An Empirical Analysis of XSS Sanitization in Web Application Frameworks

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

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-11
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-11.html

February 9, 2011

An Empirical Analysis of XSS Sanitization in Web Application Frameworks

Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin, Dawn Song
University of California, Berkeley

Abstract

Filtering or sanitization is the predominant mechanism in today's applications to defend against cross-site scripting (XSS) attacks. XSS sanitization can be difficult to get right as it ties in closely with the parsing behavior of the browser. This paper explains some of the subtleties of ensuring correct sanitization, as well as common pitfalls. We study several emerging web application frameworks including those presently used for development of commercial web applications. We evaluate how effective these frameworks are in guarding against the common pitfalls of sanitization. We find that while some web frameworks safeguard against the empirically relevant use cases, most do not. In addition, some of the security features in present web frameworks provide a false sense of security.

1. Introduction

Cross-site scripting (or XSS) attacks continue to plague existing and emerging web applications, despite receiving intense focus from both industry and academia. Researchers have proposed many novel defenses, ranging from purely server-side to browser-based or both. *Sanitization* or *filtering*, the practice of encoding or eliminating dangerous constructs in untrusted data, remains the predominant defense strategy in existing applications.

A variety of web application frameworks, including those used for major commercial applications, have emerged. As these frameworks gain wide acceptance as platforms for securing applications, it becomes important to quantify the extent to which web frameworks address the fundamental causes of XSS vulnerabilities in existing web applications.

An XSS attack occurs when the web browser executes, as higher-privileged application code, data input controlled by a low-privileged entity. Sanitization for XSS defense involves the following two main components[1]:

(a) *Identifying Untrusted Data* (or "what to sanitize"): Even for the simplest applications, only untrusted input data should be sanitized before being emitted in the HTTP response—code written by the developer is implicitly trusted and should not be sanitized.

(b) *Correctness of Sanitization*: After identifying untrusted data, the sanitization performed needs to be complete and correct to prevent code injection attacks due to insufficient or incorrect sanitization. Additionally, the application must not break functionality by overzealous application of sanitization.

The goal of this paper is to systematically understand the challenges in implementing these two components, evaluate support for them in existing web frameworks, and develop a basic understanding of sanitizer usage in large, real-world applications today.

Addressing Identification of Untrusted Data. A goal of this paper is to understand how web application frameworks address the challenges of identifying untrusted data by using HTML templating, as well as the flexibility of built-in support for embedding untrusted data in web application output. Prior research has focused on addressing this problem by employing techniques such as taint-tracking [36, 43, 60, 65] or type systems [46]. We aim to quantify how web applications address this challenge differently and how the two may interplay with each other in the future.

Addressing Correctness. Ensuring correctness for cross-site scripting sanitizers can be tricky, primarily because it involves a deep understanding of the behavior of web browsers. We first formalize XSS sanitization by developing a detailed model of the web browser, demystifying some of the cryptic intricacies of browser behavior. We explain some of the subtleties in "getting sanitization right," including how sanitization applied by the server can be "undone" during the lifetime of untrusted data. Based on this understanding of browser behavior, we illuminate error-prone cases for sanitization, and the poorly-understood subtle cases worthy of further attention. Previous research has addressed correctness of the sanitization process [5, 48, 49, 59], but we focus on why the sanitization process goes wrong rather than how to verify that it is correct.

Web application frameworks have gained popularity, as they promise ease of development and a more principled approach to security. To study their role, we gather 13 popular web frameworks, including several in commercial use. We first quantify how these frameworks provide sanitization defense for XSS. Particularly interesting is the support for *auto-sanitization*, a feature in which the web framework takes the responsibility for eliminating XSS attacks for the developers. We find that while most frameworks do not provide this feature, in the ones that do, it often provides little more than a false sense of security; unlike SQL injection (a related web vulnerability), cross-site scripting attacks require *context-sensitive* sanitization, yet most of these frameworks provide context-insensitive sanitization. We also quantify some of the other interesting variations in underlying principles employed in these frameworks pertaining to: (a) where to sanitize, (b) support for separation of HTML templates from data, and (c)

[1] Vulnerabilities in web browsers and its components as well as under specification of web standards have been another reason for XSS attacks [7, 8, 30]. However, we do not consider these in this paper.

degree of expressiveness in sanitizers.

In order to understand whether real-world applications today could take advantage of existing web frameworks in the near future, we perform an empirical evaluation of 8 popular applications ranging from 10 KLOC to over 500 KLOC. We find that most advanced web frameworks do offer support for all HTML contexts actually used by our subject applications.

Finally, we outline sanitization practices in real-world applications that web frameworks do not currently support, pointing out the need for extensible sanitization policies in web frameworks. We formulate the abstraction of a *sanitization policy function* that captures how applications pick sanitizers to apply. We show that sanitization policy functions are only implicitly enforced (rather than being explicitly stated) in application logic, are complex, and vary significantly from one application to another.

In the following sections, we:

- Formalize the concepts of embedding contexts, sanitizers, and browser transductions, and use these formalizations to guide our inquiry into the subtleties and pitfalls of XSS sanitization.
- Analyze the correctness and completeness of sanitization mechanisms of web frameworks.
- Discuss our empirical analysis of web applications, which allows us to compare sanitization mechanisms supported in frameworks and those required by real web applications.

2. Contexts, Sanitizers and Correctness

The correctness of sanitization mechanisms is closely tied to the parsing behavior of the web browser. We present an abstract model of the web browser which explains how the intricacies of XSS sanitization are more complex than classic command injection sanitization. Many formulations of cross-site scripting defense already exist in the literature [11, 42, 50, 59, 62]. Our formalization of the browser model and XSS builds upon previous work [50], with an enhanced focus on browser transductions.

2.1. A Model of The Web Browser

We can view the browser as a collection of sub-grammars, each corresponding to the different languages recognized in the web browser: HTML, JavaScript, CSS, the grammar for URI schemes, and so on. The browser parses documents it retrieves, such as HTML pages, in stages. The browser parses each portion of the input by one or more sub-grammars. For example, the browser parses HTML input according to HTML syntax rules (the latest specification effort for which is HTML 5). Certain portions of the input may contain fragments from other languages, such as within `<script>` blocks; these fragments are then shipped to the corresponding sub-parser for that language.

Figure 1 depicts the complexity of the interaction between the various sub-parsers in the browser. Note that the shaded blocks do not represent sub-parsers themselves, but other components of the browser that interact with the sub-parsers. The DOM stores all the parse-tree outputs of the sub-parsers, funnels input fragments between them, and provides methods for manipulating both the parsed outputs as well as the string fragments corresponding to them. The JavaScript runtime uses the DOM to effect changes to the page loaded in the browser, causing sub-parser invocation in the process.

Web applications embed data from untrusted input sources (such as lower privileged users) in the HTTP response, which is then served to a higher privileged entity. In a cross-site scripting attack, parsing of untrusted data as a dangerous non-terminal leads to injected code execution. Informally, we say that untrusted data "breaks out" of a trusted *embedding context*, causing script execution. The set of attack strings that accomplish this varies significantly based on the trusted context in which untrusted data is embedded. For clarity, we begin by formalizing this notion of embedding context.

Embedding Contexts.

Definition 1: (**Embedding Context**) Let $G = (V, \Sigma, S, P)$ be a context-free grammar for a sub-language defined by a quadruple consisting of a set of non-terminal symbols V, terminal symbols Σ, start symbol S and a set of productions P. Let $D : S \Rightarrow^* \gamma$ denote the leftmost derivation sequence of the sentence γ under the production rules of G.

Let each step in the derivation sequence apply a production rule and yield a "sentential form," i.e., a sequence consisting of non-terminals and terminals.

A given untrusted substring $v \in \Sigma^*$ in the browser input $I \in \Sigma^*$ is said to be embedded in a context (or non-terminal) C, if

(a) the derivation $\mathcal{X} : \alpha\mathcal{C}\beta \Rightarrow^*_G \alpha u v w \beta$ is a subsequence of the leftmost derivation of I, for terminal words α, u and w consisting only of trusted characters, and

(b) $\mathcal{X}$ is minimal, i.e., $\nexists C'$, $\mathcal{X}' : \alpha'\mathcal{C}'\beta' \Rightarrow^*_G \alpha' u' v w' \beta'$ where $\mathcal{X}'$ is a subsequence of $\mathcal{X}$, for terminal words α, u' and w' consisting only of trusted characters.

To make our definition concrete, we list some examples of embedding contexts in Table 1. Although we can define several such contexts with the above formalism, we wish to restrict ourselves to those that real-world applications actually use and those that web application frameworks support.

2.2. Context Transitions

Associated with each sub-grammar in the browser are *transition non-terminals*, i.e., non-terminals that recognize fragments of the input that belong to another sub-grammar. For instance, the parser recognizes data embedded in event handler attributes of HTML tags as JavaScript statements, and the browser must parse the attributes further and interpret them in the JavaScript engine. Whenever the parser derives a fragment of the input from a transition non-terminal, the parser ships the fragment to the corresponding sub-parser for further processing (corresponding to the edges in Figure 1). We refer

Example	Sub-grammar(s)	Embedding contexts
`<p>untrusted</p>`	HTML	HTML tag content
`<p id="untrusted">`	HTML	Double-quoted HTML attribute
`<a onclick="document.title='untrusted';">`	JavaScript, HTML	Single-quoted JavaScript string in double-quoted HTML attribute
`<img src=http://untrusted>`	URI, HTML	Domain of a URI in an unquoted HTML attribute
`<div style="color: rgb(untrusted);">`	CSS, HTML	RGB color specification in double-quoted HTML attribute

TABLE 1. Examples of common embedding contexts.

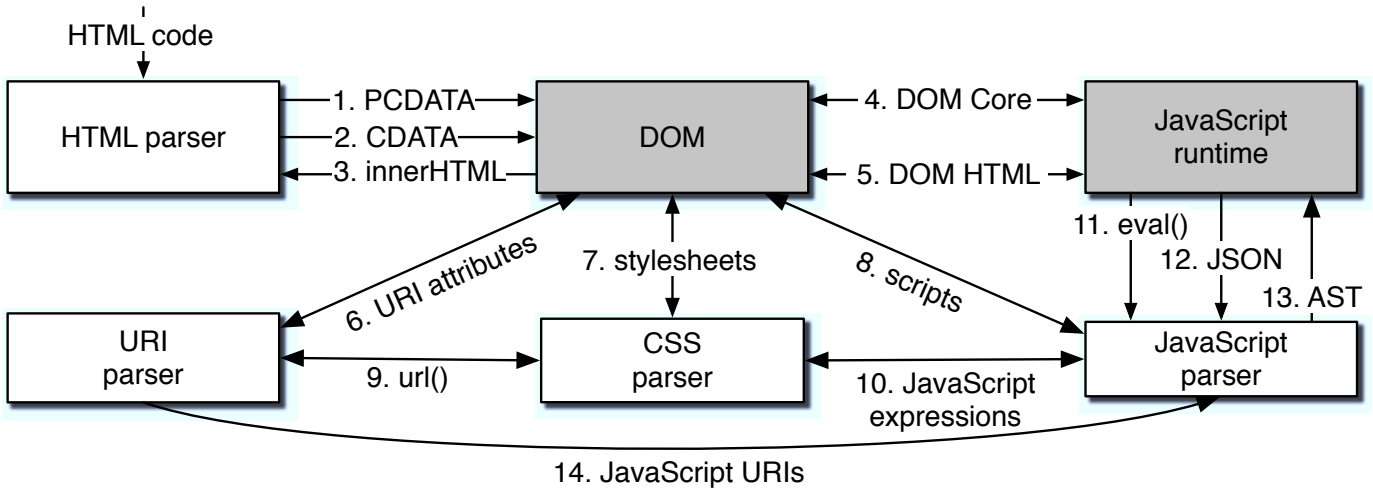

Fig. 1. A model of a web browser showing the possible flows of data between the parsers, the DOM, and the JavaScript runtime. The DOM stores all syntax-tree outputs of the parsers, and the JavaScript runtime manipulates the DOM. For most of the edges, the browser applies a transduction when they are traversed. See Table 2 for more details about the edges.

to this transition between a context in one sub-grammar to a context in another sub-grammar as a *context transition*.

Static Transitions. Some context transitions can be statically determined by the placement of the untrusted data in the web application's output, whereas others result from execution of client-side application code. Take the following example:

```
<img src="untrusted">
```

This shows a fragment of HTML code with a context transition from an HTML attribute context to the URI parser's scheme (e.g., `http:` or `mailto:`) context. Identifying static transitions is feasible using a high fidelity model of a web browser for parsing.

Dynamic Context Transitions. Dynamic context transitions occur when data is programmatically read from or written to the DOM by JavaScript. There are several ways for data parsed by the HTML engine to be read and placed into the DOM via JavaScript. Table 3 shows a common subset of these methods. Identifying dynamic transitions requires an involved analysis of the application's JavaScript code.

Browser Transductions. A significant complication is that the browser applies a transduction on input along each transition edge. For instance, when the data in a URL attribute is found, the browser first decodes HTML entities [28] in it before shipping it to the URI parser. For example, we formally denote each transition from context C_1 to C_2 accompanied by a transduction T as $C_1 \rightarrow^T C_2$.

Transductions are important because they make reasoning about sanitization non-trivial. As described later, sometimes these transductions can "undo" the sanitization applied on untrusted data, leading to subtle bugs in the sanitization process.

Transductions are edge-specific; we detail them in Table 2. In particular, Table 3 details the transductions that are automatically performed by the browser upon reading or writing DOM elements via JavaScript.

There are subtle features of these accesses that are important for sanitization correctness. Specifically, note that:

(a) Data values are HTML entity decoded when written from the HTML parser to the DOM via edge 1 in Figure 1. Therefore, when a program reads a value via JavaScript, e.g., in `getElementById('a').getAttribute('id')`, the value is entity decoded and if it is subsequently evaluated as HTML code, it results in code injection. This automatic entity decode can have the effect of negating sanitization applied by the server on untrusted data.

(b) DOM access methods expose two different ways of accessing DOM content. One set of read access APIs creates a serialized string of the AST representation of an element. These correspond to entries named "HTML serialization," "CSS serialization," and "URI normalization" (which addition-

Edge	Trigger for traversal	Transduction function	Example (input → output)
1	Normal HTML markup.	HTML parsing	`<p>1&lt;2</p>` → DOM node "p" containing `1<2`
2	Character data in HTML, which is not parsed (e.g. `<script>` blocks).	None	
3	Setting the `innerHTML` property on a DOM node, which invokes the HTML parser to obtain a corresponding DOM tree from the input string.	HTML parsing	`<p>1&lt;2</p>` → DOM node "p" containing `1<2`
4	Uses of Core DOM APIs, such as `getAttribute` or `setAttribute`.	None	
5	Uses of HTML-specific DOM APIs, such as `.innerHTML` or `.style`.	*See Table 3*	
6	Delivery from HTML parser of attribute contents that are URIs (e.g., `href`, `src`).	None	
7	Instantiation of CSS stylesheets, from `<style>` blocks or `style` attributes.	CSS escape sequence decoding	`\61` → `a`
8	JavaScript source code, in `<script>` blocks or in event handler attributes.	JavaScript parsing	`\u61` → `a`
9	URI specifiers in CSS stylesheets.	None	
10	JavaScript in CSS stylesheets (e.g., `expression`).	JavaScript parsing	`\u61` → `a`
11	Call of JavaScript's `eval` function.	JavaScript parsing	`\u61` → `a`
12	Use of built-in JSON parser.	JSON parsing	`\u61` → `a`
13	Result of parsing a JavaScript program.	N/A	

TABLE 2. Data flows between the components of our browser model as depicted in Figure 1. *Edge* corresponds to the numbered arrows in Figure 1.

DOM property	Access method	Transductions on reading	Transductions on writing
`data-*` attribute	`get/setAttribute`	None	None
	`.dataset`	None	None
	in markup	N/A	HTML entity decoding
src, href attributes	`get/setAttribute`	None	None
	`.src`, `.href`	URI normalization	None
	in markup	N/A	HTML entity decoding
id, alt, title, type, lang, class, dir attributes	`get/setAttribute`	None	None
	.[attribute name]	None	None
	in markup	N/A	HTML entity decoding
style attribute	`get/setAttribute`	None	None
	`.style.*`	CSS serialization	CSS parsing
	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 3. Transductions applied by the browser for various accesses to the DOM. These summarize transductions when traversing edges 1, 4, or 5 in Figure 1.

ally makes the URI absolute and applies URI encoding). This API corresponds to DOM HTML APIs (edge 5 in Figure 1). For example, when reading an `href` attribute that has a relative URI using the `.href` attribute of the DOM node, the browser will produce a canonicalized absolute URI. 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.

(c) The transductions vary significantly for the DOM write access API as well, as detailed in Table 3.

Section 2.5 explains in further detail how these subtleties can result in bugs.

2.3. Sanitizers

Intuitively, a sanitizer is a function that aims to transform content to ensure that the browser's parser confines the content to explicitly allowed syntactic elements. Sanitizers can perform the transformation on the server or in JavaScript code running on the client.

Definition 2: (**Sanitizer**) A sanitizer is a function

$$f : string \rightarrow string$$

Defining sanitizers as general string transformers allows us to include buggy/incorrect sanitizers in our definition. A *correct* sanitizer is a sanitizer that provides some confinement/safety property on the output.

2.4. Correctness of Sanitization

Checking correctness of XSS sanitization is more complex than command injection sanitization, formalized by Wasserman et. al. [56]. We recall below their basic definition of safety and then bulid on it.

Definition 3: (**Syntactic Confinement**) A derivation D is syntactically confined to a set of non-terminals A, denoted by

$D_{\downarrow} \subset \mathcal{L}(\mathcal{A})$, if all the sentential forms generated in D contain only non-terminals from the set A.

A function f ensures syntactic confinement to a given set of allowed non-terminals in a sub-grammar $\mathcal{G}$ if $\forall \alpha,\ D : S \Rightarrow^{*}_{\mathcal{G}} f(\alpha),\ D_{\downarrow} \subset \mathcal{L}(\mathcal{A})$

We highlight that the actual safety property that sanitization aims to achieve is the *context chain safety* defined below, which accounts for context transitions and the complications introduced by browser transductions.

Definition 4: (**Context Chain Safety**) Let $C_1 \rightarrow^{T_1} C_2 \rightarrow^{T_2} \ldots C_{n-1} \rightarrow^{T_{n-1}} C_n$ be a chain of context transitions induced by a fragment of untrusted data u in the browser. A sanitizer f is correct only if $\forall C_i, T_{i-1}(\ldots(T_1(f(u))))$ ensures syntactic confinement to only the allowed non-terminals in C_i.

As highlighted earlier in this section, transductions can vary significantly based on the context chain and based on the use of access methods in JavaScript. The next section presents several examples of common modes of failure in sanitization usage.

2.5. Evaluating Effectiveness Against Common Failures of Sanitization

We describe a number of failure reasons for XSS sanitization, and formulate the questions pertaining to correctness that our evaluation in this paper aims to answer.

Incomplete sanitization for nested contexts. Consider the earlier example that shows a nested context for the underlined untrusted data:

```
<img src="untrusted">
```

If the developer HTML entity escapes only the untrusted data, the attacker can not break out of the HTML attribute context. However, because the underlined portion is a URI context, the attacker is still able to attack the URI parser. An attack string such as `javascript:` will, when parsed by the URI parser, cause it to recognize the protocol as a JavaScript URI, initiating another context transition to the JavaScript parser.

Another possible problem is the failure to account for transductions. For instance, in the above example, even if the server performs HTML entity encoding of dangerous characters (such as `:`) in the untrusted input, the browser may "undo" the encoding to execute the attack. For instance, the attack string `javascript&58;` will execute script code in this example, because browsers HTML entity decode the `&58;` back to a `:` character before giving it to the URI parser.

An important objective of our evaluation is to identify whether built-in sanitizers in state-of-the-art web frameworks handle static transitions well for the contexts in which they support embedding data.

Incomplete sanitization for dynamic context transitions. JavaScript code execution can read untrusted data from the DOM and force evaluation or re-parsing of the data in a different context. Consider the example of a bug shown in Figure 2, which we found in PHPBB3, a popular forum application. The

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

Fig. 2. A bug in PHPBB3 that results due to failure to perform sanitization for a dynamic context transition

application's JavaScript code reads data sanitized by the server (using HTML-entity encoding) from a parsed attribute. (using HTML-entity encoding). It dynamically places the data in an HTML content context and evaluates the string as HTML.

The browser automatically decodes HTML entities, undoing the server's sanitization, and the `getAttribute` DOM API reads the decoded string, which allows injection of a `script` block on HTML evaluation. In particular, the safe string `<script>` will be decoded to `<script>` by the `getAttribute` call and writing `<script>` via `innerHTML` results in XSS. JavaScript-heavy web applications are likely to have many of these vulnerabilities. Previously developed vulnerability analysis techniques aim to solve this problem; however, their applicability to securing web applications during construction has not been demonstrated [48, 49].

In our evaluation, we aim to evaluate whether existing web frameworks address the issue of identifying dynamic context transfers, and if so, to what extent they enable correct defenses against them.

Sanitizing difficult contexts. Certain contexts are inherently difficult to sanitize. For instance, sanitization inside an unquoted HTML attribute requires elimination of a variety of untrusted characters (including `[space] % * + , - / ; < = > ^ and |`), which varies across browsers. However, if the attributes are quoted, defensive rules to ensure syntactic confinement are simple and do not vary much across browsers [44]. Similarly, embedding data directly inside a `<script>` block outside of string and number literal contexts is problematic because one must have knowledge of the JavaScript statement parser.

An important question we empirically study in Section 4 is characterizing the set of contexts security-conscious applications use in practice and those contexts that web applications aim to safeguard.

Whitelisting vs. blacklisting. Several sanitizers aim to blacklist sets of known attack strings. For instance, in SANER, the authors present examples of a blacklisting approach where the failure to account for browser variation or failure to recognize dangerous non-terminals results in attacks (examples 2-5)[5].

Another approach to sanitization is *whitelisting-based canonicalization*, which we observe is pervasively employed in state-of-the-art web frameworks as well as HTML purification engines in use by web applications. The idea is to: (a) parse untrusted data on the server within the context where it will be embedded, generating a parse tree, (b) eliminate all but a whitelist of HTML elements in the parse tree, and (c) re-encode the parse tree representation into a canonical HTML

form, the interpretation of which does not vary across web browsers.

We empirically study what fraction of web applications and web frameworks enable this form of sanitization. Previous research has not shown the insecurity of this mechanism, nor has research applied formal analysis to prove its security. This mechanism stands in contrast to previous sanitization-free defenses that assume browser variance is problematic for a sanitization defense [59].

3. Web Frameworks

We evaluate a set of popular web frameworks that covers most of the widely used web programming languages. Our focus is on picking popular frameworks, especially those in commercial use.

3.1. Expressiveness

We evaluated the set of contexts for which each framework supports sanitization. That is, for each context, we determine whether the framework provides developers with a built-in sanitizer sufficient to safely emit untrusted data into that context. For contexts not natively supported, developers must write their own sanitizer. The results of this analysis are presented in Table 4.

We observe that 6 out of 13 of the frameworks do not support contexts other than the HTML tag content context (e.g., as the content body of a tag), HTML attributes and URI attributes. The most commonly supported sanitizers for these are HTML entity encoding and URI encoding, respectively, with customization features available.

Four web frameworks, ClearSilver, Ctemplate, Django, and Smarty, provide appropriate sanitization functions for emitting untrusted data into a JavaScript string. Only 1 framework, namely Google Ctemplate, provides a sanitizer for emitting data into JavaScript, outside of the string literal context. However, the sanitizer is a very restrictive whitelist, allowing only numeric or boolean literals. Allowing untrusted JavaScript code to be emitted into JavaScript contexts is not supported by any framework that we studied. Typically, code embedded in such contexts is outside the scope of the safety property outlined in Section 2. Instead, properties such as authority safety are desired [38], which are the target of other mechanisms such as Yahoo's ADsafe [2] or Google's Caja [12].

Four 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 URLs that use the `javascript:` scheme and accept only a whitelist of safe schemes, like `http:`. The defense for static context transition attacks outlined in Section 2.5 is correct in these frameworks.

3.2. Support for Identification of Untrusted Data

One of the reasons for XSS vulnerabilities in previous systems has been failure to identify untrusted data that needs sanitization from the rest of the web application's output. We first outline how the web application frameworks address this challenge, and then put it in perspective with other techniques proposed in research.

Of all the frameworks we studied, four frameworks distinctly support a separation between HTML code output and data — namely Django, GWT, Ctemplate and ClearSilver. While the others do encourage separation to some extent, they do not support *templates* that make the distinction explicit. A template is a document representing an HTML output of the application but with *data holes* placed throughout. A data hole is a location in a template filled by content that is not static HTML. A data hole can be as simple as a variable or contain complicated application logic. This separation between static, trusted content and potentially untrusted input allows these frameworks to conservatively auto-sanitize all the data holes.

The separation makes the sanitization a default fail-close design, i.e., even data that is trusted but not included in the template will be treated by the engine as untrusted. The fact that these systems are in pervasive commercial use suggests that the mechanism has had success thus far.

Much of the previous research on taint-tracking aims to address the issue of distinguishing what to sanitize versus what is HTML content. Tracking data dependencies only in taint-tracking can have false negatives [51]. Mechanisms deployed in web frameworks coerce programs to restrictive form to err on the side of conservativeness. These offer a different point of evaluation as compared to taint-based defenses. Security-typed languages such as JIF [15, 46, 57] also aim to take user annotation as a way to ensure this distinction. In comparison to templating, both taint-tracking and type-based solutions offer more expressive separation between content than a binary metric (data and template).

3.3. Support for Auto-Sanitization

Next, we evaluated the degree of support for automatic sanitization in these frameworks. Of the 13 frameworks evaluated, seven support some form of automatic sanitization. These appear in Table 5. Auto-sanitization is a feature that shifts the burden of ensuring safety against XSS from the developers to the framework. Essentially, in a framework that includes auto-sanitization, the application developer is responsible for indicating which variables will require sanitization. The web application framework is supposed to ensure that the correct sanitizer is applied to each variable that fills a hole.

3.3.1. Context-Insensitive Sanitization. Most of these frameworks provide automatic sanitization only for HTML tag content contexts. For example, Django's `autoescape` will not protect against untrusted input that is placed in the value of an `href` attribute:

```
<a href="{{ untrusted_link }}">Link supplied by user.</a>
```

`{{untrusted_link}}` is a template variable in a data hole, in this case referring to an untrusted user input. In this case, the data

Language	Framework	HTML	URL Attribute (excluding scheme)	URL Attribute (including scheme)	JS String	JS Number or Boolean	Style Attribute or Tag
Perl	Mason [1, 40]	✓	✓				
	Template Toolkit [58]	✓	✓				
	Jifty [32]	✓	✓				
PHP	CakePHP [13]	✓	✓				
	Smarty Template Engine [54]	✓	✓		✓		
	Yii [29, 66]	✓	✓				
	Zend Framework [67]	✓	✓				
	CodeIgniter [17, 18]	✓	✓				
VB, C#, C++, F#	ASP.NET [3]	✓	✓				
Ruby	Rails [47]	✓	✓				
Python	Django [20]	✓	✓	✓	✓		
Java	GWT SafeHtml [24]	✓	✓	✓			
C++	Ctemplate [19]	✓	✓	✓	✓	✓	✓
Language-neutral	ClearSilver [16]	✓	✓	✓	✓		✓

TABLE 4. Set of manually-invocable sanitizers provided by languages and/or frameworks for various contexts. For frameworks, we also include sanitizers provided by standard packages or modules for the language. We differentiate between a sanitizer for URL contexts that can correctly sanitize a complete URL (i.e., ensure it does not use the `javascript:` scheme and one that need only concern itself with URLs for which the scheme is already set. For the latter, simple URI encoding is sufficient.

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 [4]	✓		Request Reception
Ruby	xss_terminate Rails plugin [64]	✓		Database Insertion
Python	Django	✓		Template Processing
Java	GWT SafeHtml	✓	✓	Template Processing
C++	Ctemplate	✓	✓	Template Processing
Language-neutral	ClearSilver	✓	✓	Template Processing

TABLE 5. Extent of automatic sanitization support in various frameworks and pointcut where the automatic sanitization occurs.

hole is simply a variable that happens to refer to user input. The double bracket notation for data holes is common in many of the frameworks we examine. This untrusted link can contain a `javascript:` URL, which enables a cross-site scripting attack. Django's `autoescape` correctly handles only HTML contexts.

A plugin for Rails called xss-terminate automatically removes (or escapes, depending on its configuration) tags before content is persisted to the database. This suffers from the same problem as Django; it does nothing to prevent vulnerabilities in which untrusted content is included in other contexts. Similarly, CodeIgniter's automatic sanitization uses its `xss_clean` function in a context-insensitive manner.

Context-insensitive Sanitization is inadequate. Context-insensitive sanitization provides a false sense of security to developers. For example, the Django framework applies the HTML entity escape to all untrusted data in a context-insensitive way. Consider the following code from the GRAMPS application authored on Django:

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

The above case is interesting for two reasons. First, we notice that the developer chose not to rely on auto-sanitization for the `header.url` variable, manually escaping it instead. Unfortunately, `escape` is the wrong sanitizer; it merely HTML entity encodes the value resulting in an XSS vulnerability via the `javascript:` URI vector. The second interesting point is that had the developer relied on the auto-sanitization mechanism, the automatically applied sanitizer would have been wrong as well.

Consider another example taken from the Malaysia Crime application, authored in the Django framework:

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

The `crime.icon` variable is filled in at runtime as an argument to a JavaScript function call. Clearly, the auto-sanitization in this case performs an inconsistent sanitization, and if the developer relied on it, an XSS bug would result; additional dangerous characters (including `\n \r \ ;`) can be used to break out of the context into which `crime.icon` is emitted. In this case, the incorrect sanitization did not lead to a vulnerability because `crime.icon` was explicitly sanitized (against a whitelist of acceptable strings) at the time of input.

We conclude that context-insensitive sanitization is inadequate. Developers using context-insensitive sanitization must implement their own sanitizers to ensure security anyway. To verify this, we investigated how often sites rely on Django's auto-sanitization and for what fraction of the incorrectly auto-sanitized sinks the developer resorted to manual sanitization.

We examined a random sample of 10 Django applications (from a list of open-source Django applications [21]) to determine the fraction of variables emitted into an application's templates that are protected by Django's `autoescape`. Our results are presented in Table 6.

In all the applications in our sample, the majority of variable sinks were autoescaped. Though the majority of variables are emitted into HTML tag content contexts, and are therefore protected by `autoescape`, a significant fraction are not. Table 6 further distinguishes the fraction of sinks appearing in HTML tag content context versus others. In all contexts other than the HTML tag content context, inconsistent (and technically incorrect) sanitization is applied, which can lead to XSS vulnerabilities. Our subsequent manual investigation of all these cases revealed that the developer explicitly resorted to manual sanitization.

3.3.2. Context-Sensitive Sanitization. Three web frameworks, namely GWT, Clearsilver, and Ctemplate, support context-sensitive automatic sanitization. As in the other frameworks, when each template is emitted to the output, the platform automatically sanitizes the data embedded in template data holes. However, in these frameworks, the context for each hole is inferred as the template is parsed, and the sanitizer to apply is selected based on the context of the data hole in the template. Thus, a sanitizer is automatically applied to a variable in a data hole based on its context in the document.

For example, consider the following simple Ctemplate template:

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

This template contains three separate data holes in which values of variables will be emitted. When the template is processed, the context of the first hole, {{`NAME`}}, is recognized as a JavaScript string context, and just before the hole is filled with the value of `NAME`, the `:javascript_escape` modifier will be applied to the variable. This modifier indicates to the template parser that the JavaScript escape sanitizer should be applied to variables in this data hole before variable substitution. This auto-escaping is equivalent to explicitly writing {{`NAME:javascript_escape`}} instead of just {{`NAME`}}. For the second hole, {{`URL`}}, the `:url_escape_with_arg=html` modifier is applied indicating that the data hole is in a URL context and should be sanitized appropriately. Finally, the `:html_escape` modifier is automatically applied to the value of {{`PAGENAME`}} before the third hole is filled, indicating that the data hole is in an HTML context and variables should be HTML entity encoded.

Handling Nested Contexts or Static Context Transitions. Support for nested contexts varies significantly, as shown in Table 4. The Google Ctemplate framework offers sanitization for the maximum number of contexts, including some of those that involve implicit static transition contexts. The HTML attribute context (e.g., `href`, `style`, `onclick`) is a transition context as defined in section 2, i.e., certain attributes are recognized as URI attributes. JavaScript event handlers or style attributes are examples of these. Most of the other frameworks do not support any nesting of contexts.

Handling Dynamic Context Transitions. None of the web frameworks support sanitization for dynamic context transfers. That is, they do not identify potential context transfers due to the execution of JavaScript for auto-sanitization. Even the cases that can be dispatched with simple static analyses are not handled, such as in the following example

```
<script>
document.write("<div> {{ untrusted_data }} </div>");
</script>
```

3.4. Sanitization Strategy: Blacklists vs. Whitelists

One web framework we studied employs a blacklist-based sanitization approach. CodeIgniter employs a blacklist of filters in its `xss_clean` function, which applies several regular expression replacements in sequence. These regular expressions serve to blacklist potentially dangerous code that may appear in any context. For example, all instances of `document.cookie` are replaced with the text `[removed]`. Unfortunately, this blacklisting approach inspires little confidence in the completeness of the filter. Additionally, it may break functionality because it essentially applies each context's filter regardless of which context the data will ultimately appear in. This means, for example, that any occurrence of the term `document.cookie` will be removed, even if it is simply in the body of a user's forum post, and therefore intended to be placed in an HTML context.

Other than CodeIgniter, all frameworks used a whitelisting-based canonicalization approach described in Section 2. For HTML tag contexts, basic HTML entity encoding was supported in all frameworks. In addition, we found more comprehensive handling of HTML contexts in a number of other frameworks. For instance, Yii provides an interface to HTML Purifier [29] and its documentation recommends this interface as the preferred XSS defense mechanism. HTML Purifier fully parses an HTML snippet, and it claims to remove arbitrarily nested scripts in HTML tag content contexts. While it robustly

Web Application	Num. Sinks	% Auto-sanitized Sinks	% Sinks not sanitized (marked safe)	% Sinks manually sanitized	% Sinks in HTML Context	% Sinks in URL Attr. (excluding scheme)	% Sinks in URL Attr. (including scheme)	% Sinks in JS Attr. Context	% Sinks in JS Number/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 6. Actual usage of auto-sanitization in Django applications. The first two columns are the number of sinks in the templates and the percentage of these sinks for which auto-sanitization has not been disabled. Each of the remaining columns shows the percentage of sinks that appear in the given context.

handles untrusted data emitted into an HTML tag content context, it does not provide functionality for emitting untrusted data into any other contexts.

3.5. Placement of Sanitizers

Finally, we find that for frameworks that support auto-sanitization, there are variations in where to apply the sanitizers. Borrowing terminology from aspect-oriented programming, we refer to each set of program points at which auto-sanitization is applied as a *sanitization pointcut*. We have identified three different sanitization pointcuts in the frameworks we studied; Table 5 indicates which one is employed by each framework.

Request reception. Sanitizing at request reception has the advantage of precisely identifying all untrusted data. However, the disadvantage is that at this pointcut, there is no knowledge of where this untrusted data is going to end up. This makes it impossible to apply the least restrictive safe sanitization policy; instead you must apply an extremely restrictive sanitization policy under the assumption that the untrusted data could end up being emitted into any context.

Database insertion. Sanitizing at database insertion time ensures that no unsafe data is ever stored. Accordingly, this pointcut does not protect against reflected XSS vulnerabilities, but only stored XSS vulnerabilities (and SQL injection vulnerabilities, which we do not consider in the present work). We have anecdotal evidence that this pointcut can lead to subtle bugs. In a program that we analyzed, all data was sanitized before entering the database. This meant that when a user registered for the web site, his username was escaped before being stored. For a user whose name contains a character that is escaped, all login attempts fail because the username entered (which is not sanitized) does not match any stored username (all of which have been sanitized). As in the request reception pointcut, this problem results from a lack of knowledge regarding where the untrusted data will end up being used. In this case, its correct use in a comparison is precluded by being sanitized before database insertion.

Template processing. Sanitizing at template processing time has the advantage of being fail-safe because every output is sanitized by default. The disadvantage of this pointcut is very much the inverse of that of the request reception pointcut: at template processing time, the separation between untrusted and trusted data might have been lost. You cannot tell at this pointcut whether the data is trusted or untrusted. For this reason, some variables may accidentally be escaped multiple times, which could break functionality. The developer is responsible for manually marking the pieces of data that should be considered trusted, which is an inconvenient and error-prone process, though the errors do fall on the side of breaking functionality rather than security.

4. Web Applications

We evaluate a set of large, widely used web applications to compare the features provided by web frameworks with those required for authoring real-world applications. We gather empirical evidence from 8 diverse web applications, ranging from email clients to medical record management systems, about their use of embedded contexts and sanitizers. First, we evaluate whether applications use contexts that are inherently difficult to secure (as outlined in Section 2.5). Second, we compare the set of contexts used by applications with the set of contexts supported by web frameworks. Third, we evaluate whether these applications employ whitelisting or blacklisting

Application	Description	LOC	HTML Context	URL Attr. (excluding scheme)	URL Attr. (including scheme)	JS Attr. Context	JS Number/String Context	Number of Sanitizers	Number of 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 7. Details of the web applications we study and details on various contexts used by these applications.

based sanitizers. Finally, we evaluate whether the level of expressiveness supported by web frameworks is sufficient to enforce the sanitization policy that applications presently use.

4.1. Subject Applications & Analysis Infrastructure

Systematically studying large code bases requires an automatic analysis infrastructure. To limit our effort in building analysis infrastructure, we decided to focus on one language, namely PHP. We chose PHP because it is a popular web application language with over 20 million domains running PHP applications [45].

Our smallest subject application was over 19,000 lines of PHP code, while the largest was over half a million lines of code. Several of these applications, including PHPBB, WordPress and MediaWiki have been studied in previous research evaluations. Finally, we believe that these applications are widely used — for instance, MediaWiki is the framework behind Wikipedia.

Analysis of PHP applications. We describe the analysis infrastructure we build for systematically studying PHP applications. It provides two primary features: *automatic extraction* of sanitizers (as defined by Definition 2 in Section 2) and *inference of contexts* in which untrusted data is embedded by applications.

PHP is a highly dynamic, interpreted language. PHP has dynamic types, dynamically dispatched functions, invocations through dynamic arrays, and variable argument functions, which are challenging for static analysis. Notably, Facebook's HipHop PHP compiler and analyzer provides a level of type inference support [26], but the analysis is generally imprecise, unable to infer types across function calls. Though previous research has had success in automatic extraction of sanitizers in statically typed languages [37], these are not directly usable in our study of PHP because they operate on the assumption of having (*a priori*) inter-procedural dataflow graphs. As a result, our entire analysis infrastructure relies on dynamic analysis.

To drive the analysis infrastructure, we used a combination of automated blackbox fuzzing and manual interaction with the applications. Specifically, we built a PHP fuzzer that (a) systematically invokes all the PHP files in each application, (b) identifies all uses of GET, POST, and COOKIE input parameters, and (c) re-executes each file with random values for the newly identified inputs found in step (b). PHP functionality that was not properly explored using this mechanism was subject to addition manual interaction as well. We found a large number of distinct sinks containing untrusted data in our analysis, which we summarize in Table 7.

Extracting Sanitizers. Using dynamic analysis, we identify sets of sanitizers in web applications. We check the input and output of each function call that an application makes in a dynamic run. We use Definition 2 in Section 2.3 to identify sanitizer functions, and we identify sanitizers that provide HTML context chain safety, as per Definition 4.

For each dynamically executed path in the application, we extract all of the called functions with at least one string argument. We fork execution of the PHP engine, replacing the string arguments with an XSS attack vector known to cause JavaScript execution in a large number of HTML contexts [25]. The modified PHP engine logs the argument values and the return values for all such function calls.

From these logs, we extract all the function calls that both took an attack vector as an argument and returned a string. By Definition 2, these functions are sanitizers. However, we want to approximate the fact that these are sanitizers that provide HTML context chain safety as per Definition 4. Thus, we pass the return values of the sanitizers through an edit distance algorithm, comparing the attack vector string to the function's return value. We negate the effects of standard encoding functions like `htmlentities` before running the edit distance algorithm. If the edit distance is small, we mark the function as a sanitizer. While there are pathological cases in which our algorithm would fail, we manually evaluated the output of the algorithm for Wordpress and MediaWiki and found that the algorithm had very few false positives. At worst, the number of sanitizers we find is an *undercount* of the actual number of sanitizers in the web application.

Context Inference. Our aim is to identify the context in which an untrusted input was embedded. To define untrusted data, we use the intuition that sanitized input data is very likely untrusted. Taking the list of extracted sanitizers for a particular web application, we instrument the sanitizers in the

application to surround untrusted data with a special identifier markup. We feed all GET, POST and COOKIE inputs a specific key string, so that the instrumentation in the sanitizers can distinguish untrusted inputs from other internal data. We use techniques similar to taint-inference [52] to identify where untrusted inputs are embedded in output. Finally, we log all web application output for context inference.

We use a modified Python HTML5 parser to parse the application output. While parsing, we search all contexts for the special identifying markup. When found, we record the context the parser is in. We repeat this until the document is completely parsed. When it is finished, we have a list of all contexts where the key string appeared as well as which sanitizers it went through. We present the contexts we found in Table 7.

4.2. Evaluating Sanitization Practices

We manually evaluated whether these applications predominantly use whitelisting sanitizers or blacklisting. Note that our goal is *not* to check whether sanitizers in these applications were correct, but only to illustrate what security-conscious applications are employing as a defense strategy.

We observed that the number of sanitizers varied from 18 to 119, which is much larger than the number of contexts we encountered. On inspection, we did not find any duplication – this implies that applications have multiple sanitizers applied even for the same contexts. We characterize this variation in sanitizer choice in Section 4.4.

To the extent of our analysis, we observe that the common pattern of sanitization in the PHP applications is the whitelist based canonicalization policy that we discussed previously. The applications commonly allow only a whitelist of context-specific syntactic constructs.

Example 1 WordPress allows fairly expressive HTML constructs in untrusted user content. Users can place comments on blog posts, and these comments can contain some HTML tags with HTML tag attributes for a more rich editing experience. Of course, WordPress applies a variety of sanitizers to this content to ensure its safety.

However, WordPress's sanitizers are a set of filters that take a whitelist of allowed tags, attributes, protocols, or a number of other options depending on the sanitization context, and ensure that nothing off the whitelist is syntactically present. These filters are sufficient for WordPress, and we see similar implementations in the other applications. ■

Evaluation Summary. We find that primarily all of the applications use a whitelisting-based canonicalization strategy for the HTML tag context, as discussed earlier in Section 2.5. For instance, we find that the `kses` library used in WordPress and Drupal, the `HTMLPurifier` library used in Moodle and the `PHPInputFilter` library used in Joomla all employ this strategy.

4.3. Evaluating Usage of Contexts

As Table 7 shows, the various output contexts where we observed untrusted user input is fairly small. For instance, in MediaWiki, a user may create or edit a wiki entry. Their new text passes through sanitizers and a database but eventually flows to the HTML context of the output in that particular wiki article. Thus, in the table, the "HTML" column is marked for MediaWiki because it allows untrusted content in the HTML tag content context.

The listed contexts are as follows:

- HTML tag content: The basic HTML content context between two tags. Note, however, that this does not include the content between script tags. Content between script tags is a separate JavaScript context because the JavaScript, not HTML, parser is applied to it
- URL: A context where a URL is expected, such as an anchor tag href attribute or an image tag src attribute. This also includes the document schema.
- JS String: A JavaScript context where data appears within a quoted JavaScript string. For example, if the program places a user name inside a JavaScript string to add to a DOM text node.
- Attr: A non-URL HTML attribute context, such as the name attribute or the alt attribute.

Evaluation Summary.

To the extent of our analysis, we find that the set of contexts in use by web applications matches well with the expressiveness supported by the most expressive web frameworks and previous research [59]. This implies that, in principle if the Ctemplate framework were available for PHP, its context support would be empirically sufficient for all the applications we study.

Another interesting point is that applications do not emit data in HTML contexts that are inherently hard to secure. For example, we did not find any untrusted content flows to the general, non-string JavaScript context. Similarly, all the applications refrained from embedding untrusted data in unquoted HTML attributes, which are hard to sanitize due to browser variation and a large set of attack vectors.

4.4. Evaluating Expressiveness of Sanitization Policies

As we observe in Table 7, there is more than one sanitizer for each embedding context. This implies that each web application may sanitize different fragments of untrusted data in a different way based on its security policy. This notion of "picking a sanitizer" for each fragment of untrusted data before outputting it to the HTTP response is what we formulate as a *sanitization policy function* below.

Definition 5: (**Sanitization Policy Function**) The application's sanitization policy function ψ is an n-ary function

$$\psi : (I_1, I_2, \ldots I_n) \rightarrow S^*$$

where S is the set of sanitizers in the application and $I_1, \ldots, I_n$ are application-specific policy attributes.

Note that the sanitization policy function returns an ordered list of sanitizers to apply, and could even return with an empty list.

One of our goals is to identify what the policy attributes are for real applications, how much they vary across applications and how complex they may be. We argue that the larger the n, the more complex the decision for picking a sanitizer.

We find that an application's sanitization policy can be very complex. Specifically, how an application *selects* a sanitizer can be more complex than simply looking at the syntactic context. This real-world example from the Drupal application illustrates the complexity.

Example 2 In the content management system Drupal, one of the default behaviors is the ability to post and reply to comments on a page. By default, any logged in user can do this, but one may enable it for "anonymous" (not logged in) users as well.

When the user creates or replies to a post, the application presents a choice of HTML filters to apply to their post. The set of roles assigned to a user by the system administrator determines the choice of filters presented to the user. By default, "authenticated user" is the sole role of a logged in user, but the administrator may assign an arbitrary number of roles. Similarly, the program assigns the "anonymous user" role to anonymous users.

An administrator may assign HTML filters to each role. By default, the "authenticated user" role is only assigned the "Full HTML" filter that filters out all HTML control structures. However, the administrator can create arbitrary HTML filter policies and assign them to various roles. Thus, the administrator can apply the "Full HTML" filter to all users by default, but explicitly allow trusted users to post comments with links in HTML anchor tags. The administrator is not governed by roles and is always allowed to post arbitrary HTML.Thus, when a logged in user is posting a comment or a reply, a different sanitization function may be applied to her post based on her role. If she is an administrator, the program will not apply a filter, but if she is in a different role, the program may apply a restrictive filter.

Thus, context alone does not define the arity of the sanitization function policy. In this case, the role of the logged-in user increases the arity. That is, not just the context determines the sanitizer to apply (a comment or reply HTML context in this case) but also on the user's role.

However, in this instance, it is even more complicated, because the user may select any of the valid filters assigned by her roles. For example, if the user is both a trusted user for posting links and, separately, a trusted user for posting images, she may choose which filter (either the link or image filter) applies to her post. Therefore, the arity of the Drupal sanitization function policy again increases because user selection is also a deciding factor in sanitization function application. ■

Evaluation Summary. We find that sanitization policy functions are implicit in today's web application code. Specifically, we found that for many applications, sanitizer *selection* is more complex than simply looking at the syntactic HTML context, which is the functionality of the default auto-sanitization protections enabled in those web frameworks that support context-sensitive auto-sanitization. This implies that the sanitization policy functions are definitely beyond 0-ary functions (as context-insensitive auto-sanitization assumes), or unary functions (as assumed by context-sensitive auto-sanitization).

We have found several variations in policy functions in implicit use by these applications. One common sanitization policy attribute observed is that of *role-based* sanitization, where the application applies different sanity checks based on the privilege of the content. Drupal's variations in sanitizers applied based on the user's role is perhaps an extreme. Other simpler policies we observe are where the site administrator's content is not subject to sanitization (by design), as in the case of PHPBB3. For such simple policies, there are legitimate code paths which have no sanitization requirements.

Sanitization policies have a direct impact on the security offered by web frameworks. If a framework does not recognize and support the intended security model of the application, it cannot ensure that the sanitization is consistently applied in the application. Presently, this represents a gap between the expressiveness of the abstractions supported by web frameworks and what web applications require and implement.

5. Results and Findings

In this study, we characterized the nature of XSS sanitization. Using a formal notion of XSS sanitization allowed us to systematically investigate and contrast the nature of XSS sanitization in applications and frameworks. The following are some of our key findings:

1) Only 3 out of the 13 frameworks studied support all the contexts that complex web applications employ (Tables 4 and 7).
2) Many existing frameworks that provide auto-sanitization perform *context-insensitive* sanitization. With multiple examples, we demonstrated that context-insensitive sanitization can give a false sense of security. We also find that replacing auto-sanitization with the use of a manual sanitizer is not uncommon.
3) Newer frameworks support simple templating mechanisms for identifying untrusted data, defaulting to a fail-closed design. Such mechanisms put the burden of trusted data identification on the developer.
4) We find that context-sensitive sanitization is the norm across complex web applications.
5) Across frameworks and applications, the approach to sanitization generally adopted is whitelisting. Developers identified a subset of HTML that is sufficiently expressive for their needs. Variance in browser implementation does not seem to be a factor in any of the sanitizers we observed.

6) Many web applications employ a set of sanitizers much larger that the set of contexts they support.
7) We also find that the decision to pick a sanitizer is not a simple zero- or single-arity function; fine grained data attributes (e.g., is the source of the data an administrator?) along with application configuration often dictate the sanitizer picked for a particular flow.
8) None of the web frameworks we studied support the complex sanitization policies we observed in real world web applications.

It is not clear to us why such a large number of sanitizer functions are employed by web applications. We conjecture that in the absence of easy correctness arguments for sanitizers, applications err on the side of caution. Simple and practical mechanisms for correctness guarantees of sanitizers and sanitizer policies deserve research attention. The complete absence of support for complex multi-arity sanitizer policy in frameworks was also salient. We believe that the next generation of frameworks need to address this issue to achieve adoption by complex web applications — our notion of sanitization policy function is one possible abstraction that newer frameworks could employ.

6. Related Work

Cross-site scripting defense techniques have received a great deal of attention in research.

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 [5, 31, 33, 35, 36, 39, 43, 63, 65] but also more recently on JavaScript code [6, 23, 48, 49]. 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 [50] and XSS-GUARD [11] are two server-side solutions that have provided insight into context-sensitive sanitization. In particular, BLUEPRINT provides a deeper model of the web browser and points to paths between the browser components may vary across browsers. The browser model detailed in this work builds upon BLUEPRINT's model and more closely upon SCRIPTGARD's formalization. We provide additional details in our model to demystify the browser's parsing behavior and explain subtleties in sanitization which the prior work did not address.

Purely browser-based solutions, such as XSSAuditor, and client-only solutions, such as DSI, are implemented in modern browsers. These mechanisms are useful in nullifying common attack scenarios by observing 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. [10]. Other language-based solutions for customizable XSS security policies are also an area of active research [41]. Research shows that cross-site scripting attacks sometimes results from unsafe parsing of CSS [30], optimistic content-sniffing algorithms in browsers [7], and from vulnerabilities in extensions [6, 9]. Failure to isolate mashups and advertisements may also result in code injection vulnerabilities, but typically the safety properties that these attacks violate are treated as a separate class From XSS vulnerabilities. These violated properties include isolation of principles in web browser primitives [61], authority safety [38] and statically verified containment [22].

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 a modified browser ensures that XSS attacks can not result from parsing the untrusted data. While these proposals are encouraging, they require updates in browser implementations as well as server-side code. The closest practical implementation of such client-server defense architecture is the recent *content security policy* specification [55].

Correctness of Sanitization. While several systems have analyzed server-side code, the SANER [5] system empirically showed that custom sanitization routines in web applications can be error-prone. FLAX [49] and KUDZU [48] 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.

Among server-side defenses, BLUEPRINT provided a sanitization-free strategy for preventing cross-site scripting attacks, which involved the explicit construction of the intended parse tree in the browser via JavaScript. We observe that sanitization-free mechanisms stand in contrast to whitelist-based canonicalization sanitization which is what is generally implemented in emerging frameworks, the security of which has neither been fundamentally broken nor proven. Research on string analysis and other automata-based verification systems is currently active, and this research is directly relevant to these questions [27, 34, 48].

Techniques for Separating Untrusted Content. Taint-tracking based techniques aimed to address the problem of identifying and separating untrusted data from HTML output to ensure that untrusted data gets sanitized before it is output [14, 33, 43, 51, 60, 65]. Challenges in implementing taint-analysis as well as performance overheads have precluded their use in deployed web applications. Security-typed languages and type-systems offer another mechanism to ensure the robust isolation of untrusted data from HTML code output [15, 46, 53, 57]. The generality of type systems allows for creating a finer separation between untrusted inputs, a property we motivate with our empirical analysis. HTML templating engines, such

as those 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.

References

[1] G. Aas. CPAN: URI::Escape. http://search.cpan.org/~gaas/URI-1.56/URI/Escape.pm.

[2] Adsafe : Making javascript safe for advertising. http://www.adsafe.org/.

[3] How To: Prevent Cross-Site Scripting in ASP.NET. http://msdn.microsoft.com/en-us/library/ff649310.aspx.

[4] Microsoft ASP.NET: Request Validation – Preventing Script Attacks. http://www.asp.net/LEARN/whitepapers/request-validation.

[5] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.

[6] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. Vex: Vetting browser extensions for security vulnerabilities, 2010.

[7] A. Barth, J. Caballero, and D. Song. 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*, pages 360–371, Washington, DC, USA, 2009. IEEE Computer Society.

[8] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities, 2009.

[9] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities, 2010.

[10] D. Bates, A. Barth, and C. 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, New York, NY, USA, 2010. ACM.

[11] P. Bisht and V. N. Venkatakrishnan. Xss-guard: Precise dynamic prevention of cross-site scripting attacks. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 23–43, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] google-caja a source-to-source translator for securing javascript-based web content. http://code.google.com/p/google-caja/.

[13] CakePHP: Sanitize Class Info. http://api.cakephp.org/class/sanitize.

[14] E. Chin and D. Wagner. Efficient character-level taint tracking for java. In *Proceedings of the 2009 ACM workshop on Secure web services*, SWS '09, pages 3–12, New York, NY, USA, 2009. ACM.

[15] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proceedings of twenty-first ACM SIGOPS Symposium on Operating systems principles*, pages 31–44, New York, NY, USA, 2007. ACM.

[16] ClearSilver: Template Filters. http://www.clearsilver.net/docs/man_filters.hdf.

[17] CodeIgniter/system/libraries/Security.php. http://bitbucket.org/ellislab/codeigniter/src/tip/system/libraries/Security.php.

[18] CodeIgniter User Guide Version 1.7.2: Input Class. http://codeigniter.com/user_guide/libraries/input.html.

[19] Ctemplate: Guide to Using Auto Escape. http://google-ctemplate.googlecode.com/svn/trunk/doc/auto_escape.html.

[20] django: Built-in template tags and filters. http://docs.djangoproject.com/en/dev/ref/templates/builtins.

[21] Django sites : Websites powered by django. http://www.djangosites.org/.

[22] M. Finifter, J. Weinberger, and A. Barth. Preventing capability leaks in secure javascript subsets. In *Proc. of Network and Distributed System Security Symposium, 2010*.

[23] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for ajax intrusion detection. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 561–570, New York, NY, USA, 2009. ACM.

[24] Google Web Toolkit: Developer's Guide – SafeHtml. http://code.google.com/webtoolkit/doc/latest/DevGuideSecuritySafeHtml.html.

[25] G. Heyes. One vector to rule them all. http://www.thespanner.co.uk/2010/09/15/one-vector-to-rule-them-all.

[26] HipHop for PHP. http://github.com/facebook/hiphop-php/wiki.

[27] P. Hooimeijer and W. Weimer. A decision procedure for subset constraints over regular languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 188–198, June 2009.

[28] PHP Manual : html_entity_decode. http://php.net/manual/en/function.html-entity-decode.php.

[29] HTML Purifier : Standards-Compliant HTML Filtering. http://htmlpurifier.org/.

[30] L.-S. Huang, Z. Weinberg, C. Evans, and C. Jackson. Protecting browsers from cross-origin css attacks. In *ACM Conference on Computer and Communications Security*, 2010.

[31] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 40–52, New York, NY, USA, 2004. ACM.

[32] JiftyManual. http://jifty.org/view/JiftyManual.

[33] N. Jovanovic, C. Krügel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symposium on Security and Privacy*, 2006.

[34] A. Kieżun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A solver for string constraints. In *International Symposium on Software Testing and Analysis*, 2009.

[35] B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the Usenix Security Symposium*, 2005.

[36] B. Livshits, M. Martin, and M. S. Lam. SecuriFly: Runtime protection and recovery from Web application vulnerabilities. Technical report, Stanford University, Sept. 2006.

[37] B. Livshits, A. Nori, S. Rajamani, and A. Banerjee. Merlin: Specification inference for explicit information flow problems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2009.

[38] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pages 125–140, Washington, DC, USA, 2010. IEEE Computer Society.

[39] M. Martin and M. S. Lam. Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In *17th USENIX Security Symposium*, 2008.

[40] The Mason Book: Escaping Substitutions. http://www.masonbook.com/book/chapter-2.mhtml.

[41] L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *IEEE Symposium on Security and Privacy*, May 2010.

[42] Y. Nadji, P. Saxena, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. *Proceedings of the 16th Network and Distributed System Security Symposium*, 2009.

[43] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. *20th IFIP International Information Security Conference*, 2005.

[44] XSS Prevention Cheat Sheet. http://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet.

[45] PHP usage statistics. http://www.php.net/usage.php.

[46] W. Robertson and G. Vigna. Static enforcement of web application integrity through strong typing. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 283–298, Berkeley, CA, USA, 2009. USENIX Association.

[47] Ruby on Rails Security Guide. http://guides.rubyonrails.org/security.html.

[48] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, Washington, DC, USA, 2010. IEEE Computer Society.

[49] P. Saxena, S. Hanna, P. Poosankam, and D. Song. FLAX: Systematic dis-

covery of client-side validation vulnerabilities in rich web applications. In *17th Annual Network & Distributed System Security Symposium, (NDSS)*, 2010.

[50] P. Saxena, D. Molnar, and B. Livshits. Scriptgard: Preventing script injection attacks in legacy web applications with automatic sanitization. Technical report, Microsoft Research, September 2010.

[51] E. J. Schwartz, T. Avgerinos, and D. Brumley. 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*, SP '10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.

[52] R. Sekar. An efficient black-box technique for defeating web application attacks. In *NDSS*, 2009.

[53] J. Seo and M. S. Lam. Invisitype: Object-oriented security policies, 2010.

[54] Smarty Template Engine: escape. http://www.smarty.net/manual/en/language.modifier.escape.php.

[55] S. Stamm. Content security policy, 2009.

[56] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 372–382, New York, NY, USA, 2006. ACM.

[57] N. Swamy, B. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 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] W. Venema. Taint support for PHP. ftp://ftp.porcupine.org/pub/php/php-5.2.3-taint-20071103.README.html, 2007.

[61] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in mashupos. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 1–16, New York, NY, USA, 2007. ACM.

[62] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 32–41, New York, NY, USA, 2007. ACM.

[63] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the Usenix Security Symposium*, 2006.

[64] xssterminate. http://code.google.com/p/xssterminate/.

[65] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, pages 121–136, 2006.

[66] Yii Framework: Security. http://www.yiiframework.com/doc/guide/1.1/en/topics.security.

[67] Zend Framework: Zend_Filter. http://framework.zend.com/manual/en/zend.filter.set.html.