it comes to a proper conversion. For many HTML attributes and CSS properties this is not a problem, since their set of allowed values already excludes these particular special characters. Unfortunately, in case of free-form content, such a policy may be too stringent. For HTML attributes, we can easily refine our directive by observing that ambiguity only occurs when the browser omits quotes from its serialized representation. Insertion of quotes can be guaranteed by, for example, appending a trailing whitespace to text, a change unlikely to modify the semantics of the original text. Indeed, the W3C specification states that user agents may ignore surrounding whitespace in attributes. A more aggressive transformation would only insert a space when the attribute was to be serialized without quotes, yet contained a backtick. It should be noted that backtick remains the only character which causes Internet Explorer to mis-parse the resulting HTML.

For CSS, refining our policy is more difficult. Due to the improper conversion of escape sequences, we cannot allow any CSS special characters in general, even in their escaped form. For URLs in particular, parentheses and single quotes are valid characters in a URL, but are simultaneously considered special characters in CSS. Fortunately, most major web servers are ready to accept percent encoded versions of these characters as equivalent, so it is sufficient to utilize the common percent-escaping for these characters in URLs instead.

We have implemented these mitigation strategies in HTML Purifier, a popular HTML filtering library [32]; as HTML Purifier does not implement any anomaly detection, the filter was fully vulnerable to these

attacks. These fixes were reminiscent of similar security bugs that were tackled in 2010 [31] and subsequent releases in 2011 and 2012. In that case, the set of unambiguous encodings was smaller than that suggested by the specification, so a very delicate fix had to be crafted in result, both fixing the bug and still allowing the same level of expressiveness. Since browser behavior varies to a great degree, a server-side mitigation of this style is solely practical for the handling of a subset of HTML, which would normally be allowed for high-risk user-submitted content. Furthermore, this strategy cannot protect against dynamically generated content, a limitation which will be addressed in the next section. Note that problems such as the backtick-mutation still affect the HTML Purifier as well as Blueprint and Google Caja; they have only just been addressed successfully by the OWASP Java HTML Sanitizer Project <sup>4</sup>.

## 5.2 Client-side mitigation

Browsers implementing ECMA Script 5 and higher offer an interface for another client-side fix. The approach makes use of the developer-granted possibility to overwrite the handlers of *innerHTML* and *outerHTML*-access to intercept the performance optimization and, consequently, the markup mutation process as well. Instead of permitting a browser to employ its own proprietary HTML optimization routines, we utilize the internal XML processor a browser provides via DOM. The technique describing the wrapping and sanitation process has been labeled *TrueHTML*.

The TrueHTML relies on the *XMLSerializer* DOM object provided by all of the user agents tested. The *XMLSerializer* can be used to perform several operations on XML documents and strings. Interestingly, *XMLSerializer.serializeToString()* will accept an arbitrary DOM structure or node collection and transform it into an XML string. We decided to replace the *innerHTML*-getters with an interceptor to process the accessed contents as if they were actual XML. This has the following benefits:

1. The resulting string output is free from all mutations described and documented in Section 3. The attack surface can therefore be mitigated by a simple replacement of the browsers' *innerHTML*-access logic with our own code. The code has been made available to a selected group of security researches in the field, who have been tasked with ensuring its robustness and reliability.
2. The *XMLSerializer* object is a browser component. Therefore, the performance impact is low compared to other methods of pre-processing or filtering *innerHTML*-data before or after mutations take place. We elaborate on the specifics of the performance impact in the 6 Section.

---

<sup>4</sup> OWASP Wiki, [https://www.owasp.org/index.php/OWASP\\_Java\\_HTML\\_Sanitizer\\_Project](https://www.owasp.org/index.php/OWASP_Java_HTML_Sanitizer_Project), Feb. 2013

3. The solution is transparent and does not require additional developer effort, coming down to a single JavaScript implementation. No existing JavaScript or DOM code needs to be modified, the script hooks silently into the necessary property accessors and replaces the insecure browser code. At present, the script works on all modern browsers tested (Internet Explorer, Firefox, Opera and Chrome) and can be extended to work on Internet Explorer 6 or earlier versions.
4. The *XMLSerializer* object post-validates potentially invalid code and thereby provides yet another level of sanitation. That means that even insecure or non-well-formed user-input can be filtered and kept free from mutation XSS and similar attack vectors.
5. The TrueHTML approach is generic, transparent and website-agnostic. This means that a user can utilize this script as a core for a protective browser extension, or apply the user-script to globally protect herself against cause and impact of mutation XSS attacks.

## 6 Evaluation

This section is dedicated to description of settings and dataset used for evaluating the performance penalty introduced by TrueHTML. We focus on assessing the client-side mitigation approach. While HTMLPurifier has been changed to reflect determination for mitigating this class of attacks, the new features are limited to adding items on the internal list of disallowed character combinations. This does not measurably increase the overhead introduced by HTMLPurifier. Performance takes a central stage as a focus of our query, as the transfer overhead introduced by TrueHTML is exceptionally low. The *http archive*<sup>5</sup> has analysed a set of more than 290,000 URLs and over the course of this project it has been determined that the average transfer size of a single web page is more than 1,200 kilobyte, 52kB of which are taken up by HTML content and 214kB by JavaScript. The prototype of TrueHTML is implemented in only 820 byte of code, which we consider to be a negligible transfer overhead.

### 6.1 Evaluation Environment

To assess the overhead introduced by TrueHTML in a realistic scenario, we conducted an evaluation based on the Alexa top 10,000 most popular web sites. We crawled these sites with a recursion depth of one. As pointed out in Section 4, approximately one third of these sites make use of innerHTML. In a next step we determine the performance impact of TrueHTML in a web browser by accessing 5,000 URLs randomly chosen

---

<sup>5</sup> <http://www.httparchive.org/>, Nov. 2012

from this set. Additionally, we assess the performance of TrueHTML in typical usage scenarios, like displaying an e-mail in a web mailer or accessing popular websites, as well as, investigate the relation between page load time overhead and page size in a controlled environment.

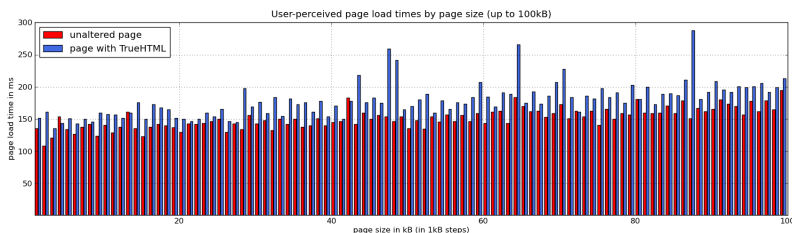
To demonstrate the versatility of the client-side mitigation approach, we used different hardware platforms for the different parts of the evaluation. The Alexa traffic ranking data on virtual machines constituted the grounds for performing this evaluation. Each instance was assigned one core of an Intel Xeon X5650 CPU running at 2.67GHz and had access to 2 GB RAM. The instances ran Ubuntu 12.04 Desktop and Mozilla Firefox 14.0.1. As an example for a mid-range system, we used a laptop with an Intel Core2Duo CPU at 1.86GHz and 2GB RAM, running Ubuntu 12.04 Desktop and Mozilla Firefox 16.0.2, so that to assess the performance in typical usage scenarios.

The evaluation environment is completed by a proxy server to inject TrueHTML into the HTML context of the visited pages. Once a website has been successfully loaded in the browser, we log the URL and the user-perceived page loading time using the Navigation Timing API defined by the W3C Web Performance Working Group [29]. We measure this time as the difference between the time when the `onload` event is fired and the time immediately after the user agent finishes prompting to unload the previous document, as given by the `performance.timing.navigationStart` method.

## 6.2 Evaluation Results

Using the virtual machines we first determine the user-perceived page loading time of the unaltered pages. In a second run we use the proxy server to inject TrueHTML and measure the page loading time again. We calculate the overhead as the increase of page loading time in percentage ratios of the loading time the page needed without TrueHTML. The minimum overhead introduced by TrueHTML is 0.01% while the maximum is 99.94%. On average, TrueHTML introduces an overhead of 30.62%. The median result is 25.73%, the 90th percentile of the overhead is 68.37%. However, the significance of these results is limited as we are unable to control for network-induced delay. In order to eliminate these effects, we conducted the following experiments locally.

Using the laptop, we determined how the user experience is affected by TrueHTML in typical scenarios, e.g. browsing popular webpages. We therefore assigned `document.body.innerHTML` of an otherwise empty DOM to the content of a typical email body of a multipart message (consisting of both the content types `text/plain` and `text/html`), the scraped content of the landing pages of `google.com`, `yahoo.com`, `baidu.com`, `duckduckgo.com`, `youtube.com`, and the scraped content of a map display on Google Maps, as well as of a Facebook profile and a Twitter timeline.



**Fig. 2.** Page load time plotted against page size/#markup elements

Each generated page was accessed three times and the load times logged per criteria described earlier on. The data were generated locally, thus the results do not contain network-induced delays. Table 2 shows the average values.

The results of the previous test show that the user-perceived page load time is not only dependent on the size of the content, but also reliant on the structure and type of the markup. While the data show that in no case the user experience is negatively affected in the typical use cases, this kind of evaluation does not offer a generic insight into how TrueHTML performance overhead relates to content size and the amount of markup elements. To evaluate this in a controlled environment, we generate a single `<p></p>` markup fragment that contains 1kB of text. Again, we assigned `document.body.innerHTML` of an otherwise empty DOM this markup element between one and one hundred times, creating pages containing one element with 1kB text content, scaling up to pages containing one thousand with 1000kB of text content. As before, the data was generated locally. We compare page load times with and without TrueHTML as described above. While the load time increases slightly with size and the amount of markup elements, it can be seen from Figure 2 that the performance penalty introduced through TrueHTML does not raise significantly.

## 7 Related Work

**XSS.** First reported back in the year 2000 [6], Cross-Site Scripting (XSS) attacks gained recognition and attention from a larger audience with the Samy MySpace worm in 2005 [17]. Several types of XSS attacks have been described thus far.

*Reflected XSS*, which typically present a user with an HTML document accessed with maliciously manipulated parameters (GET, HTTP

| Content          | Size w/o TH w/ TH |         |         |
|------------------|-------------------|---------|---------|
| DuckDuckGo       | 8.2 kB            | 336 ms  | 361 ms  |
| Email Body       | 8.5 kB            | 316 ms  | 349 ms  |
| Baidu.com        | 11 kB             | 336 ms  | 466 ms  |
| Facebook profile | 58 kB             | 539 ms  | 520 ms  |
| Google           | 111 kB            | 533 ms  | 577 ms  |
| Youtube          | 174 kB            | 1216 ms | 1346 ms |
| Twitter timeline | 190 kB            | 1133 ms | 1164 ms |
| Yahoo            | 244 kB            | 893 ms  | 937 ms  |
| Google Maps      | 299 kB            | 756 ms  | 782 ms  |

**Table 2.** User-perceived page load times ordered by content size with and without TrueHTML (TH)

header, cookies). These parameters are sent to the server for application logic processing and the document is then rendered along with the injected content.

*Stored XSS*, which is injected into web pages through user-contributed content stored on the server. Without proper processing on the server-side, scripts will be executed for any user that visits a web page with this content.

*DOM XSS*, or *XSS of the third kind*, which was first described by Klein [18]. It may be approached as a type of reflected XSS attack where the processing is done by a JavaScript library within the browser rather than on the server. If the malicious script is placed in the hash part of the URL, it is not even sent to the server, meaning that server-side protection techniques fail in that instance.

Server-side mitigation techniques range from simple character encoding or replacement, to a full rewriting of the HTML code. The advent of DOM XSS was one of the main reasons for introducing XSS filters on the client-side. The IE8 XSS Filter was the first fully integrated solution [25], timely followed by the Chrome XSS Auditor in 2009 [4]. For Firefox, client-side XSS filtering is implemented through the NoScript extension<sup>6</sup>. XSS attack mitigation has been covered in a wide range of publications [5, 8, 9, 16, 26, 35]. Noncespaces [10] use randomized XML namespace prefixes as a XSS mitigation technique, which would make detection of injected content reliable. DSI [23] tries to achieve the same goal based on a classification of HTML content into trusted and untrusted content on the server side, subsequently changing browser parsing behavior to take this distinction into account. Blueprint [21] generates a model of the user input on the server-side and transfers this model, together with

<sup>6</sup> mXSS is mostly not in scope for these, thus remains undetected

the user-contributed content, to the browser; browser behavior is modified by injecting a Javascript library to process the model along with the input. While the method to implement Blueprint in current browsers is remarkably similar to our mitigation approach, it seems hard to exclude the mXSS string from the model as it looks like legitimate content. mXSS attacks are likely to bypass all three of those defensive techniques given that the browser itself is instrumented to create the attack payload from originally benign-looking markup.

**Mutation-based Attacks.** Weinberger et al. [30] give an example where `innerHTML` is used to execute a DOM-based XSS; this is a different kind of attack than those described in this paper, because no mutations are imposed on the content, and the content did not pass the server-side filter. Comparable XSS attacks based on changes to the HTML markup have been initially described for client-side XSS filters. Vela Nava et al. [24] and Bates et al. [4] have shown that the IE8 XSS Filter could once be used to "weaponize" harmless strings and turn them into valid XSS attack vectors by applying a mutation carried out by the regular expressions used by the XSS Filter, thus circumventing server-side protection. Zalewski covers concatenation problems based on NUL strings in `innerHTML` assignments in the *Browser Security Handbook* [34] and later dedicates a section to backtick mutation in his book "The Tangled Web" [33]. Other mutation-based attacks have been reported by Barth et al. [2] and Heiderich [12]. Here, mutation may occur *after* client-side filtering (WebKit corrected a self-closing script tag before rendering, thus activating the XSS vector) or *during* XSS filtering (XSS Auditor strips the `code` attribute value from an applet tag, thus activating a second malicious code source). Hooimeijer et al. describe dangers associated with sanitization of content [15] and claim that they were able, for each of a large number of XSS vectors, to produce a string that would result in that valid XSS vector *after* sanitization. The vulnerabilities described by Kolbitsch et al. may form the basis for an extremely targeted attack by web malware [19]. Those authors state that attack vectors may be prepared for taking into account the mutation behavior of different browser engines. Further, our work can be seen as another justification of the statement from Louw et al. [22]: "The main obstacle a web application must overcome when implementing XSS defenses is the divide between its understanding of the web content represented by an HTML sequence and the understanding web browsers will have of the same".

We show that there is yet another data processing layer in the browser, which managed to remain unknown to the web application up till now. Note that our tests showed that Blueprint would have to be modified to be able to handle prevention of mXSS attacks. The current status of standardization can be retrieved from [20]. Aside from the aforementioned

“print preview problem” referenced in Section 2.2, another early report on XSS vulnerabilities connected to *innerHTML* was filed in 2010 for WebKit browsers by Vela Nava [28]. Further contributions to this problem scope have been submitted by Silin, Hasegawa and others, being subsequently documented on the HTML5 Security Cheatsheet [14].

## 8 Conclusion

The paper describes a novel attack technique based on a problematic and mostly undocumented browser behavior that has been in existence for more than ten years – initially introduced with Internet Explorer 4 and adopted by other browser vendors afterwards. It identifies the attacks enabled by this behavior and delivers an easily implementable solution and protection for web application developers and site-owners. The discussed browser behavior results in a widely usable technique for conducting XSS attacks against applications otherwise immune to HTML and JavaScript injections. These internal browser features transparently convert benign markup, so that it becomes an XSS attack vector once certain DOM properties – such as *innerHTML* and *outerHTML* – are being accessed or other DOM operations are being performed. As we label this kind of attack Mutation-based XSS (mXSS), we dedicate this paper to thoroughly introducing and discussing this very attack. Subsequently, we analyze the attack surface and propose an action plan for mitigating the dangers via several measurements and strategies for web applications, browsers and users. We also supply research-derived evaluations of the feasibility and practicability of the proposed mitigation techniques.

The insight gained from this publication indicates the prevalence of risks and threats caused by the multilayer approach that the web is being designed with. Defensive tools and libraries must gain awareness of the additional processing layers that browsers possess. While server- as well as client-side XSS filters have become highly skilled protection tools to cover and mitigate various attack scenarios, mXSS attacks pose a problem that has yet to be overcome by the majority of the existing implementations. A string mutation occurring during the communication between the single layers of the communication stack from browser to web application and back is highly problematic. Given its place and time of occurrence, it cannot be predicted without detailed case analysis.



## References

1. A. Barth, J. Caballero, and D. Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 360–371. IEEE, 2009.
2. Adam Barth. Bug 29278: XSSAuditor bypasses from [sla.ckers.org](http://sla.ckers.org). [https://bugs.webkit.org/show\\_bug.cgi?id=29278](https://bugs.webkit.org/show_bug.cgi?id=29278).
3. A. Barua, H. Shahriar, and M. Zulkernine. Server side detection of content sniffing attacks. In *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, pages 20–29. IEEE, 2011.
4. Daniel Bates, Adam Barth, and Collin Jackson. Regular expressions considered harmful in client-side XSS filters. In *Proceedings of the 19th international conference on World wide web, WWW '10*, pages 91–100, 2010.
5. Prithvi Bisht and V. N. Venkatakrishnan. XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, 2008.
6. CERT.org. CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests. <http://www.cert.org/advisories/CA-2000-02.html>, 2012.
7. The jQuery Foundation. jQuery: The Write Less, Do More, JavaScript Library. <http://jquery.com/>, November 2012.
8. M.T. Gebre, K.S. Lhee, and M.P. Hong. A robust defense against content-sniffing xss attacks. In *Digital Content, Multimedia Technology and its Applications (IDC), 2010 6th International Conference on*, pages 315–320. IEEE, 2010.
9. Baptiste Gourdin, Chinmay Soman, Hristo Bojinov, and Elie Bursztein. Toward secure embedded web interfaces. In *Proceedings of the Usenix Security Symposium*, 2011.
10. Matthew Van Gundy and Hao Chen. Noncespaces: Using randomization to defeat Cross-Site Scripting attacks. *Computers & Security*, 31(4):612–628, 2012.
11. Yosuke Hasegawa, March 2007.
12. M. Heiderich. *Towards Elimination of XSS Attacks with a Trusted and Capability Controlled DOM*. PhD thesis, Ruhr-University Bochum, 2012.
13. M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk. Scriptless attacks—stealing the pie without touching the sill. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.
14. Mario Heiderich. HTML5 Security Cheatsheet. <http://html5sec.org/>.
15. Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. Fast and precise sanitizer analysis with bek. In *Proceedings of the 20th USENIX conference on Security, SEC'11*, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.
16. Martin Johns. *Code Injection Vulnerabilities in Web Applications - Exemplified at Cross-site Scripting*. PhD thesis, University of Passau, Passau, July 2009.
17. S. Kamkar. *Technical explanation of The MySpace Worm*.
18. Amit Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. *Web Application Security Consortium*, 2005.

19. Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Rozzle: De-Cloaking Internet Malware. In *Proc. IEEE Symposium on Security & Privacy*, 2012.
20. Travis Leithead. Dom parsing and serialization (w3c editor's draft 07 november 2012). <http://dvcs.w3.org/hg/innerhtml/raw-file/tip/index.html>.
21. Mike Ter Louw and V. N. Venkatakrishnan. Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 331–346, Washington, DC, USA, 2009. IEEE Computer Society.
22. Mike Ter Louw and V. N. Venkatakrishnan. Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. *Proc. IEEE Symposium on Security & Privacy*, 2009.
23. Yacin Nadji, Prateek Saxena, and Dawn Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *NDSS*. The Internet Society, 2009.
24. Eduardo Vela Nava and David Lindsay. Abusing Internet Explorer 8's XSS Filters. [http://p42.us/ie8xss/Abusing\\_IE8s\\_XSS\\_Filters.pdf](http://p42.us/ie8xss/Abusing_IE8s_XSS_Filters.pdf).
25. David Ross. IE8 XSS Filter design philosophy in-depth. <http://blogs.msdn.com/b/dross/archive/2008/07/03/ie8-xss-filter-design-philosophy-in-depth.aspx>, April 2008.
26. Prateek Saxena, David Molnar, and Benjamin Livshits. SCRIPTGARD: Automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 601–614. ACM, 2011.
27. Bobby van der Sluis. swfobject - SWFObject is an easy-to-use and standards-friendly method to embed Flash content, which utilizes one small JavaScript file. <http://code.google.com/p/swfobject/>.
28. Eduardo Vela. Issue 43902: innerHTML decompilation issues in textarea. <http://code.google.com/p/chromium/issues/detail?id=43902>.
29. W3C. Navigation Timing. <http://www.w3.org/TR/2012/PR-navigation-timing-20120726/>, July 2012.
30. Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Eui Chul Richard Shin, and Dawn Song. A systematic analysis of xss sanitization in web application frameworks. In *ESORICS*, 2011.
31. Edward Z. Yang. HTML Purifier CSS quoting full disclosure. <http://htmlpurifier.org/>, September 2010.
32. Edward Z. Yang. HTML Purifier. <http://htmlpurifier.org/>, March 2011.
33. M. Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2011.
34. Michal Zalewski. Browser Security Handbook. <http://code.google.com/p/browsersec/wiki/Main>, July 2010.
35. Gavin Zuchlinski. The Anatomy of Cross Site Scripting. *Hitchhiker's World*, 8, November 2003.