A Systematic Analysis of XSS Sanitization in Web Application Frameworks

Joel Weinberger, Prateek Saxena, Devdatta Akhawe,
Matthew Finifter, Richard Shin, and Dawn Song

University of California, Berkeley

Abstract. While most research on XSS defense has focused on techniques for securing existing applications and re-architecting browser mechanisms, sanitization remains the industry-standard defense mechanism. By streamlining and automating XSS sanitization, web application frameworks stand in a good position to stop XSS but have received little research attention. In order to drive research on web frameworks, we systematically study the security of the XSS sanitization abstractions frameworks provide. We develop a novel model of the web browser and characterize the challenges of XSS sanitization. Based on the model, we systematically evaluate the XSS abstractions in 14 major commercially-used web frameworks. We find that frameworks often do not address critical parts of the XSS conundrum. We perform an empirical analysis of 8 large web applications to extract the requirements of sanitization primitives from the perspective of real-world applications. Our study shows that there is a wide gap between the abstractions provided by frameworks and the requirements of applications.

1 Introduction

Cross-site scripting (XSS) attacks are an unrelenting threat to existing and emerging web applications. Major web services such as Google Analytics, Facebook and Twitter have had XSS issues in recent years despite intense research on the subject [34, 52, 61]. Though XSS mitigation and analysis techniques have enjoyed intense focus [6, 7, 12, 13, 33, 36, 37, 39, 41, 43, 44, 47, 49, 50, 59, 64, 66, 68], research has paid little or no attention to a promising sets of tools for solving the XSS riddle—*web application frameworks*—which are gaining wide adoption [18, 21, 22, 28, 35, 42, 48, 55, 58, 69, 71]. Many of these frameworks claim that their sanitization abstractions can be used to make web applications secure against XSS [24, 69]. Though possible in principle, this paper investigates the extent to which it is presently true, clarifies the assumptions that frameworks make, and outlines the fundamental challenges that frameworks need to address to provide comprehensive XSS defense.

Researchers have proposed defenses ranging from purely server-side to browser-based or both [6, 13, 37, 43, 47, 64]. However, *sanitization* or *filtering*, the practice of encoding or eliminating dangerous constructs in untrusted data, remains the industry-standard defense strategy [45]. At present, each web application needs to implement XSS sanitization manually, which is prone to errors [7, 51]. Web frameworks offer a platform to automate sanitization in web applications, freeing developers from existing ad-hoc and error-prone manual analysis. As web applications increasingly rely on web frameworks, we must understand the assumptions web frameworks build on and the security of their underlying sanitization mechanisms.

XSS sanitization is deviously complex; it involves understanding how the web browser parses and interprets web content in non-trivial detail. Though immensely important, this issue has not been fully explained in prior XSS research. For instance, prior research does not detail the security ramifications of the complex interactions between the sub-languages implemented in the browser or the subtle variations in different interfaces for accessing or evaluating data via JavaScript's DOM API. This has important implications on the security of XSS sanitization, as we show through multiple examples in this paper. For instance, we show examples of how sanitization performed on the server-side can be effectively "undone" by the browser's parsing of content into the DOM, which may introduce XSS vulnerabilities in client-side JavaScript code.

A web framework can address XSS using sanitization if it correctly addresses all the subtleties. Whether existing frameworks achieve this goal is an important question and a subject of this paper. A systematic study of today's web frameworks should evaluate their security and assumptions along the following dimensions to quantify their benefits:

- **Context Expressiveness**. Untrusted data needs to be sanitized differently based on its *context* in the HTML document. For example, the sanitization requirements of a URI attribute are different from those of an HTML tag. *Do web frameworks provide sanitizers for different contexts that applications commonly use in practice?*
- **Auto-sanitization and Context-sensitivity**. Applying sanitizers in code automatically, which we term *auto-sanitization*, shifts the burden of ensuring safety against XSS from developers to frameworks. However, a sanitizer that may be safe for use in one context may be unsafe for use in another. Therefore, to achieve security, auto-sanitization must be *context-sensitive*; otherwise, as we explain in Section 3.1, it may provide a false sense of security. *To what extent do modern web frameworks offer context-sensitive auto-sanitization?*
- **Security of dynamic client-side evaluation**. AJAX applications have significant client-side code components, such as in JavaScript. There are numerous subtleties in XSS sanitization because client-side code may read values from the DOM. *Do frameworks support complete mediation on DOM accesses in client-side code?*

Contributions and Approach. We explain the challenges inherent in XSS sanitization. We present a novel model of the web browser's parsing internals in sufficient detail to explain the subtleties of XSS sanitization. Our model is the first to comprehensively conceptualize the difficulties of sanitization. Our browser model includes details of the sub-languages supported by HTML5, their internal interactions, and the transductions browsers introduce on content. We provide examples of XSS scenarios that result.

This paper is a first step towards initiating research on secure web frameworks. It systematically identifies the features and pitfalls in XSS sanitization abstractions of today's web frameworks and the challenges a secure framework must address. We compare existing abstractions in frameworks to the requirements of web applications, which we derive by an empirical analysis. We study 14 mature, commercially used web frameworks and 8 popular open-source web applications. We establish whether the applications we study could be migrated to use the abstractions of today's web frameworks. We quantify the security of the abstractions in frameworks and clarify the liability developers will continue to take even if they were to migrate their applications to today's frameworks. We provide the first in-depth study of the gap between the sanitization abstractions provided by web frameworks and what web applications require for safety against XSS. We conclude that though web frameworks have the potential to secure applications against XSS, most existing frameworks fall short of achieving this goal.

2 A Systematic Browser Model for XSS

We formulate XSS with a comprehensive model of the browser's parsing behavior in Section 2.1. We discuss the challenges and subtleties XSS sanitization must address in Section 2.2, and how web frameworks could offer a potential solution in Section 2.3. We outline our evaluation objectives and formulate the dimensions along which we empirically measure the security of web frameworks in Section 2.4.

2.1 Problem Formulation: XSS Explained

Web applications mix control data (code) and content in their output, generated by server-side code, which is consumed as client-side code by the web browser. When data controlled by the attacker is interpreted by the web browser as if it was code written by the web developer, an XSS attack results. A canonical example of an XSS attack is as follows. Consider a blogging web application that emits untrusted content, such as anonymous comments, on the web page. If the developer is not careful, an attacker can input text such as `<script>...<script>`, which may be output verbatim in the server's output HTML page. When a user visits this blog page, her web browser will execute the attacker controlled text as script code.

XSS sanitization requires removal of such dangerous tags from the untrusted data. Unfortunately, not all cases are as simple as this `<script>` tag example. In the rest of this section, we identify browser features that make preventing XSS much more complicated. Previous research has indicated that this problem is complex, but we are not aware of an in-depth, systematic problem formulation.

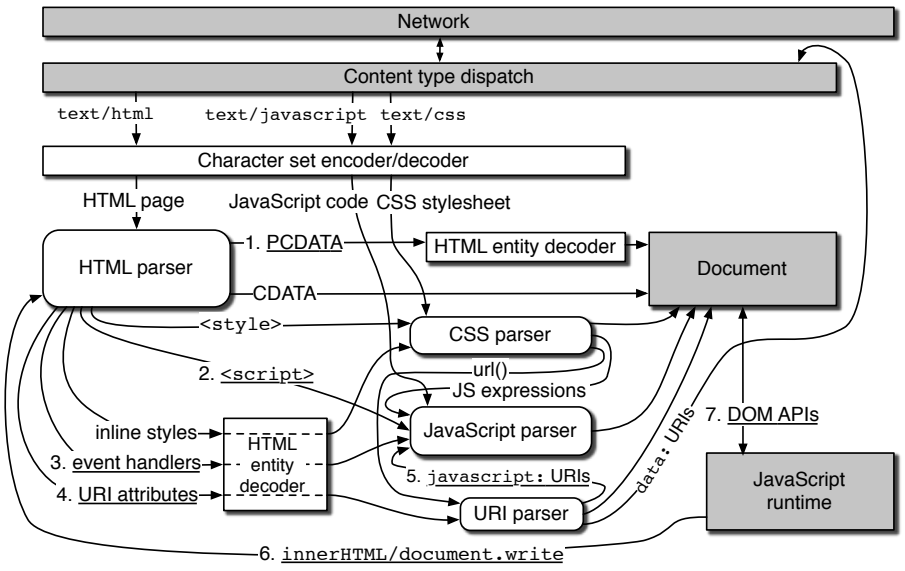

Fig. 1. Flow of Data in our Browser Model. Certain contexts such as `PCDATA` and `CDATA` directly refer to parser states in the HTML 5 specification. We refer to the numbered and underlined edges during our discussion in the text.

The Browser Model. We present a comprehensive model of the web browser's parsing behavior. While the intricacies of browser parsing behavior have been discussed before [70], a formal model has not been built to fully explore its complexity. We show this model in Figure 1. Abstractly, the browser can be viewed as a collection of HTML-related sub-grammars and a collection of transducers. Sub-grammars correspond to parsers for languages such as URI schemes, CSS, HTML, and JavaScript (the rounded rectangles in Figure 1). Transducers transform or change the representation of the text, such as in HTML-entity encoding/decoding, URI-encoding, JavaScript Unicode encoding and so on (the unshaded rectangles in Figure 1). The web application's output, i.e., HTML page, is input into the browser via the network; it can be directly fed into the HTML parser after some pre-processing or it can be fed into JavaScript's HTML evaluation constructs. The browser parses these input fragments in stages—when a fragment is recognized as a term in another sub-grammar, it is shipped to the corresponding sub-grammar for reparsing and evaluation (e.g., edge 2). For example, while the top-level HTML grammar identifies an anchor (`<a>`) tag in the HTML document, the contents of the `href` attribute are sent to the URI parser (edge 4). The URI parser handles a `javascript:` URI by sending its contents to the JavaScript parser (edge 3), while other URIs are sent to their respective parsers.

2.2 Subtleties and Challenges in XSS Sanitization

The model shows that the interaction between sub-components is complex; burdening developers with fully understanding their subtleties is impractical. We now describe a number of such challenges that correct sanitization-based defense needs to address.

Challenge 1: Context Sensitivity. Sanitization for XSS defense requires knowledge of where untrusted input appears structurally and semantically in the web application. For example, simple HTML-entity encoding is a sufficient sanitization procedure to neutralize XSS attacks when is placed inside the body of an HTML tag, or, in the `PCDATA` (edge 1) parsing context, as defined by HTML5 [30]. However, when data is placed in a resource URI, such as the `src` or `href` attribute of a tag, HTML-encoding is insufficient to block attacks such as via a `javascript:` URI (edge 4 and 5). We term the intuitive notion of *where* untrusted data appears as its *context*. Sanitization requirements vary by contexts. Frameworks providing sanitization primitives need to be mindful of such differences from context to context. The list of these differences is large [29].

Challenge 2: Sanitizing nested contexts. We can see in the model that a string in a web application's output can be parsed by multiple sub-parsers in the browser. We say that such a string is placed in *nested contexts*. That is, its interpretation in the browser will cause the browser to traverse more than one edge shown in Figure 1.

Sanitizing for nested contexts adds its own complexity. Consider an embedding of an untrusted string inside a script block, such as `<script> var x = ' UNTRUSTED DATA'...</script>`. In this example, when the underlined data is read by the browser, it is simultaneously placed in two contexts. It is placed in a JavaScript string literal context by the JavaScript parser (edge 2) due to the single quotes. But, before that, it is inside a `<script>` HTML tag (or `RCDATA` context according to the HTML 5 specification) that is parsed by the HTML parser. Two distinct attack vectors can be used here: the attacker could use a single quote to break out of the JavaScript string context, or inject `</script>` to break out of the script tag. In fact, sanitizers commonly fail to account for the latter because they do not recognize the presence of nested contexts.

Challenge 3: Browser Transductions. If dealing with multiple contexts is not arduous enough, our model highlights the *implicit transductions* that browsers perform when handing data from one sub-parser to another. These are represented by edges from rounded rectangles to unshaded rectangles in Figure 1. Such transductions and browser-side modifications can, surprisingly, *undo* sanitization applied on the server.

Consider a blog page in which comments are hidden by default and displayed only after a user clicks a button. The code uses an `onclick` JavaScript event handler:

```
<div class='comment-box'onclick='displayComment("UNTRUSTED",this)'>
   ... hidden comment ... </div>
```

The underlined untrusted comment is in two nested contexts: the HTML attribute and single-quoted JavaScript string contexts. Apart from preventing the data from escaping out of the two contexts separately (Challenge 2), the sanitization must worry about an additional problem. The HTML 5 standard mandates that the browser HTML-entity decode an attribute value (edge 3) before sending it to a sub-grammar. As a result, the attacker can use additional attack characters even if the sanitization performs HTML-entity encoding to prevent attacks. The characters `&quot;` will get converted to `"` before being sent to the JavaScript parser. This will allow the untrusted comment to break out of the string context in the JavaScript parser. We call such implicit conversions *browser transductions*. Full details of the transductions are available in Appendix A.

Challenge 4: Dynamic Code Evaluation. In principle, the chain of edges traversed by the browser while parsing a text can be arbitrarily long because the browser can dynamically evaluate code. Untrusted content can keep cycling through HTML and JavaScript contexts. For example, consider the following JavaScript code fragment:

```
function foo(untrusted) {
    document.write("<input onclick='foo(" + untrusted + ")' >");
}
```

Since `untrusted` text is repeatedly pumped through the JavaScript string and HTML contexts (edges 3 and 6 of Figure 1), statically determining the context traversal chain on the server is infeasible. In principle, purely server-side sanitization is not sufficient for context determination because of dynamic code evaluation. Client-side sanitization is needed in these cases to fully mitigate potential attacks. Failure to properly sanitize such dynamic evaluation leads to the general class of attacks called DOM-based XSS or client-side code injection [60]. In contrast to Challenges 2 and 3, such vulnerabilities are caused not by a lack of understanding the browser, but of frameworks not understanding application behavior.

Another key observation is that browser transductions along the edges of Figure 1 vary from one edge to another, as detailed in Appendix A. This mismatch can cause XSS vulnerabilities. During our evaluation, we found one such bug (Section 3.2). We speculate that JavaScript-heavy web applications are likely to have such vulnerabilities.

Challenge 5: Character-set Issues. Successfully sanitizing a string at the server side implicitly requires that the sanitizer and the browser are using the same character set while working with the string. A common source of XSS vulnerabilities is a mismatch in the charset assumed by the sanitizer and the charset used by the browser. For example, the ASCII string `+ADw-` does not have any suspicious characters. But when interpreted by the browser as UTF-7 character-set, it maps to the dangerous `<` character: this mismatch between the server-side sanitization and browser character set selection has led to multiple XSS vulnerabilities [62].

Challenge 6: MIME-based XSS, Universal XSS, and Mashup Confinement. Browser quirks, especially in interpreting content or MIME types [10], contribute their own share of XSS vulnerabilities. Similarly, bugs in browser implementations, such as capability leaks [26] and parsing inconsistencies [9], or in browser extensions [11] are important components of the XSS landscape. However, these do not pertain to sanitization defenses in web frameworks. Therefore, we consider them to be out-of-scope for this study.

2.3 The Role of Web frameworks

Web application development frameworks provide components to enable typical work flows in web application development. These frameworks can abstract away repetitive and complex tasks, freeing the developer to concentrate on his particular scenario. Consider session management, a common feature that is non-trivial to implement securely. Most web application frameworks automate session management, hiding this complexity from the developer. Similarly, web application frameworks can streamline and hide the complexity of XSS sanitization from the developer. In fact, increased security is often touted as a major benefit of switching to web application frameworks [24, 69].

Frameworks can either provide XSS sanitization routines in a library or they can automatically add appropriate sanitization code to a web application. We term the latter approach *auto-sanitization*. In the absence of auto-sanitization, the burden of calling the sanitizers is on the developer, which we have seen is an error-prone requirement. On the other hand, auto-sanitization, if incorrectly implemented, can give a false sense of security because a developer may defer all sanitization to this mechanism.

2.4 Analysis Objectives

In theory, use of a web application framework should free the developer from the complexities of XSS sanitization as discussed earlier and illustrated in Figure 1. If true, this requires the framework to grapple with all these complexities instead. We abstract the most important challenges into the following three dimensions:

- **Context Expressiveness and Sanitizer Correctness.** As we detailed in Challenge 1, sanitization requirements change based on the context of the untrusted data. We are interested in investigating the set of contexts in which untrusted data is used by web applications, and whether web frameworks support those contexts. In the absence of such support, a developer will have to revert to manually writing sanitization functions. The challenges outlined in Section 2.1 make manually developing *correct* sanitizers a non-starter. Instead, we ask, *do web frameworks provide correct sanitizers for different contexts that web applications commonly use in practice?*
- **Auto-sanitization and Context-Sensitivity.** Providing sanitizers is only a small part of the overall solution necessary to defend against XSS attacks. Applying sanitizers in code automatically, which we term *auto-sanitization*, shifts the burden of ensuring safety against XSS from developers to frameworks. The benefit of this is self-evident: performing correct sanitization in framework code spares each and every developer from having to implement correct sanitization himself, and from having to remember to perform that sanitization everywhere it should be performed. Furthermore, correct auto-sanitization needs to be context-sensitive—context-insensitive auto-sanitization can lead to a false sense of security. *Do web frameworks offer auto-sanitization, and if so, is it context-sensitive?*

– **Security of client-side code evaluation.** Much of the research on XSS has focused on the subtleties of parsing in HTML contexts across browsers. But AJAX web applications have significant client-side code components, such as in JavaScript. There are numerous subtleties in XSS sanitization because client-side code may read values from the DOM. Sanitization performed on the server-side may be "undone" during the browser's parsing of content into the DOM (Challenge 3 and Challenge 4). *Do frameworks support complete mediation on DOM accesses in client-side code?*

In this study, we focus solely on XSS sanitization features in web frameworks and ignore all other framework features. We also do not include purely client-side frameworks such as jQuery [1] because these do not provide XSS protection mechanisms. Additionally, untrusted data used in these libraries also needs server-side sanitization.

3 Analysis of Web Frameworks and Applications

In this section, we empirically analyze web frameworks and the sanitization abstractions they provide. We show that there is a mismatch in the abstractions provided by frameworks and the requirements of applications.

We begin by analyzing the "auto-sanitization" feature—a security primitive in which web frameworks sanitize untrusted data automatically—in Section 3.1. We identify the extent to which it is available, the pitfalls of its implementation, and whether developers can blindly trust this mechanism if they migrate to or develop applications on existing auto-sanitizing frameworks. We then evaluate the support for dynamic code evaluation via JavaScript in frameworks in Section 3.2. In the previous section, we identified subtleties in the browser's DOM interface. In Section 3.2, we discuss whether applications adequately understand it to prevent XSS bugs.

Frameworks may not provide auto-sanitization, but instead may provide sanitizers that developers can manually invoke. Arguably, the sanitizers implemented by frameworks would be more robust than the ones implemented by the application developer. We evaluate the breadth of contexts for which each framework provides sanitizers, or the *context expressiveness* of each framework, in Section 3.3. We also compare it to the requirements of the applications we study today to evaluate whether this expressiveness is enough for real-world applications.

Finally, we evaluate frameworks' assumptions regarding correctness of sanitization and compare these to the sanitization practices in security-conscious applications.

Methodology and Analysis Subjects. We examine 14 popular web application frameworks in commercial use for different programming languages and 8 popular PHP web applications ranging from 19 KLOC to 532 KLOC in size. We used a mixture of manual and automated exploration to identify sanitizers in the web application running on an instrumented PHP interpreter. We then executed the application again along paths that use these sanitization functions and parsed the outputs using an HTML 5-compliant browser to determine the contexts for which they sanitize. Due to space constraints, this paper focuses solely on the results of our empirical analysis. A technical report provides the full details of the techniques employed [65].

3.1 Auto-Sanitization: Features and Pitfalls

Auto-sanitization is a feature that shifts the burden of ensuring safety against XSS from the developer to the framework. In a framework that includes auto-sanitization, the ap-

Language	Framework, Plugin, or Feature	Automatically Sanitizes in HTML Context	Performs Context-Aware Sanitization	Pointcut
PHP	CodeIgniter	•		Request Reception
VB, C#, C++, F#	ASP.NET Request Validation [5]	•		Request Reception
Ruby	xss_terminate Rails plugin [67]	•		Database Insertion
Python	Django	•		Template Processing
Java	GWT SafeHtml	•	•	Template Processing
C++	Ctemplate	•	•	Template Processing
Language-neutral	ClearSilver	•	•	Template Processing

Table 1. Extent of automatic sanitization support in the frameworks we study and the pointcut (set of points in the control flow) where the automatic sanitization is applied.

plication developer is responsible for indicating which variables will require sanitization. When the page is output, the web application framework can then apply the correct sanitizer to these variables. Our findings, summarized in Table 1, are as follows:

- Of the 14 frameworks evaluated, only 7 support some form of auto-sanitization.
- 4 out of the 7 auto-sanitization framework apply a "one-size-fits-all" strategy to sanitization. That is, they apply the same sanitizer to all flows of untrusted data irrespective of the context into which the data flows. We call this *context-insensitive* sanitization, which is fundamentally unsafe, as explained later.
- We measure the fraction of application output sinks actually protected by context-insensitive auto-sanitization mechanism in 10 applications built on Django, a popular web framework. Table 2 presents our findings. The mechanism fails to correctly protect between 14.8% and 33.6% of an application's output sinks.
- Only 3 frameworks perform context-sensitive sanitization.

No auto-sanitization. Only half of the studied frameworks provide any auto-sanitization support. In those that don't, developers must deal with the challenges of selecting where to apply built-in or custom sanitizers. Recent studies have shown that this manual process is prone to errors, even in security-audited applications [25, 51]. We also observed instances of this phenomenon in our analysis. The following example is from a Django application called GRAMPS.

Example 1

```
{% if header.sortable %}
    <a href="{{header.url|escape}}">
{% endif %}
```

The developer sanitizes a data variable placed in the `href` attribute but uses the HTML-entity encoder (`escape`) to sanitize the data variable `header.url`. This is an instance of Challenge 2 outlined in Section 2. In particular, this sanitizer fails to prevent XSS attack vectors such as `javascript:` URIs.

Insecurity of Context-insensitive auto-sanitization. Another interesting fact about the above example is that even if the developer relied on Django's default auto-sanitization, the code would be vulnerable to XSS attacks. Django employs context-insensitive auto-sanitization, i.e., it applies the same sanitizer (`escape`) irrespective of the output context. `escape`, which does an HTML entity encode, is safe for use in HTML tag context but unsafe for other contexts. In the above example, applying `escape`, automatically or otherwise, fails to protect against XSS attacks. Auto-sanitization support in Rails [67], .NET (request validation [5]) and CodeIgniter are all context-insensitive and have similar problems.

Context-insensitive auto-sanitization provides a false sense of security. On the other hand, relying on developers to pick a sanitizer consistent with the context is error-prone,

Web Application	No. Sinks	% Auto-sanitized Sinks	% Sinks not sanitized (marked safe)	% Sinks manually sanitized	% Sinks in HTML Context	% Sinks in URI Attr. (excl. scheme)	% Sinks in URI Attr. (incl. scheme)	% Sinks in JS Attr. Context	% Sinks in JS Number or String Context	% Sinks in Style Attr. Context
GRAMPS Genealogy Management	286	77.9	0.0	22.0	66.4	3.4	30.0	0.0	0.0	0.0
HicroKee's Blog	92	83.6	7.6	8.6	83.6	6.5	7.6	1.0	0.0	1.0
FabioSouto.eu	55	90.9	9.0	0.0	67.2	7.2	23.6	0.0	1.8	0.0
Phillip Jones' Eportfolio	94	92.5	7.4	0.0	73.4	11.7	12.7	0.0	2.1	0.0
EAG cms	19	94.7	5.2	0.0	84.2	0.0	5.2	0.0	0.0	10.5
Boycott Toolkit	347	96.2	3.4	0.2	71.7	1.1	25.3	0.0	1.7	0.0
Damned Lies	359	96.6	3.3	0.0	74.6	0.5	17.8	0.0	0.2	6.6
oebfare	149	97.3	2.6	0.0	85.2	6.0	8.0	0.0	0.0	0.6
Malaysia Crime	235	98.7	1.2	0.0	77.8	0.0	1.7	0.0	20.4	0.0
Philippe Marichal's web site	13	100.0	0.0	0.0	84.6	0.0	15.3	0.0	0.0	0.0

Table 2. Usage of auto-sanitization in Django applications. The first 2 columns are the number of sinks in the templates and the percentage of these sinks for which auto-sanitization has not been disabled. Each remaining column shows the percentage of sinks that appear in the given context.

and one XSS hole is sufficient to subvert the web application's integrity. Thus, because it covers some limited cases, context-insensitive auto-sanitization is better protection than no auto-sanitization.

We measure the percentage of output sinks protected by context-insensitive auto-sanitization in 10 Django-based applications that we randomly selected for further investigation [23]. We statically correlated the automatically applied sanitizer to the context of the data; the results are in Table 2. The mechanism protects between 66.4% and 85.2% of the output sinks, but conversely permits XSS vectors in 14.8% to 33.6% of the contexts, subject to whether attackers control the sanitized data or not. We did not determine the exploitability of these incorrectly auto-sanitized cases, but we observed that in most of these cases, developers resorted to custom manual sanitization. An auto-sanitization mechanism that requires developers to sanitize diligently is self-defeating. Developers should be aware of this responsibility when building on such a mechanism.

Context-Sensitive Sanitization. Context-sensitive auto-sanitization addresses the above issues. Three web frameworks, namely GWT, Google Clearsilver, and Google Ctemplate, provide this capability. In these frameworks, the auto-sanitization engine performs runtime parsing, keeping track of the context before emitting untrusted data. The correct sanitizer is then automatically applied to untrusted data based on the tracked context. These frameworks rely on developers to identify untrusted data. The typical strategy is to have developers write code in *templates*, which separate the HTML content from the (untrusted) data variables. For example, consider the following simple template supported by the Google Ctemplate framework:

Example 2

```
{{%AUTOESCAPE context="HTML"}}
<html><body><script> function showName() {
document.getElementById("sp1").textContent = "Name: {{NAME}}";} </script>
<span id="sp1" onclick="showName()">Click to display name.</span><br/>
Homepage: <a href="{{URI}}"> {{PAGENAME}} </a></body></html>
```

Variables that require sanitization are surrounded by `{{` and `}}`; the rest of the text is HTML content to be output. When the template executes, the engine parses the output and determines that `{{NAME}}` is in a JavaScript string context and automatically applies the sanitizer for the JavaScript string context, namely `:javascript_escape`. For other variables, the same mechanism applies the appropriate sanitizers. For instance, the variable `{{URI}}` is sanitized with the `:url_escape_with_arg=html` sanitizer.

3.2 Security of Client-side Code Evaluation

In Section 2, we identified subtleties of dynamic evaluation of HTML via JavaScript's DOM API (Challenge 4). The browser applies different transductions depending on the DOM interface used (Challenge 3 and listed in Appendix A). Given the complexity of sanitizing dynamic evaluation, we believe web frameworks should provide support for this important class of XSS attack vectors too. Ideally, a web framework could incorporate knowledge of these subtleties, and provide automatic sanitization support during JavaScript code execution.

Support in web frameworks. The frameworks we studied do not support sanitization of dynamic flows. Four frameworks support sanitization of untrusted data used in a JavaScript string or number context (Table 3). This support is only *static*: it can ensure that untrusted data doesn't escape out during the parsing by the browser, but such sanitization can't offer any safety during dynamic code evaluation, given that dynamic code evaluation can *undo* previously applied transductions (Challenge 4).

Context-insensitivity issues with auto-sanitization also extend to JavaScript code. For example, Django uses the context-insensitive HTML-escape sanitizer even in JavaScript string contexts. Dangerous characters (e.g., `\n,\r,;`) can still break out of the JavaScript string literal context. For example, in the Malaysia Crime Application (authored in Django), the `crime.icon` variable is incorrectly auto-sanitized with HTML-entity encoding and is an argument to a JavaScript function call.

Example 3

```
map.addOverlay(new GMarker(point, {{ crime.icon }}))
```

Awareness in Web Applications. DOM-based XSS is a serious problem in web applications [49, 50]. Recent incidents in large applications, such as vulnerabilities in Google optimizer [46] scripts and Twitter [60], show that this is a continuing problem. This suggests that web applications are not fully aware of the subtleties of the DOM API and dynamic code evaluation constructs (Challenge 3 and 4 in Section 2).

To illustrate this, we present a real-world example from one of the applications we evaluated, `phpBB3`, showing how these subtleties may be misunderstood by developers.

Example 4

```
text = element.getAttribute('title');
// ... elided ...
desc = create_element('span', 'bottom');
desc.innerHTML = text;
tooltip.appendChild(desc);
```

In the server-side code, which is not shown here, the application sanitizes the `title` attribute of an HTML element by HTML-entity encoding it. If the attacker enters a string like `<script>`, the encoding converts it to `&lt;script&gt;`. The client-side code subsequently reads this attribute via the `getAttribute` DOM API in JavaScript code

Language	Framework	HTML tag content or non-URI attribute	URI Attribute (excluding scheme)	URI Attribute (including scheme)	JS String	JS Number or Boolean	Style Attribute or Tag
Perl	Mason [2, 42]	•	•				
	Template Toolkit [58]	•	•				
	Jifty [35]	•	•				
PHP	CakePHP [15]	•	•				
	Smarty Template Engine [55]	•	•		•		
	Yii [32, 69]	•	•				
	Zend Framework [71]	•	•				
	CodeIgniter [19, 20]	•	•				
VB, C#, C++, F#	ASP.NET [4]	•	•				
Ruby	Rails [48]	•	•				
Python	Django [22]	•	•	•	•		
Java	GWT SafeHtml [28]	•	•	•			
C++	Ctemplate [21]	•	•	•	•	•	•
Language-neutral	ClearSilver [18]	•	•	•	•		•

Table 3. Sanitizers provided by languages and/or frameworks. For frameworks, we also include sanitizers provided by standard packages or modules for the language.

(shown above) and inserts it back into the DOM via the `innerHTML` method. The vulnerability is that the browser automatically decodes HTML entities (through edge 1 in Figure 1) while constructing the DOM. This effectively undoes the server's sanitization in this example. The `getAttribute` DOM API reads the decoded string (e.g., `<script>`) from the DOM (edge 7). Writing `<script>` via `innerHTML` (edge 6) results in XSS.

This bug is subtle. Had the developer used `innerText` instead of `innerHTML` to write the data, or used `innerHTML` to read the data, the code would *not* be vulnerable. The reason is that the two DOM APIs discussed here read different serializations of the parsed page, as explained in Appendix A.

The prevalence of DOM-based XSS vulnerabilities and the lack of framework support suggest that this is a challenge for web applications and web frameworks alike. Libraries such as Caja and ADsafe model JavaScript and DOM manipulation but target isolation-based protection such as authority safety, not DOM-based XSS [3, 14]. Protection for this class of XSS requires further research.

3.3 Context Expressiveness

Having analyzed the auto-sanitization support in web frameworks for static HTML evaluation as well as dynamic evaluation via JavaScript, we turn to the support for manual sanitization. Frameworks may not provide auto-sanitization but instead may provide sanitizers which developers can call. This improves security by freeing the developer from (re)writing complex, error-prone sanitization code. In this section, we evaluate the breadth of contexts for which each framework provides sanitizers, or the *context expressiveness* of each framework. For example, a framework that provides built-in sanitizers for more than one context, say in URI attributes, CSS keywords, JavaScript string contexts, is more expressive than one that provides a sanitizer only for HTML tag context.

Expressiveness of Framework Sanitization Contexts. Table 3 presents the expressiveness of web frameworks we study and Table 4 presents the expressiveness required by our subject web applications. The key insights are:

– We observe that 9 out of the 14 frameworks do not support contexts other than the HTML context (e.g., as the content body of a tag or inside a non-URI attribute) and

Application	Description	LOC	HTML Context	URI Attr. (excl. scheme)	URI Attr. (incl. scheme)	JS Attr. Context	JS Number or String Context	No. Sanitizers	No. Sinks
RoundCube	IMAP Email Client	19,038	•	•	•	•	•	30	75
Drupal	Content Management System	20,995	•	•	•	•	•	32	2557
Joomla	Content Management System	75,785	•	•	•	•		22	538
WordPress	Blogging Application	89,504	•	•	•	•		95	2572
MediaWiki	Wiki Hosting Application	125,608	•	•	•	•	•	118	352
PHPBB3	Bulletin Board Software	146,991	•	•	•	•	•	19	265
OpenEMR	Medical Records Management	150,384	•			•	•	18	727
Moodle	E-Learning Software	532,359	•	•	•	•	•	43	6282

Table 4. The web applications we study and the contexts for which they sanitize.

the URI attribute context. The most common sanitizers for these are HTML entity encoding and URI encoding, respectively.

- 4 web frameworks, ClearSilver, Ctemplate, Django, and Smarty, provide appropriate sanitization functions for emitting untrusted data into a JavaScript string. Only 1 framework, Ctemplate, provides a sanitizer for emitting data into JavaScript outside of the string literal context. However, the sanitizer is a restrictive whitelist, allowing only numeric or boolean literals. No framework we studied allows untrusted JavaScript code to be emitted into JavaScript contexts. Supporting this requires a client-side isolation mechanism such as ADsafe [3] or Google's Caja [14].
- 4 web frameworks, namely Django, GWT, Ctemplate, and Clearsilver, provide sanitizers for URI attributes in which a complete URI (i.e., including the URI protocol scheme) can be emitted. These sanitizers reject URIs that use the `javascript:` scheme and accept only a whitelist of schemes, such as `http:`.
- Of the frameworks we studied, we found only one that provides an interface for customizing the sanitizer for a given context. Yii uses HTML Purifier [32], which allows the developer to specify a custom list of allowed tags. For example, a developer may specify a policy that allows only `<b>` tags. The other frameworks (even the context-sensitive auto-sanitizing ones) have sanitizers that are not customizable. That is, untrusted content within a particular context is always sanitized the same way. Our evaluation of web applications strongly invalidates this assumption, showing that applications often sanitize data occurring in the same context differently based on other attributes of the data.

The set of contexts for which a framework provides sanitizers gives a sense of how the framework expects web applications to behave. Specifically, frameworks assume applications will not emit sanitized content into multiple contexts. More than half of the frameworks we examined do not expect web applications to insert content with arbitrary schemes into URI contexts, and only one of the frameworks supports use of untrusted content in JavaScript `Number` or `Boolean` contexts. Below, we challenge these assumptions by quantifying the set of contexts for which applications need sanitizers.

Expressiveness of Contexts and Sub-Context Variance in Web Applications. We examined our 8 subject PHP applications, ranging from 19 to 532 KLOC, to understand what expressiveness they require and whether they could, theoretically, migrate to the existing frameworks. We systematically measure and enumerate the contexts into which these applications emit untrusted data. Table 4 shows the result of this evaluation. We observe that nearly all of the applications insert untrusted content into all of the outlined contexts. Contrast this with Table 3, where most frameworks support a much more limited set of contexts with built-in sanitizers.

More surprisingly, we find that applications often employ more than one sanitizer for each context. That is, an abstraction that ties a single sanitizer to a given context may be insufficient. We term this variation in sanitization across code paths *sub-context variance*. Sub-context variance evidence suggests that directly migrating web applications to web frameworks' (auto-) sanitization support may not be directly possible given that even context-sensitive web frameworks rigidly apply one sanitizer for a given context.

Sub-context variance is particularly common in the form of *role-based* sanitization, where the application applies different sanitizers based on the privilege of the user. We found that it is common to have a policy in which the site administrator's content is subject to no sanitization (by design). Examples include phpBB, WordPress, Drupal. For such simple policies, there are legitimate code paths that have no sanitization requirements. To illustrate this, we present a real-world example from the popular WordPress application which employs different sanitization along different code paths.

Example 5

WordPress, the popular blogging application, groups users into *roles*. A user in the author role can create a new post on the blog with most non-code tags permitted. An anonymous commenter, on the other hand, can only use a small number of text formatting tags. In particular, the latter cannot insert images in comments while an author can insert images in his post. Note that neither can insert `<script>` tags, or any other active content. In both cases, untrusted input flows into HTML tag context, but the sanitizer applied changes as a function of the user role.

Most auto-sanitizing frameworks do not support such rich abstractions to support auto-sanitization specifications at a sub-context granularity. Nearly all sanitization libraries (not part of web frameworks) are customizable. However, their connection to special role-based sanitization (or similar cases) are not supported presently. We believe that web frameworks can fill this gap. Only 1 framework, Yii, provides the flexibility to handle such customizations using the HTMLPurifier sanitization library. Unfortunately, Yii only provides this flexibility for the HTML tag context.

3.4 Enabling Reasoning of Sanitizer Correctness

Prior research on web applications has shown that developing sanitization functions, especially custom sanitizers, is tricky and prone to errors [7]. We investigate how the sanitizers in web frameworks handle this issue. We compare the structure of the sanitizers used in frameworks to the structure we observe in our subject applications and characterize the ground assumptions that developers should be aware of.

Blacklists vs. Whitelists. We find that most web frameworks structure their sanitizers as a *declarative-style whitelist* of code constructs explicitly allowed in untrusted content. For instance, one sanitization library employed in the Yii is HTML-Purifier [32], which permits a declarative list of HTML elements like event attributes of special tags in untrusted content. All of the web applications we studied also employ this whitelisting mechanism, such as the KSES library used in Wordpress [38]. Such sanitizers assume that the whitelist is only contains a well understood and safe subset of the language specification, and does not permit any unsafe structures.

In contrast, we find that only 1 subject web framework, viz. CodeIgniter, employs a blacklist-based sanitization approach. Even if one verifies that all the elements on a blacklist conform to an unsafe subset of the language specification, the sanitizer may

still allow unsafe parts of the language. For example, CodeIgniter's `xss_clean` function removes a blacklist of potentially dangerous strings like `document.cookie` that may appear in any context. Even if it removes *all* references to `document.cookie`, there still may be other ways for attacker code to reference cookies, such as via `document['cookie']`.

Correctness of the sanitizers used is fundamental to the safety of the sanitization based strategy used in web frameworks. Based on the above examples, we claim that it is easier to verify that a whitelist policy is safe and recommend frameworks adopt such a strategy.

HTML Canonicalization. Essential to the safety of sanitization-based defense is that the user's browser parse the untrusted string in a manner consistent with the parsing applied by the sanitizer. For instance, if the context-determination in the frameworks differs from the actual parsing in the browser, the wrong sanitizer could be applied by the framework.

We observe that frameworks employ a *canonicalization* strategy to ensure this property; the web frameworks identify a 'canonical' subset of HTML-related languages into which all application output is generated. The assumption they rely on is that this canonical form parses the same way across major web browsers. We point out explicitly that these assumptions are not systematically verified today and, therefore, framework outputs may still be susceptible to XSS attacks. For example, a recent XSS vulnerability in the HTML Purifier library (used in Yii) was traced back to "quirks in Internet Explorer's parsing of string-like expressions in CSS [31]."

Finally, we point out that sanitization-based defense isn't the only alternative—proposals for *sanitization-free* defenses, such as DSI [43], BLUEPRINT [59] and the Content Security Policy [56] have been presented. Future frameworks could consider these. Verifying the safety of the whitelist-based canonicalization strategy and its assumptions also deserves research attention.

4 Related Work

XSS Analysis and Defense. Much of the research on cross-site scripting vulnerabilities has focused on finding XSS flaws in web applications, specifically on server-side code [7, 33, 36, 39–41, 44, 66, 68] but also more recently on JavaScript code [8, 27, 49, 50]. These works have underscored the two main causes of XSS vulnerabilities: *identifying untrusted data* at output and *errors in sanitization* by applications. There have been three kinds of defenses: purely server-side, purely browser-based, and those involving both client and server collaboration.

BLUEPRINT [59], SCRIPTGARD [51] and XSS-GUARD [13] are three server-side solutions that have provided insight into context-sensitive sanitization. In particular, BLUEPRINT provides a deeper model of the web browser and points out that browsers differ in how the various components communicate with one another. The browser model detailed in this work builds upon BLUEPRINT's model and more closely upon SCRIPTGARD's formalization [51]. We provide additional details in our model to demystify the browser's parsing behavior and explain subtleties in sanitization that the prior work did not address.

Purely browser-based solutions, such as XSSAuditor, are implemented in modern browsers. These mechanisms are useful in nullifying common attack scenarios by ob-

serving HTTP requests and intercepting HTTP responses during the browser's parsing. However, they do not address the problem of separating untrusted from trusted data, as pointed out by Barth et al. [12].

BEEP, DSI and NonceSpaces investigated client-server collaborative defenses. In these proposals, the server is responsible for identifying untrusted data, which it reports to the browser, and the browser ensures that XSS attacks can not result from parsing the untrusted data. While these proposals are encouraging, they require browser and server modifications. The closest practical implementation of such client-server defense architecture is the recent *content security policy* specification [56].

Correctness of Sanitization. While several systems have analyzed server-side code, the SANER [7] system empirically showed that custom sanitization routines in web applications can be error-prone. FLAX [50] and KUDZU [49] empirically showed that sanitization errors are not uncommon in client-side JavaScript code. While these works highlight examples, the complexity of the sanitization process remained unexplained. Our observation is that sanitization is pervasively used in emerging web frameworks as well as large, security-conscious applications. We discuss whether applications should use sanitization for defense in light of previous bugs.

Techniques for Separating Untrusted Content. Taint-tracking based techniques [16, 36, 44, 53, 63, 68] as well as security-typed languages [17, 47, 54, 57] aim to address the problem of identifying and separating untrusted data from HTML output to ensure that untrusted data gets sanitized before it is output. Web templating frameworks, some of which are studied in this work, offer a different model in which they coerce developers into explicitly specifying trusted content. This offers a fail-closed design and has seen adoption in practice because of its ease of use.

5 Conclusions and Future Work

We study the sanitization abstractions provided in 14 web application development frameworks. We find that frameworks often fail to comprehensively address the subtleties of XSS sanitization. We also analyze 8 web applications, comparing the sanitization requirements of the applications against the abstractions provided by the frameworks. Through real-world examples, we quantify the gap between what frameworks provide and what applications require.

Auto-sanitization Support and Context Sensitivity. Automatic sanitization is a step in the right direction. For correctness, auto-sanitization needs to be context-sensitive: context-insensitive sanitization can provide a false sense of security. Our application study finds that applications do, in fact, need to emit untrusted data in multiple contexts. However, the total number of contexts used by applications in our study is limited, suggesting that frameworks only need to support a useful subset of contexts.

Security of Client-side Code Evaluation. DOM-based XSS is a serious challenge in web applications, but no framework supports sanitization for dynamic evaluation on the client. Application developers must be particularly alert when using the DOM API. Of particular relevance to XSS sanitization is the possibility of the browser "undoing" server-side sanitization, making the application vulnerable to DOM-based XSS.

Context Expressiveness and Sanitizer Correctness. Some frameworks offer sanitization primitives as library functions the developer can invoke. We find that most frame-

works do not provide sufficiently expressive sanitizers, i.e., the sanitizers provided do not support all the contexts that applications use. For instance, applications emit untrusted data into URI attribute and JavaScript literal contexts, but most of the frameworks we study do not provide sanitizers for these contexts. As a result, application developers must implement these security-critical sanitizers themselves, a tedious and error-prone exercise. We also find that sub-context variance, such as role-based sanitizer selection, is common. Only one of the frameworks we examined provides any support for this pattern, and its support is limited.

Finally, our study identifies the set of assumptions fundamental to frameworks. Namely, frameworks assume that their sanitizers can be verified for correctness, and that HTML can be canonicalized to a single, standard form. Developers need to be aware of these assumptions before adopting a framework.

Future Directions. As we outline in this work, the browser's parsing and transformation of the web content is complex. If we develop a formal abstract model of the web browser's behavior for HTML 5, sanitizers can be automatically checked for correctness. Our browser model is a first step in this direction. We identify that parts of the web browser are either transducers or language recognizers. There have been practical guides for dealing with these issues, but a formal model of the semantics of browsers could illuminate all of the intricacies of the browser [70]. Verification techniques and tools for checking correctness properties of web code is an active area of research.

If one can show the correctness of a framework's sanitizers, we can prove the security and correctness for code generated from it. Though existing auto-sanitization mechanisms are weak today, they can be improved. Google AutoEscape is one attempt at this type of complete sanitization but is currently limited to a fairly restrictive templating language [21]. If these abstractions can be extended to richer web languages, it would provide a basis to build web applications secure from XSS from the ground up—an important future direction for research.

Acknowledgments

This material is based on work partially supported by the National Science Foundation (NSF) under Grants No. 0311808, No. 0832943, No. 0448452, No. 0842694, and No. CCF-0424422, and a NSF Graduate Research Fellowship, as well as by the Air Force Office of Scientific Research under Grant No. A9550-09-1-0539. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. jQuery, `http://jquery.com/`
2. Aas, G.: CPAN: URI::Escape, `http://search.cpan.org/~gaas/URI-1.56/URI/Escape.pm`
3. Adsafe : Making javascript safe for advertising, `http://www.adsafe.org/`
4. How To: Prevent Cross-Site Scripting in ASP.NET, `http://msdn.microsoft.com/en-us/library/ff649310.aspx`
5. Microsoft ASP.NET: Request Validation – Preventing Script Attacks, `http://www.asp.net/LEARN/whitepapers/request-validation`

6. Athanasopoulos, E., Pappas, V., Krithinakis, A., Ligouras, S., Markatos, E., Karagiannis, T.: xJS: practical XSS prevention for web application development. In: Proceedings of the 2010 USENIX conference on Web application development (2010)
7. Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In: Proceedings of the IEEE Symposium on Security and Privacy. Oakland, CA (2008)
8. Bandhakavi, S., King, S.T., Madhusudan, P., Winslett, M.: Vex: Vetting browser extensions for security vulnerabilities (2010)
9. Baron, D.: Mozilla's quirks mode, `https://developer.mozilla.org/en/mozilla's_quirks_mode`
10. Barth, A., Caballero, J., Song, D.: Secure content sniffing for web browsers *or* how to stop papers from reviewing themselves. In: Proceedings of the 30th IEEE Symposium on Security and Privacy. Oakland, CA (May 2009)
11. Barth, A., Felt, A.P., Saxena, P., Boodman, A.: Protecting browsers from extension vulnerabilities (2009)
12. Bates, D., Barth, A., Jackson, C.: Regular expressions considered harmful in client-side xss filters. In: Proceedings of the 19th international conference on World wide web. pp. 91–100. WWW '10, ACM, New York, NY, USA (2010)
13. Bisht, P., Venkatakrishnan, V.: XSS-GUARD: precise dynamic prevention of cross-site scripting attacks. Detection of Intrusions and Malware, and Vulnerability Assessment pp. 23–43 (2008)
14. Google-caja: A source-to-source translator for securing javascript-based web content, `http://code.google.com/p/google-caja/`
15. CakePHP: Sanitize Class Info, `http://api.cakephp.org/class/sanitize`
16. Chin, E., Wagner, D.: Efficient character-level taint tracking for java. In: Proceedings of the 2009 ACM workshop on Secure web services. pp. 3–12. SWS '09, ACM, New York, NY
17. Chong, S., Liu, J., Myers, A.C., Qi, X., Vikram, K., Zheng, L., Zheng, X.: Secure web applications via automatic partitioning. In: Proceedings of twenty-first ACM SIGOPS Symposium on Operating systems principles. pp. 31–44. ACM, New York, NY, USA (2007)
18. ClearSilver: Template Filters, `http://www.clearsilver.net/docs/man_filters.hdf`
19. CodeIgniter/system/libraries/Security.php, `http://bitbucket.org/ellislab/codeigniter/src/tip/system/libraries/Security.php`
20. CodeIgniter User Guide Version 1.7.2: Input Class, `http://codeigniter.com/user_guide/libraries/input.html`
21. Ctemplate: Guide to Using Auto Escape, `http://google-ctemplate.googlecode.com/svn/trunk/doc/auto_escape.html`
22. django: Built-in template tags and filters, `http://docs.djangoproject.com/en/dev/ref/templates/builtins`
23. Django sites : Websites powered by django, `http://www.djangosites.org/`
24. The Django Book: Security, `http://www.djangobook.com/en/2.0/chapter20/`
25. Finifter, M., Wagner, D.: Exploring the Relationship Between Web Application Development Tools and Security. In: Proceedings of the 2nd USENIX Conference on Web Application Development. USENIX (June 2011)
26. Finifter, M., Weinberger, J., Barth, A.: Preventing capability leaks in secure javascript subsets. In: Proc. of Network and Distributed System Security Symposium, 2010
27. Guha, A., Krishnamurthi, S., Jim, T.: Using static analysis for ajax intrusion detection. In: Proceedings of the 18th international conference on World wide web. pp. 561–570. WWW '09, ACM, New York, NY, USA (2009)
28. Google Web Toolkit: Developer's Guide – SafeHtml, `http://code.google.com/webtoolkit/doc/latest/DevGuideSecuritySafeHtml.html`
29. Hansen, R.: XSS cheat sheet (2008)
30. Hickson, I.: HTML 5 : A vocabulary and associated apis for html and xhtml, `http://www.w3.org/TR/html5/`

31. HTML Purifier Team: Css quoting full disclosure (2010), http://htmlpurifier.org/security/2010/css-quoting
32. HTML Purifier : Standards-Compliant HTML Filtering, http://htmlpurifier.org/
33. Huang, Y.W., Yu, F., Hang, C., Tsai, C.H., Lee, D.T., Kuo, S.Y.: Securing web application code by static analysis and runtime protection. In: Proceedings of the 13th international conference on World Wide Web. pp. 40–52. WWW '04, ACM, New York, NY, USA (2004)
34. Jean, J.: Facebook CSRF and XSS vulnerabilities: Destructive worms on a social network, http://seclists.org/fulldisclosure/2010/Oct/35
35. JiftyManual, http://jifty.org/view/JiftyManual
36. Jovanovic, N., Krügel, C., Kirda, E.: Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In: IEEE Symposium on Security and Privacy (2006)
37. Kirda, E., Kruegel, C., Vigna, G., Jovanovic, N.: Noxes: a client-side solution for mitigating cross-site scripting attacks. In: Proceedings of the 2006 ACM symposium on Applied computing. pp. 330–337. ACM (2006)
38. KSES Developer Team: Kses php html/xhtml filter, http://sourceforge.net/projects/kses/
39. Livshits, B., Lam, M.S.: Finding security errors in Java programs with static analysis. In: Proceedings of the Usenix Security Symposium (2005)
40. Livshits, B., Martin, M., Lam, M.S.: SecuriFly: Runtime protection and recovery from Web application vulnerabilities. Tech. rep., Stanford University (Sep 2006)
41. Martin, M., Lam, M.S.: Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In: 17th USENIX Security Symposium (2008)
42. The Mason Book: Escaping Substitutions, http://www.masonbook.com/book/chapter-2.mhtml
43. Nadji, Y., Saxena, P., Song, D.: Document structure integrity: A robust basis for cross-site scripting defense. In: NDSS (2009)
44. Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., Evans, D.: Automatically hardening web applications using precise tainting. 20th IFIP International Information Security Conference (2005)
45. XSS Prevention Cheat Sheet, http://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet
46. Pullicino, J.: Google XSS Flaw in Website Optimizer Explained (December 2010), http://www.acunetix.com/blog/web-security-zone/articles/google-xss-website-optimizer-scripts/
47. Robertson, W., Vigna, G.: Static enforcement of web application integrity through strong typing. In: Proceedings of the 18th conference on USENIX security symposium. pp. 283–298. SSYM'09, USENIX Association, Berkeley, CA, USA (2009)
48. Ruby on Rails Security Guide, http://guides.rubyonrails.org/security.html
49. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for javascript. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy. pp. 513–528. SP '10, IEEE Computer Society, Washington, DC, USA (2010)
50. Saxena, P., Hanna, S., Poosankam, P., Song, D.: FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications. In: 17th Annual Network & Distributed System Security Symposium, (NDSS) (2010)
51. Saxena, P., Molnar, D., Livshits, B.: Scriptgard: Preventing script injection attacks in legacy web applications with automatic sanitization. Tech. rep., Microsoft Research (September 2010)
52. Schmidt, B.: Google Analytics XSS Vulnerability, http://spareclockcycles.org/2011/02/03/google-analytics-xss-vulnerability/
53. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: Proceedings of the 2010 IEEE Symposium on Security and Privacy. pp. 317–331. SP '10, IEEE Computer Society, Washington, DC, USA (2010)

54. Seo, J., Lam, M.S.: Invisitype: Object-oriented security policies (2010)
55. Smarty Template Engine: escape, http://www.smarty.net/manual/en/language.modifier.escape.php
56. Stamm, S.: Content security policy (2009), https://wiki.mozilla.org/Security/CSP/Spec
57. Swamy, N., Corcoran, B., Hicks, M.: Fable: A language for enforcing user-defined security policies. In: Proceedings of the IEEE Symposium on Security and Privacy (May 2008)
58. Template::Manual::Filters, http://template-toolkit.org/docs/manual/Filters.html
59. Ter Louw, Mike and V.N. Venkatakrishnan: BluePrint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In: Proceedings of the IEEE Symposium on Security and Privacy (2009)
60. TwitPwn: DOM based XSS in Twitterfall (2009), http://www.twitpwn.com/2009/07/motb-08-dom-based-xss-in-twitterfall.html
61. Twitter: All about the "onMouseOver" incident, http://blog.twitter.com/2010/09/all-about-onmouseover-incident.html
62. UTF-7 XSS Cheat Sheet, http://openmya.hacker.jp/hasegawa/security/utf7cs.html
63. Venema, W.: Taint support for PHP (2007), ftp://ftp.porcupine.org/pub/php/php-5.2.3-taint-20071103.README.html
64. Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Cross site scripting prevention with dynamic data tainting and static analysis. In: Proceeding of the Network and Distributed System Security Symposium (NDSS). vol. 42. Citeseer (2007)
65. Weinberger, J., Saxena, P., Akhawe, D., Finifter, M., Shin, R., Song, D.: An empirical analysis of xss sanitization in web application frameworks. Tech. Rep. UCB/EECS-2011-11, EECS Department, University of California, Berkeley (Feb 2011)
66. Xie, Y., Aiken, A.: Static detection of security vulnerabilities in scripting languages. In: Proceedings of the Usenix Security Symposium (2006)
67. xssterminate, http://code.google.com/p/xssterminate/
68. Xu, W., Bhatkar, S., Sekar, R.: Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In: Proceedings of the 15th USENIX Security Symposium. pp. 121–136 (2006)
69. Yii Framework: Security, http://www.yiiframework.com/doc/guide/1.1/en/topics.security
70. Zalewski, M.: Browser security handbook. Google Code (2010), http://code.google.com/p/browsersec/wiki/Part1
71. Zend Framework: Zend_Filter, http://framework.zend.com/manual/en/zend.filter.set.html

A Transductions in the Browser

Table 5 details browser transductions that are automatically performed upon reading or writing to the DOM. The DOM property denotes the various aspects of an element accessible through the DOM APIs, while the access method describes the specific part of the API through which a developer may edit or examine these attributes. Excepting "specified in markup", the methods are all fields or functions of DOM elements.

Table 6 describes the specifics of the transducers employed by the browser. Except for "HTML entity decoding", the transductions all occur in the parsing and serialization processes triggered by reading and writing these properties as strings. When writing to a property, the browser parses the string to create an internal AST representation. When reading from a property, the browser recovers a string representation from the AST.

DOM property	Access method	Transductions on reading	Transductions on writing
`data-*` attribute	`get/setAttribute`	None	None
	`.dataset`	None	None
	specified in markup	N/A	HTML entity decoding
src, href attributes	`get/setAttribute`	None	None
	`.src`, `.href`	URI normalization	None
	specified in markup	N/A	HTML entity decoding
id, alt, title, type, lang, class dir attributes	`get/setAttribute`	None	None
	.[attribute name]	None	None
	specified in markup	N/A	HTML entity decoding
style attribute	`get/setAttribute`	None	None
	`.style.*`	CSS serialization	CSS parsing
	specified in markup	N/A	HTML entity decoding
HTML contained by node	`.innerHTML`	HTML serialization	HTML parsing
Text contained by node	`.innerText`, `.textContent`	None	None
HTML contained by node, including the node itself	`.outerHTML`	HTML serialization	HTML parsing
Text contained by node, surrounded by markup for node	`.outerText`	None	None

Table 5. Transductions applied by the browser for various accesses to the document. These summarize transductions when traversing edges connected to the "Document" block in Figure 1.

Type	Description	Illustration
HTML entity decoding	Replacement of character entity references with the actual characters they represent.	`&` → `&`
HTML parsing	Tokenization and DOM construction following the HTML parsing rules, including entity decoding as appropriate.	`<p>&gt;</p>` → *HTML element* P *with body* `>`
HTML serialization	Creating a string representation of an HTML node and its children.	*HTML element* P *with body* `>` → `<p>&gt;</p>`
URI normalization	Resolving the URI to an absolute one, given the context in which it appears.	`/article title` → `http://www.example.com/article%20title`
CSS parsing	Parsing CSS declarations, including character escape decoding as appropriate.	`color: \72\65\64` → `color: red`
CSS serialization	Creating a canonical string representation of a CSS style declaration.	"`color:#f00`" → "`color: rgb(255, 0, 0);` "

Table 6. Details regarding the transducers mentioned in Table 5. They all involve various parsers and serializers present in the browser for HTML and its related sub-grammars.

Textual values are HTML entity decoded when written from the HTML parser to the DOM via edge 1 in Figure 1. Thus, when a program reads a value via JavaScript, the value is entity decoded. In some cases, the program must re-apply the sanitization to this decoded value or risk having the server's sanitization negated.

One set of DOM read access APIs creates a serialized string of the AST representation of an element, as described in Table 6. The other API methods simply read the text values of the string versions (without serializing the ASTs to a string) and perform no canonicalization of the values.

The transductions vary significantly for the DOM write access API as well, as detailed in Table 5. Some writes cause input strings to be parsed into an internal AST representation, or apply simple replacements on certain character sequences (such as URI percent-decoding), while others store the input as is.

In addition, the parsers in Figure 1 apply their own transductions internally on certain pieces of their input. The CSS and JavaScript parsers unescape certain character sequences within string literals (such as Unicode escapes), and the URI parser applies some of its own as well (undoing percent-encoding).