Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense

Yacin Nadji*
Illinois Institute of Technology
Chicago, IL, USA
yacin@ir.iit.edu

Prateek Saxena
University of California
Berkeley, CA, USA
prateeks@eecs.berkeley.edu

Dawn Song
University of California
Berkeley, CA, USA
dawnsong@cs.berkeley.edu

Abstract

Cross-site scripting (or XSS) has been the most dominant class of web vulnerabilities in 2007. The main underlying reason for XSS vulnerabilities is that web markup and client-side languages do not provide principled mechanisms to ensure secure, ground-up isolation of user-generated data in web application code. In this paper, we develop a new approach that combines randomization of web application code and runtime tracking of untrusted data both on the server and the browser to combat XSS attacks. Our technique ensures a fundamental integrity property that prevents untrusted data from altering the structure of trusted code throughout the execution lifetime of the web application. We call this property document structure integrity *(or DSI). Similar to prepared statements in SQL, DSI enforcement ensures automatic syntactic isolation of inline user-generated data at the parser-level. This forms the basis for confinement of untrusted data in the web browser based on a server-specified policy.*

We propose a client-server architecture that enforces document structure integrity in a way that can be implemented in current browsers with a minimal impact to compatibility and that requires minimal effort from the web developer. We implemented a proof-of-concept and demonstrated that such DSI enforcement with a simple default policy is sufficient to defeat over 98% of the 5,328 real-world reflected XSS vulnerabilities documented in 2007, with very low performance overhead both on the client and server.

1 Introduction

Cross-site scripting (XSS) attacks have become the most prevalent threat to the web in the last few years. According to Symantec's Internet Threat Report, over 17,000 site-specific XSS vulnerabilities have been documented in 2007 alone, which constitute over 4 times as many as the traditional vulnerabilities observed in that period [36]. Web Application Security Consortium's XSS vulnerability report shows that over 30% of the web sites analyzed in 2007 were vulnerable to XSS attacks [42]. In addition, there exist publicly available XSS attack repositories where new attacks are being added constantly [43].

Web languages, such as HTML, have evolved from light-weight mechanisms for static data markup, to full blown vehicles for supporting dynamic code execution of web applications. HTML allows inline constructs both to embed untrusted data and to invoke code in higher-order languages such as JavaScript. Due to their somewhat ad-hoc evolution to support demands of the growing web, HTML and other web languages lack principled mechanisms to separate trusted code from inline data and to further isolate untrusted data (such as user-generated content) from trusted data. As a result, web developers pervasively use fragile input validation and sanitization mechanisms, which have been notoriously hard to get right and have lead to numerous subtle security holes. We make the following observations explaining why it is not surprising that XSS vulnerabilities plague such a large number of web sites.

Purely server-side defenses are insufficient. Server-side validation of untrusted content has been the most commonly adopted defense in practice, and a majority of defense techniques proposed in the research literature have also focused on server-side mitigation [3, 44, 5, 16, 25, 22]. A common problem with purely server-side mitigation strategies is the assumption that parsing/rendering on the client browser is consistent with the server-side processing. In practice, this consistency has been missing. This weakness has been targeted by several attacks recently. For example, one such vulnerability [34] was found in Facebook in 2008. The vulnerability is that the server-side XSS filter recognizes the ":" character as a namespace identifier separator, whereas the web browser (Firefox v $<$ 2.0.0.2) strip it as a whitespace character. As a result, a string such as `<img src=''...'' onload:=attackcode>` is interpreted by the browser as `<img src=''...''`

*This work was done while the author was visiting UC Berkeley.

Browser	# Attacks
Internet Explorer 7.0	49
Internet Explorer 6.0	89
Netscape 8.1-IE Rendering	89
Netscape 8.1-Gecko Rendering	47
Firefox 2.0	45
Firefox 1.5	50
Opera 9.02	61
Netscape 4	5

Figure 1: XSS attacks vary significantly from browser to browser. A classification of 92 publicly available XSS attacks showing the diversity in the number of attacks that affect different web browsers [12].

`onload=attackcode>`, which executes `attackcode` as a Javascript code. In contrast, the Facebook XSS filter fails to recognize the attack string `attackcode` as code altogether. In general, it is problematic to expect the web server to accurately validate input data consistently with the browser, because actual browser behavior varies with browser implementation quirks and user configuration settings. Figure 1 highlights the diversity in the range of attacks that a user may be susceptible to depending on the browser implementation being used.

Integrity of client-side scripting code is subject to dynamic attacks. Several attacks target code injection vulnerabilities in client-side scripting code which processes untrusted data in an unsafe manner during its execution. Such attacks subvert the integrity of dynamic operations performed by web applications in the browser. Automatic XSS detection tools which employ server-side static analysis [22] or runtime analysis [44] are designed to identify attacks that target integrity of HTML code alone; these tools are severely handicapped as they do not model semantics of a diverse set of client-side languages supported by the browser. With the increasing popularity of AJAX applications such XSS vulnerabilities are a serious threat for Web 2.0 applications. The onus of eliminating such vulnerabilities places heavy burden on web developers who are ill-equipped to robustly detect them before deployment. One example of attacks that target dynamic processing of untrusted content by client-side JavaScript is the vulnerability [20] in the OnlineNow mechanism of MySpace.com web application. The OnlineNow mechanism provides dynamic online/offline status of a user's friends on MySpace. The vulnerability allows an attacker "friend" to place a crafted `<div>` tag below his/her picture, which when viewed by a victim causes a JavaScript `eval` statement to execute the attacker's code. Such vulnerabilities are not targeted at lack of validation in server-side scripting code (such as PHP); rather they target web application code written in client-side scripting languages.

XSS attacks are not limited to JavaScript injection and cookie-stealing. Attackers need not use JavaScript as a vector for script based attacks — attack vectors can be based on Flash (ActionScript), QuickTime, VBScript, CSS, XUL and even languages supported by web browser extensions. For instance, XSS attacks were demonstrated using certain features of the PDF language supported by Adobe Acrobat Reader plugin for the web browser [29]. Another observation worthy of note is that XSS vulnerabilities can result in damage well beyond automatic password or cookie theft. One compelling example of an attack that does not target cookie theft, is the recent XSS vulnerability on a banking web site reported by Netcraft [26]. Fraudsters exploited this vulnerability for a phishing attack, by injecting a modified login form (using an `iframe`) onto the bank's login page. This allows the attacker to steal the user's credentials by having them manually submit their credentials, rather than covertly stealing the password via a script.

Content validation is an error-prone mechanism. The most commonly used mechanism for preventing XSS is validation of untrusted data. *Sanitization* is one kind of validation which removes possibly malicious elements from untrusted data; *escaping* is another form which transforms dangerous elements so that they are prevented from being interpreted as special characters. Balzarotti et. al. [3] showed that sanitization mechanism is often insufficient to prevent all attacks, especially when web developers use custom built-in sanitization routines provided by popular scripting languages such as PHP. In general, there has been no "one-size-fits-all" sanitization mechanism, as validation checks change with the policy that the server wishes to enforce and no single primitive filters out dangerous content independent of the context where the untrusted data is inlined and used.

Defense Requirements. Based on these empirical observations, we formulate the following four requirements for a cross-site scripting defense.

1. The defense should not rely on server-side sanitization of untrusted data; instead it should form a second level of defense to safeguard against holes that result from error-prone sanitization mechanism.

2. The defense should confine untrusted data in a manner consistent with the browser implementation and user configuration.

3. The defense must address attacks that target server-side as well as client-side languages.

4. The defense should proactively protect against attacks without relying on detection of common symptoms of malicious activity such as cross-domain sensitive information theft.

Our Approach. In this paper, we develop an approach that significantly shifts the burden of preventing XSS attacks from the web developer to the web execution platform. Our approach can be implemented transparently in the web server and the browser requiring minimal web developer intervention and it provides a second line of defense for preventing XSS attacks. In our approach, XSS is viewed as a privilege escalation vulnerability, as opposed to an input validation problem. Sanitization and filtering/escaping of untrusted content aims to block or modify the content to prevent it from being interpreted as code. Our approach does not analyze the values of the untrusted data; instead, it restricts the interpretation of untrusted content to certain lexical and syntactic operations—more like a type system. The web developer specifies a restrictive policy for untrusted content, and the web browser enforces the specified policy.

To realize this system we propose a new scheme, which uses markup primitives for the server to securely demarcate inline user-generated data in the web document, and is designed to offer robustness in the face of an adaptive adversary. This allows the web browser to verifiably isolate untrusted data while initially parsing the web page. Subsequently, untrusted data is tracked and isolated as it is processed by higher-order languages such as JavaScript. This ensures the integrity of the document parse tree — we term this property as *document structure integrity* (or DSI). DSI is similar to PreparedStatements [9] which provide query integrity in SQL. DSI is enforced using a fundamental mechanism, which we call *parser-level isolation* (or PLI), that isolates inline untrusted data and forms the basis for uniform runtime enforcement of server-specified syntactic confinement policies.

We discuss the deployment of this scheme in a client-server architecture that can be implemented with a minimum impact to backwards compatibility in modern browsers. Our proposed architecture employs server-side taint tracking proposed by previous research to minimize changes to the web application code. We implemented a proof-of-concept that embodies our approach and evaluated it on a dataset of 5,328 web sites with known XSS vulnerabilities and 500 other popular web sites. Our preliminary evaluation demonstrates that parser-level isolation with a single default policy is sufficient to nullify over 98% of the attacks we studied. Our evaluation also suggests that our techniques can be implemented with very low false positives, in contrast to false positives that are likely to arise due to fixation of policy in purely client-side defenses. In comparison to existing XSS defenses, DSI enforcement offers a more comprehensive defense against attacks that extend beyond script injection and sensitive information stealing, and safeguards against both static as well as dynamic integrity threats.

Summary of Contributions. We outline the contributions of this paper below.

- We develop a new approach to XSS defense that provides principled isolation and confinement of inline untrusted data that has the following distinguishing features.
  - Employs a new markup randomization scheme, which is similar to instruction set randomization [17], to provide robust isolation in the face of an adaptive attacker.
  - Preserves the structural integrity of the web application code throughout its lifetime including during dynamic updates and operations performed by execution of client-side code.
  - Ensures that confinement of untrusted data is consistent with the browser processing.
  - Eliminates some of the main difficulties with server-side sanitization mechanism.
- We empirically show that DSI enforcement with a single default policy effectively thwarts over 98% of reflected real-world attack vectors we study. We discuss how the full implementation of client-server architecture could achieve these gains at very low performance costs and with almost no false positives.

2 XSS Definition and Examples

An XSS vulnerability is one that allows injection of untrusted data into a victim web page which is subsequently interpreted in a malicious way by the browser on behalf of the victim web site. This untrusted data could be interpreted as any form of code that is not intended by the server's policy, including scripts and HTML markup. We treat only user-generated input as untrusted and use the terms "untrusted data" and "user-generated data" interchangeably in this paper. We also refer to content as being either *passive*, i.e, consisting of elements derived by language terminals (such as string literals and integers)– or *active*, i.e, code that is interpreted (such as HTML and JavaScript).

Running Example. To outline the challenges of preventing exploits for XSS vulnerabilities, we show a toy example of a social networking site in Figure 2. The

```
 1:   <body>
 2:   <div id='WelcomeMess'> Welcome! </div>
 3:   <div id='$GET['FriendID-Status']' name='status'> </div>
 4:   <script>
 5:      if($GET['MainUser'])  {
 6:          document.getElementById('WelcomeMess').innerHTML =
 7:                          "Welcome" + "$GET['MainUser']";
 8:      }
 9:      var divname = document.getElementsByName("status")[0].id;
10:      var Name = divname.split("=")[0]; var Status = divname.split("=")[1];
11:      eval("divname.innerHTML = \"" + Name + " is " + Status + "\"");
12:   </script>
13:   </body>
```

Figure 2: Running example showing a snippet of HTML pseudocode generated by a vulnerable social networking web site server. Untrusted user data is embedded inline, identified by the `$GET['...']` variables.

	Untrusted variable	Attack String
Attack 1	`$GET['FriendID-Status']`	`' onmouseover=javascript:document.location="http://a.com"`
Attack 2	`$GET['MainUser']`	`</script><script>alert(document.cookie);</script>`
Attack 3	`$GET['FriendID-Status']`	`Attacker=Online"; alert(document.cookie);+"`
Attack 4	`$GET['MainUser']`	`<iframe src=http://attacker.com></iframe>`

Figure 3: Example attacks for exploiting vulnerabilities in Figure 2.

pseudo HTML code is shown here and places where untrusted user data is inlined are denoted by elements of `$GET['...']` array (signifying data directly copied from GET/POST request parameters). In this example, the server expects the value of `$GET['MainUser']` to contain the name of the current user logged into the site, and `$GET['FriendID-Status']` to contain a string with the name of another user and his status message ("online" or "offline") separated by a delimiter ("="). Assuming no sanitization is applied, this code has at least 4 places where vulnerabilities arise, which we illustrate with possible exploits[1] summarized in Figure 3.

- *Attack 1: String split & Attribute injection*. In this attack, the untrusted `$GET['FriendID-Status']` variable could prematurely break out of the `id` attribute of the `<div>` tag on line 3, and inject unintended attributes and/or tags. In this particular instance, the attack string shown in Figure 3 closes the string delimited by the single quote character, which allows the attacker to inject the `onmouseover` JavaScript event. The event causes the page to redirect to `http://a.com` potentially fooling the user into trusting the attacker's website.

  A similar attack is possible at line 7, wherein the attacker breaks out of the JavaScript string literal using an end-of-string delimiter (`"`) character in the value for the variable `$GET['MainUser']`.

- *Attack 2: Node splitting*. Even if this server sanitizes `$GET['MainUser']` on line 7 to disallow JavaScript end-of-string delimiters, another attack is possible. The attacker could inject a string to split the enclosing `<script>` environment, and then inject a new `script` tag, as shown by the second attack string in Figure 3.

- *Attack 3: Dynamic code injection*. A more subtle attack is one that targets the integrity of the `eval` query on line 11. Notice that JavaScript variable `Name` and `Status` are derived from parsing the untrusted `id` of the `div` element on line 3. Even if the server sanitizes the `$GET['FriendID-Status']` value for use in the `div` element context on line 3 by removing the `'` delimiter, the attacker could still inject code in the dynamically generated javascript `eval` statement. The vulnerability on line 10 parses the `id` attribute value of each div element into separate user name and status variables, which performs no sanitization for variable named `Status`. The attacker can use an attack string value as shown as the third string in Figure 3 to execute the arbitrary JavaScript code at line 11.

- *Attack 4: Dynamic active HTML update*. The attacker could inject active elements inside the `<div>`

[1]The sample attacks are illustrative of attacks seen in the past, and are not guaranteed to work on all browsers. See Figure 1 for more details.

with id `WelcomeMess` at line 6-7, by using the fourth attack string in Figure 3 as the value for `$GET['MainUser']`. This attack updates the web page DOM [2] tree dynamically on the client side after the web page has been parsed and the script code has been executed.

Motivation for our approach. We observe that all of the attacks outlined in Figure 3 require breaking the intended structure of the parse tree on the browser. The resulting parse trees from all attacks are shown superimposed in Figure 4. It is worth noting that attacks 1 and 2 break the structure of the web page during its initial parsing by the HTML and JavaScript parsers, whereas attack 3 and 4 alter the document structure during dynamic client-side operations.

If the browser could robustly isolate untrusted data on the web page, then it can *quarantine* untrusted data with respect to an intended policy. In this example, the server wishes to coerce untrusted nodes to leaf nodes in the parse tree, by treating them as string literals. This disallows injection of any language non-terminal (possible active HTML/JavaScript content) in the web page.

These examples bring out the complexity in defending against attacks with sanitization alone. To reinforce our observations, it is easy to understand that server side sanitization would be hard to perform in a moderately large application. The application developer would need to understand all possible contexts in which the data could be used with respect to multiple languages. Sanitization for each kind of active content varies based on the policy that server wishes to enforce, and could also vary based on the target browser's mechanism for rendering. Attacks need not be JavaScript based and may target a variety of goals (scripting with Flash and click fraud[3], in addition to sensitive information stealing).

3 Approach Overview

Web pages are parsed by various language parsers that are part of the web browser into internal parse trees. Under a benign query, the web server produces a web page that when parsed, results in a parse tree with a certain structure. This parse tree represents the structure that the web server aims to allow in the web document, and hence we term it as the *document structure*. In our approach, we ensure that the browser can identify and isolate nodes derived from user-generated data, in the parse tree during parsing. In principle, we whitelist the intended document structure and prevent the untrusted nodes from changing this structure in unintended ways. We call the property of ensuring intended document structure as enforcing *document structure integrity* or *DSI*.

We clearly separate the notion of a *confinement policy* from the parser-level isolation mechanism. As in our running example, web sites often wish to restrict untrusted data to leaf nodes in the document structure, as this is an effective way to stop an attacker from injecting active content. We refer to this confinement policy as *terminal confinement*, i.e., confinement of untrusted data to leaves in the document structure, or equivalently, to strings derived from terminals in the grammar representing valid web pages. Figure 5 is the parse tree obtained by DSI enforcement for our running example.

The server may wish to instruct the browser to enforce other higher-level semantic policy, such as specifying a restricted sandbox, but this is possible only if the underlying language or application execution framework provides primitives that prevent an attacker can from breaking out of the confinement region. For instance, the new proposal of *sandbox* attributes for `iframe` tags (introduced in HTML 5 [40]) defines semantic confinement policies for untrusted data from another domain. However, it relies on the `iframe` abstraction to provide the isolation. Similar to `iframes`, DSI forms the basis for higher level policy specification on web page regions that contain inline untrusted data. Our isolation primitives have no dependence on escaping/quoting or input sanitization for their internal working, thus making our mechanism a strong second line of defense for input validation checks already being used in web application code.

Key challenges in ensuring DSI in web applications. The high-level concept of terminal confinement has been proposed to defend against attacks such as SQL injection [35], but HTML differs from SQL in two significant ways. First, HTML can embed code written in various higher-order languages which share the same inline data. For instance, there are both generic (such as JavaScript URI) and browser-specific ways to invoke functions in VBScript, XUL, JavaScript, CSS and so on. To account for this difficulty, we treat the document structure as that implied by the superimposition of the parse trees obtained from code written in all languages (including HTML, JavaScript) used in a web page.

A second distinguishing challenge in securing web applications, specially AJAX driven applications, is that the document parse trees can be dynamically generated and updated on the client side. In real web pages, code in client-side scripting languages parses web content asynchronously, which results in repeated invocations of different language parsers. To address dynamic parsing, we treat the document structure as having two different components—a static component and a dynamic one. A

[2] DOM is the parse tree for the HTML code of the web page

[3] Using XSS to trick the user into clicking a "pay-per-click" link or advertisement through injecting HTML [11].

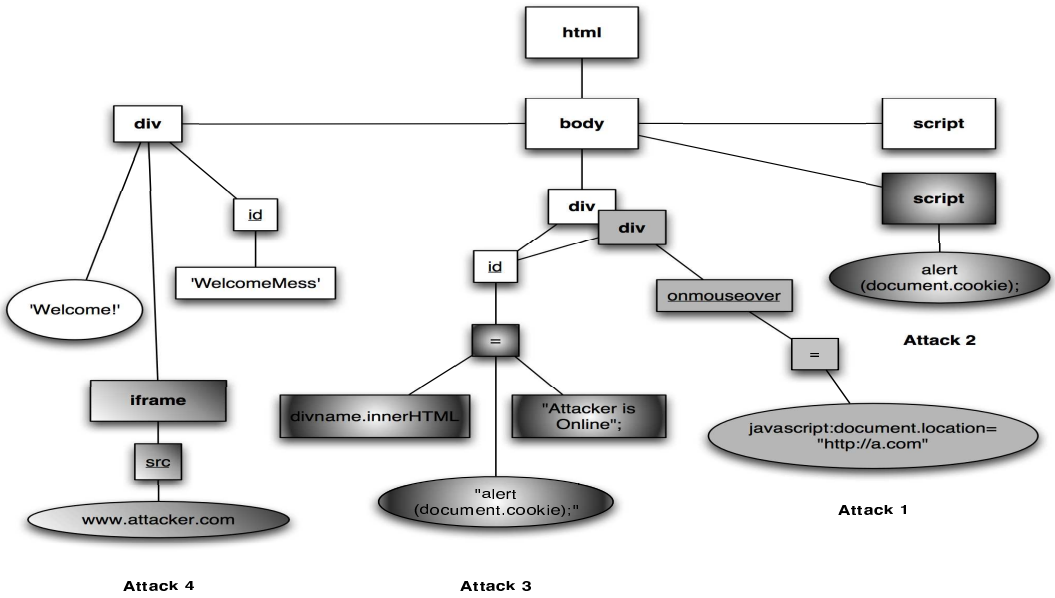


Figure 4: Coalesced parse tree for the vulnerable web page in Figure 2 showing superimposition of parse trees resulting from all attacks simultaneously. White node show the valid intended nodes whereas the dark nodes show the untrusted nodes inserted by the attacker.

web page must have a *static document structure*, i.e., the document structure implied by the parse tree obtained from the initial web page markup received by the browser. Similarly, a web page also has a *dynamic document structure*, i.e., the structure implied by the set of parse trees created by different parsers dynamically. To illustrate the distinction, we point out that attacks 1 and 2 in our running example violate static DSI, whereas attacks 3 and 4 violate dynamic DSI.

Goals. Parser-level isolation is a set of mechanisms to ensure robust isolation of untrusted data in the document structure throughout the lifetime of the web application. Using PLI we outline three goals that enforce DSI for a web page with respect to a server-specified policy, say P. First, we aim to enforce *static DSI with respect to* P, from the point web page is generated by the server to the point at which it is parsed into its initial parse trees in the browser. As a result, the browser separates untrusted data from trusted data in its initial parse tree robustly. Second, we aim to enforce *dynamic DSI with respect to* P in the browser, across all subsequent parsing operations. Third, we require that the attacker can not evade PLI by embedding untrusted content that results in escalated interpretation of untrusted data. These three goals enforce DSI based on uniform parser-level isolation.

Outline of Mechanisms. We view the operation of encoding the web page in HTML, merely as *serialization* (or marshaling[4]) of the content and the static document structure on the server side, and browser-side parsing of HTML as the deserialization step. We outline 4 steps that implement PLI and ensure the document structure is reconstructed by the browser from the point that the web server generates the web page.

Server-side. There are two steps that the server takes.

- *Step 1—Separation of trusted and user-generated data*. As a first step, web servers need to identify untrusted data at their output interface, and should distinguish it from trusted application code. We make this assumption to begin with, and discuss some ways to achieve this step through automatic methods in Section 5. We believe that this is not an unrealistic assumption—previous work on automatic dynamic taint tracking [44, 27] has shown that tracking untrusted user-generated data at the output interface is possible; in fact, many popular server-side scripting language interpreters (such as PHP) now have built-in support for this. Our goal in subsequent steps is to supplement integrity preserving primitives to ensure that the server-specified policy is correctly enforced in the client browser, instead of the sanitization at the server output interface for reasons outlined in Section 1.

[4] akin to serialization in other programming languages and RPC mechanisms

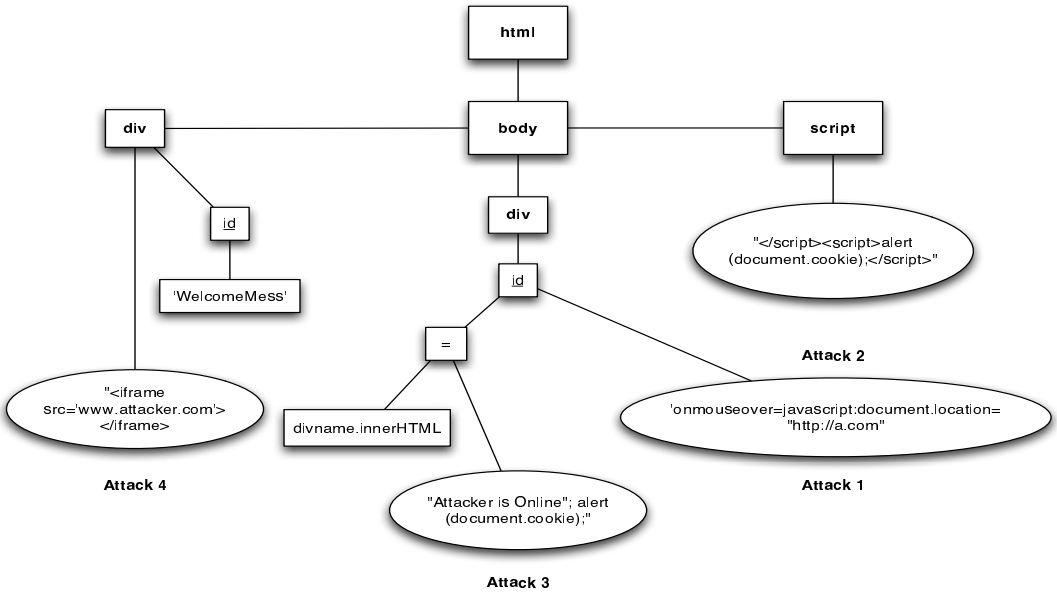


Figure 5: Coalesced parse tree (corresponding to parse tree in Figure 4) resulting from DSI enforcement with the terminal confinement policy—untrusted subtrees are forced into leaf nodes.

- *Step 2—Serialization: Enhancement of static structure with markup*. The key to robust serialization is to prevent embedded untrusted data from subverting the mechanism that distinguishes trusted code from inline untrusted data in the browser. To prevent such attacks, we propose the idea of markup randomization, i.e., addition of non-deterministic changes to the markup. This idea is similar to instruction set randomization [17] proposed for preventing traditional vulnerabilities.

Browser-side. There are two steps that the browser takes.

- *Step 3—Deserialization: Browser-side reconstruction of static document structure*. The web browser parses the web page into its initial parse tree, coercing the parse tree to preserve the intended structure. Thus, it can robustly identify untrusted data in the document structure at the end of the deserialization step.

- *Step 4—Browser-side dynamic PLI*. This step is needed to ensure DSI when web pages are dynamically updated. In essence, once untrusted data is identified in the browser at previous step, we initialize it as *quarantined* and track quarantined data in the browser dynamically. Language parsers for HTML and other higher-order languages like JavaScript are modified to disallow quarantined data from being used during parsing in a way that violates the policy. This step removes the burden of having the client-side code explicitly check integrity of the dynamic document structure, as it embeds a reference monitor in the language parsers themselves. Thus, no changes need to be made to existing client-side code for DSI-compliance.

4 Enforcement Mechanisms

We describe the high level ideas of the mechanisms in this section. Concrete details for implementing these are described in Section 5.

4.1 Serialization

Web pages are augmented with additional markup at the server's end, in such a way that the browser can separate trusted structural entities from untrusted data in the static document structure. We call this step serialization, and it is ideally performed at the output interface of the web server.

Adaptive Attacks. One naive way to perform serialization is to selectively demarcate or annotate untrusted data in the web page with special markup. The key concern is that an adaptive attacker can include additional markup to evade the isolation. For instance, let us say that we embed the untrusted data in a contained region with a special tag that disallows script execution that looks like:

```
<div class="noexecute">
    possibly-malicious content
</div>
```

```
...
3 :  <div id="⟦5367 GET['FriendId-Status'] ⟧5367">
4 :  <script>
5 :  if (⟦3246 GET['MainUser'] ⟧3246) {
...
```

Figure 6: Example of minimal serialization using randomized delimiters for lines 3-5 of the example shown in Figure 2.

This scheme is proposed in BEEP [15]. As the authors of BEEP pointed out, this naive scheme is weak because an adaptive attacker can prematurely close the `<div>` environment by including a `</div>` in a node splitting attack. The authors of BEEP suggest an alternative mechanism that encodes user data as a JavaScript string, and uses server-side quoting of string data to prevent it from escaping the JavaScript string context. They suggest the following scheme:

```
<div class="noexecute" id="n5"></div>
<script>
 document.getElementById("n5").innerHTML =
 "quoted possibly-malicious content";
</script>
```

We point out that it can be tricky to prevent the malicious content from breaking out of even the simple static JavaScript string context. It is not sufficient to quote the JavaScript end-of-string delimiters (") – an attack string such as `</script><iframe>...</iframe>` perpetrates a node splitting attack closing the script environment altogether, without explicitly breaking out the string context. Sanitization of HTML special characters `<,>` might solve this instance of the problem, but a developer may not employ such a restrictive mechanism if the server's policy allows some form of HTML markup in untrusted data (such as `<p>` or `<b>` tags in user content).

Our goal is to separate the isolation mechanism from the policy. The above outlined attack reiterates that content server-side quoting or validation may vary depending upon the web application's policy and is an error-prone process; keeping the isolation mechanism independent of input validation is an important design goal. We propose the following serialization schemes as an alternative.

Minimal Serialization. In this form of serialization, only the regions of the static web page that contain untrusted data are surrounded by special delimiters. Delimiters are added around inlined untrusted data independent of the context where the data is embedded. For our running example shown in the Figure 2, the serialization step places these delimiters around all occurrences of the `GET` array variables. If the markup elements used as delimiters are statically fixed, an adaptive attacker could break out of the confinement region by embedding the ending special delimiter in its attack string as discussed above. We propose an alternative mechanism called *markup randomization* to defend against such attacks.

The idea is to generate randomized markup values for special delimiters each time the web page is served, so that the attacker can not deterministically guess the confining context tag it should use to break out. Abstractly, the server appends a integer suffix $c, c \in C$ to a matching pair $[\![\ ]\!]$ of delimiters enclosing an occurrence of untrusted data, to generate $[\![_c\ ]\!]_c$ while serializing. The set C is randomly generated for each web page served. C is sent in a confidential, tamper-proof communication to the browser along with the web page. Clearly, if we use a pseudo-random number generator with a seed C_s to generate C, it is sufficient to send $\{C_s, n\}$, where n is the number of elements in C obtained by repeated invocations of the pseudo-random number generator. In the Figure 6 , we show the special delimiters added to the lines 3-5 of our running example in Figure 2. One instance of a minimal serialization scheme is the tag matching scheme proposed in the informal `jail` tag[7], which is formally analyzed by Louw et. al. [21].

Full Serialization. An alternative to minimal serialization is to mark all trusted structural entities explicitly, which we call full serialization. For markup randomization, the server appends a random suffix $c, c \in C$, to each trusted element (including HTML tags, attributes, values of attributes, strings) and so on.

Though a preferable mechanism from a security standpoint, we need a scheme that can mark trusted elements independent of the context of occurrence with a very fine granularity of specification. For instance, we need mechanism to selectively mark the `id` attribute of the `div` element of line 3 in the running example (shown in Figure 2) as trusted (to be able to detect attribute injection attacks), without marking the attribute value as trusted. Only then can we selectively treat the value part as untrusted which can be essential to detect dynamic code injection attacks, such as attack 3 in Figure 3.

Independently and concurrent with our work, Gundy et. al. have described a new randomization based full serialization scheme, called Noncespaces [10] that uses XML namespaces. However, XML namespaces does not have the required granularity of specification that is described above, and hence we have not experimented with this scheme. It is possible, however, to apply the full serialization scheme described therein as part of our architecture as well, sacrificing some of the dynamic integrity protection that is only possible with a finer-grained specification. We do not discuss full serialization further, and interested readers are referred to Noncespace [10] for details.

$V \longrightarrow [\![_c N]\!]_c$	$\{N.mark = Untrusted;\}$
$X \longrightarrow Y_1 Y_2$	{**if** $(X.mark == Untrusted)$
	then $(Y_1.mark = X.mark;$
	$Y_2.mark = X.mark;)$
	else $(Y_1.mark = Trusted;$ $\}$
	$Y_2.mark = Trusted;)$

Figure 7: Rules for computing `mark` attributes in minimal deserialization.

```
...
3 :  <div id="[[5367..  [[2222...  ]]5367">
4 :  <script>
5 :  if ([[3246 ..  ]]2222...  ]]3246) {
...
```

Figure 8: One possible attack on minimal serialization, if C were not explicitly sent. The attacker provides delimiters with the suffix 2222 to produce 2 valid parse trees in the browser.

4.2 Deserialization

When the browser receives the serialized web page, it first parses it into the initial static document structure. The document parse tree obtained from deserialization can verifiably identify the untrusted nodes.

Minimal deserialization . Conceptually, to perform deserialization the browser parses as normal, except that it does special processing for randomized delimiters $[\![_c$, $]\!]_c$. It ensures that the token corresponding to $[\![_c$ matches the token corresponding to $]\!]_c$, iff their suffixes are the same random value c and $c \in C$. It also marks the nodes in the parse tree that are delimited by special delimiters as untrusted.

Algorithm to mark untrusted nodes. Minimal deserialization is a syntax-directed translation scheme, which computes an inherited attribute, `mark`, associated with each node in the parse tree, denoting whether the node is `Trusted` or `Untrusted`. For the sake of conceptual explanation, let us assume that we can represent valid web pages that the browser accepts by a context-free grammar G [5].*Let $G = \{V, \Sigma, S, P\}$, where V denotes non-terminals, Σ denotes terminals including special delimiters, S is the start symbol, and P is a set of productions. Assuming that C is the set of valid randomized suffix values, the serialized web page s obeys the following rules:*

(a) All untrusted data is confined to a subtree rooted at some non-terminal N, such that a production, $V \longrightarrow [\![_c N]\!]_c$, is in P.

(b) Productions of the form $V \longrightarrow [\![_{c_1} N]\!]_{c_2}, c_1 \neq c_2$ are not allowed in P.

(c) $\forall c \in C$, all productions of the form $V \longrightarrow [\![_c N]\!]_c$ are valid in P.

The rules to compute the inherited attribute `mark` *are defined in Figure 7, with* `mark` *attribute for S initialized to* `Trusted`.

Fail-Safe. Appending random suffixes does not lead to robust design by itself. Sending the set C of random values used in randomizing the additional markups adds robustness against attacker spoofing delimiters.

To see why, suppose C was not explicitly sent in our design. Consider the scenario where an adaptive attacker tries to confuse the parser by generating two valid parse trees. In Figure 8 the attacker embeds delimiter $[\![_{2222}$ in `GET['FriendId-Status']` and a matching delimiter $]\!]_{2222}$ in `GET['MainUser']`. There could be two valid parse trees—one that matches delimiters with suffix 5367 and 3246, and another that matches the delimiters with suffix 2222. Although, the browser could allow the former to be selected as valid as delimiter with 5367 is seen first earlier in the parsing, this is a fragile design because it relies on the server's ability to inject the constraining tag first and requires sequential parsing of the web page. In practice, we can even expect the delimiter placement may be imperfect or missing in cases. For instance in Figure 8, if the special delimiters with suffix 5367 were missing, then even if the server had sanitized `GET['FriendId-Status']` perfectly against string splitting attack (attack 1 in Section 2), the attacker possesses an avenue to inject a spurious delimiter tag $[\![_{2222}$. All subsequent tags placed by the server would be discarded in an attempt to match the attacker provided delimiter. The attacker's ability to inject isolation markup is a weakness in the mechanism which does not explicitly send C. The informal `<jail>` proposal may be susceptible to such attacks as well [7]. Our explicit communication of C alleviates this concern.

4.3 Browser-side dynamic PLI

Once data is marked untrusted, we initialize it as quarantined. With each character we associate a *quarantine bit*, signifying whether it is quarantined or not. We dynamically track quarantined metadata in the browser. Whenever the base type of the data is converted from the data type in one language to a data type in another, we preserve the quarantine bit through the type transformation. For instance, when the JavaScript code reads a string from the browser DOM into a JavaScript string, appropriate quarantine bit is preserved. Similarly, when a JavaScript string is written back to a DOM property, the corresponding HTML lexical entities preserve the dynamic quarantine bit.

[5]practical implementations may not strictly parse context-free grammars

Quarantine bits are updated to reflect data dependences between higher-order language variables, i.e. for arithmetic and data operations (including string manipulation), the destination variable is marked quarantined, iff any source operand is marked quarantined. We do not track control dependence code as we do not consider this a significant avenue of attack in benign application. We do summarize quarantine bit updates for certain functions which result in data assignment operations but may internally use table lookups or control dependence in the interpreter implementation to perform assignments. For instance, the JavaScript `String.fromCharCode` requires special processing, since it may use conditional switch statement or a table-lookup to convert the parameter bytes to a string elements. In this way, all invocations of the parsers track quarantined data and preserve this across data structures representing various parse trees.

Example. For instance, consider the attack 3 in our example. It constructs a parse tree for `eval` statement as shown in Figure 4. The initial string representing the terminal `id` on line 3 is marked quarantined by the deserialization step. With our dynamic quarantine bit tracking, the JavaScript internal representation of `divname.id` and the variables `divname`, `Name` and `Status` are marked quarantined. According to the terminal confinement policy, while parsing our mechanism detects that the variable `Status` contains a delimiter non-terminal "`;`". It coerces the lexeme "`;`" to be treated a terminal character rather than interpreting it as a separator non-terminal, thus nullifying the attack.

5 Architecture

In this section, we discuss the details of a client/server architecture that embodies our approach. We first outline the goals we aim to achieve in our architecture and then outline how we realize the different steps proposed in section 4.

5.1 Architecture Goals

We propose a client-server architecture to realize DSI. We outline the following goals for web sites employing DSI enforcement, which are most important to make our approach amenable for adoption in practice.

1. *Render in non-compliant*[6] *browsers, with minimal impact*. At least the trusted part of the document should render as original in non-compliant browsers. Most user-generated data is benign, so even inlined untrusted data should render with minimal impact in non-compliant browsers.

2. *Low false positives*. DSI-compliant browsers should raise very few or no false positives. A client-server architecture, such as ours, reduces the likelihood of false positives that arise from a purely-client side implementation of DSI (see Section 7).

3. *Require minimal web application developer effort*. Automated tools should be employed to retrofit DSI mechanisms to current web sites, without requiring a huge developer involvement.

5.2 Client-Server Co-operation Architecture

Identification of Untrusted data. Manual code refactoring is possible for several web sites. Several web mashup components, such as Google Maps, separate the template code of the web application from the untrusted data already, but rely on sanitization to prevent DSI attacks. Our explicit mechanisms would make this distinction easier to specify and enforce.

Automatic transformation to enhance the markup generated by the server is also feasible for several commercial web sites. Several server side dynamic and static taint-tracking mechanisms [44, 19, 38] have been developed in the past. Languages such as PHP, that are most popularly used, have been augmented to dynamically track untrusted data with moderate performance overheads, both using automatic source code transformation [44] as well as manual source code upgrades for PHPTaint [38]. Automatic mechanisms that provide taint information could be directly used to selectively place delimiters at the server output.

We have experimented with PHPTaint [38], an implementation of taint-tracking in the PHP 5.2.5 engine, to automatically augment the minimal serialization primitives for all tainted data seen in the output of the web server. We enable dynamic taint tracking of GET/POST request parameters and database pulls. We disable taint declassification of data when sanitized by PHP sanitization functions (since we wish to treat even sanitized data as potentially malicious). All output tainted data are augmented with surrounding delimiters for minimal serialization. Our modifications shows that automatic serialization is possible using off-the-shelf tools.

For more complex web sites that use a multi-component architecture, cross-component dynamic taint analysis may be needed. This is an active area of research and automatic support for minimal serialization at the server side would readily benefit from advances in this area. Recent techniques proposed for program analysis to identify taint-style vulnerabilities [22, 16] could help identify taint sink points in larger web application, where manual identification is hard. Similarly, Nanda et al. have recently shown cross-component dynamic taint tracking for the LAMP architecture is possible [25].

[6] Web browsers that are not DSI-compliant are referred to as *non-compliant*

Communicating valid suffixes. In our design it is sufficient to communicate $\{C_s, n\}$ in a secure way, where C_s is the random number generator seed to use and n is the number of invocations to generate the set C of valid delimiter suffixes. Our scheme communicates these as two special HTML tag attributes, (`seed` and `suffixsetlength`), as part of the HTML `head` tag of the web page. We assume that the server and the browser use the same implementation of the psuedo-random number generator. Once read by the browser, it generates this set for the entire lifetime of the page and does not recompute it even if the attacker corrupts the value of the special attributes dynamically. We have verified that this scheme is backwards compatible with HTML handling in current browsers, i.e, these special attributes are completely ignored for rendering in current browsers[7].

Choice of serialization alphabet for encoding delimiters. We discuss two schemes for encoding delimiters.

- We propose use of byte values from the Unicode Character Database [37] which are rendered as whitespace on the major browsers independent of the selected character set used for web page decoding. Our rationale for using whitespace characters is its uniformity across all common character sets, and the fact that this does not hinder parsing of HTML or script in most relevant contexts (including between tags, between attributes and values and strings). In certain exceptional contexts where these may hinder semantics of parsing, these errors would show up in pre-deployment testing and can easily be fixed. There are 20 such character values which can be used to encode start and end delimiter symbols. All of the characters, as shown in appendix A, render as whitespace on cuurent browsers. To encode the delimiters' random suffixes we could use the remaining 18 (2 are used for delimiters themselves) as symbols. Thus, each symbol can encode 18 possible values, so a suffix $\ell - symbols$ long, should be sufficient to yield an entropy of $\ell \times (lg(18))$ or $(\ell \times 4.16)$ bits.

  It should be clear that a compliant browser can easily distinguish pages served from a non-compliant web server to a randomization compliant web server—it looks at the `seed` attribute in the `<head>` element of the web page. When a compliant browser views a non-compliant page, it simply treats the delimiter encoding bytes as whitespace as per current semantics, as this is a non-compliant web page. When a compliant browser renders a compliant web page, it treats any found delimiter characters as valid iff they have valid suffixes, or else it discards the sequence of characters as whitespace (these may occur by chance in the original web page, or may be attacker's spoofing attempts). Having initialized the enclosed characters as untrusted in its internal representation, it strips these whitespace characters away. Thus, the scheme is secure whether the page is DSI-compliant or not.

- Another approach is to use special delimiter tags, `<qtag>`, with an attribute `check=`*`suffix`*, as well. Qtags have a lesser impact on readability of code than the above scheme. Qtags have the same encoding mechanism as `<jail>` tags proposed informally [7]. We verified that it renders safely in today's popular browsers in most contexts, but is unsuitable to be used in certain contexts such as within strings. Another issue with this scheme is that XHTML does not allow attributes in end tags, and so they don't render well in XHTML pages on non-compliant browsers, and may be difficult to accepted as a standard.

Policy Specification. Our policies confine untrusted data only. Currently, we support per-page policies that are enforced for the entire web page, rather than varying region-based policies. By default, we enforce the terminal confinement policy which is a default fail-close policy. In most cases, this policy is sufficient for several web sites to defend against reflected XSS attacks. A more flexible policy that is useful is to allow certain HTML syntactic constructs in inline untrusted data, such as restricted set of HTML markup in user blog posts. We support a whitelist of syntactic HTML elements as part of a configurable policy.

We allow configurable specification of whitelisted HTML construct names through a `allowuser` tag attribute for HTML `<meta>` tag which can have a comma-separated list of allowed tags. For instance, the following specification would allow untrusted nodes corresponding to the paragraph, boldface, line break elements, the attribute `id` (in all elements) and the anchor element with optional `href` attribute (only with anchor element) in parse tree to not be flagged as an exploit. The following markup renders properly in non-compliant browsers since unknown markup is discarded in the popular browsers.

```
<meta allowuser='p,b,br,@id,a@href'>
```

For security, untrusted data is disallowed to define `allowuser` tag without exception. Policy development and standardization of default policies are important problems which involve a detail study of common elements that are safe to allow on most web sites. However, we consider this beyond the scope of this paper, but deem worthy of future work.

[7] "current browsers" refers to: Safari, Firefox 2/3, Internet Explorer 6/7/8, Google Chrome, Opera 9.6 and Konqueror 3.5.9 in this paper.

6 Implementation

We discuss details of our prototype implementation of a PLI enabled web browser and a PLI enabled web server first. Next, we demonstrate an example forum application that was deployed on this framework requiring no changes to application code. Finally, we outline the implementation of a web proxy server used for evaluation in section 7.

DSI compliant browser. We have implemented a proof-of-concept PLI enabled web browser by modifying Konqueror 3.5.9. Before each HTML parsing operation, the HTML parsing engine identifies special delimiter tags. This step is performed before any character decoding is performed, and our choice of unicode alphabet for delimiters ensures that we deal with all character set encodings. The modified browser simulates a pushdown automaton during parsing to keep track of delimiter symbols for matching. Delimited characters are initialized as quarantined, which is represented by enhancing the type declaration for the character class in Konqueror with a quarantine bit. Parse tree nodes that are derived from quarantined characters are marked quarantined as well. Before any quarantined internal node is updated to the document's parse tree, the parser invokes the policy checker which ensures that the parse tree update is permitted by the policy. Any internal nodes that are not permitted by the policy are collapsed with their subtree to be treated as a leaf node and rendered as a string literal.

We modified the JavaScript interpreter in Konqueror 3.5.9 to facilitate automatic quarantine bit tracking and prevented tainted access through the JavaScript-DOM interface. The modifications required were a substantial implementation effort compared to the HTML parser modifications. Internal object representations were enhanced to store the quarantine bits and handlers for each JavaScript operation had to be altered to propagate the quarantine bits. The implemented policy checks ensure that quarantined data is only interpreted as a terminal in the JavaScript language.

DSI compliant server. We employed PHPTaint [38] which is an existing implementation dynamic taint tracking in the PHP interpreter. It enables taint variables in PHP and can be configured to indicate which sources of data are marked tainted in the server. We made minor modifications to PHPTaint to integrate in our framework. By default when untrusted data is processed by a built-in sanitization routine, PHPTaint endorses the data as safe and *declassifies*(or clears) the taint; we changed this behavior to not declassify taint in such situations even though the data is sanitized. Whenever data is echoed to the output we interpose in PHPTaint and surround tainted data with special delimiter tags with randomized values at runtime. For serialization, we used the unicode characters `U+2029` as a start-delimiter. Immediately following the start-delimiter are ℓ randomly chosen unicode whitespace characters, the *key*, from the remaining 18 unicode characters. We have chosen $\ell = 10$, though this is easily configurable in our implementation. Following the key is the end-delimiter `U+2028` to signify the key has been fully read.

Example application. Figure 9(a) shows a vulnerable web forum application, phpBB version 2.0.18, running on a vanilla Apache 1.3.41 web server with PHP 5.2.5 when viewed with a vanilla Konqueror 3.5.9 with no DSI enforcement. The attacker posts a post containing a script tag which results in a cookie alert. To prevent such attacks, we deployed the phpBB forum application on our DSI-compliant web server next. We required *no* changes to the web application code to deploy it on our prototype DSI-compliant web server. Figure 9(b) shows how the attack is nullified by our client-server DSI enforcement prototype which employs PHPTaint to automatically mark forum data (derived from the database) as tainted, enhances it with minimal serialization which enables a DSI-compliant version of Konqueror 3.5.9 to nullify the attack.

Client-side Proxy Server. For evaluation of the 5,328 real-world web sites, we could not use our prototype taint-enabled PHP based server because we do not have access to server code of the vulnerable web sites. To overcome this practical limitation, we implemented a client-side proxy server that approximately mimics the server-side operations.

When the browser visits a vulnerable web site, the proxy web server records all GET/POST data sent by the browser, and maintains state about the HTTP request parameters sent. The proxy essentially performs *content based tainting* across data sent to the real server and the received response, to approximate what the server would do in the full deployment of the client-server architecture.

The web server proxy performs a lexical string match between the sent parameter data and the data it receives in the HTTP response. For all data in the HTTP response that matches, the proxy performs minimal serialization (approximating the operations of a DSI-compliant server) i.e, it lexically adds randomized delimiters to demarcate matched data in the response page as untrusted, before forwarding it to the PLI enabled browser.

7 Evaluation

To evaluate the effectiveness and overhead of PLI and PLI enabled browsers we conducted experiments with two configurations. The first configuration consists of running

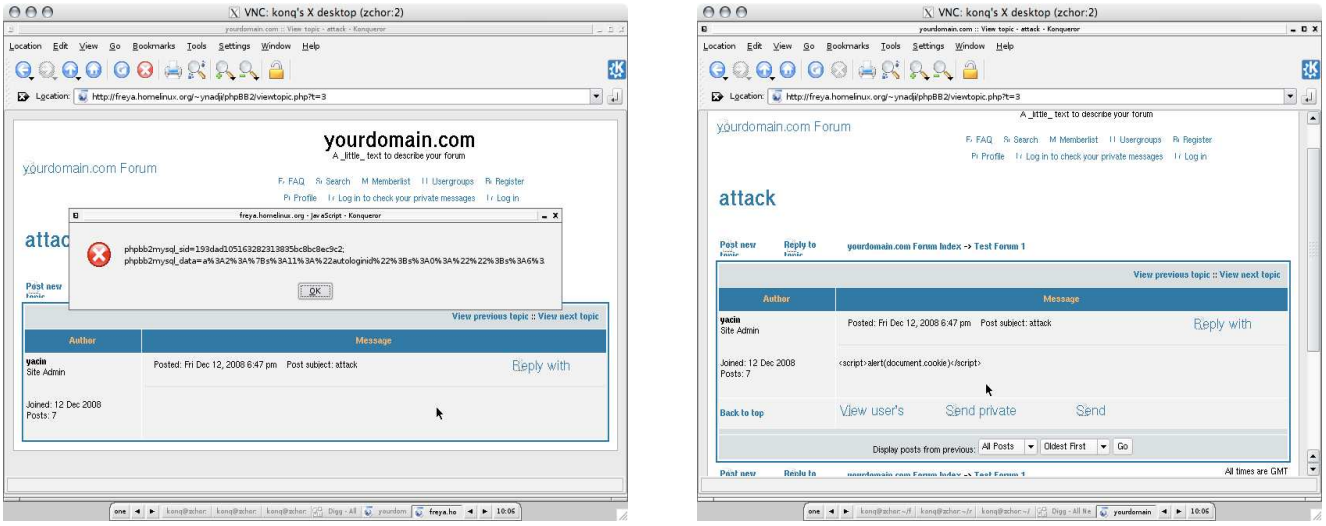


Figure 9: (a) A sample web forum application running on a vulnerable version of phpBB 2.0.18, victimized by stored XSS attack as it shows with vanilla Konqueror browser (b) Attack neutralized by our proof-of-concept prototype client-server DSI enforcement.

our prototype PLI enabled browser and a server running PHPTaint with the phpBB application. This configuration was used to evaluate effectiveness against stored XSS attacks. The second configuration ran our PLI enabled web browser directing all HTTP requests to the proxy web server described in section 7. The second configuration was used to study real-world reflected attacks, since we did not have access to the vulnerable web server code.

Attack Category	*# Attacks*	*# Prevented*
Reflected XSS	5,328	5,243 (98.4%)
Stored XSS	25	25 (100%)

Figure 10: Effectiveness of DSI enforcement against both reflected XSS attacks [43] as well as stored XSS attack vectors [12].

7.1 Experimental Setup

Our experiments were performed on two systems—one ran a Mac OS X 10.4.11 on a 2.0 GHz Intel processor with 2GB of memory, and the other runs Gentoo GNU/Linux 2.6.17.6 on a 3.4 GHz Intel Xeon processor with 2 GB of memory. The first machine ran an Apache 1.3.41 web server with PHP 5.2.5 engine and MySQL back-end, while the second ran the DSI compliant Konqueror. The two machines were connected by a 100 Mbps switch. We configured our prototype PLI enabled browser and server to apply the default policy of terminal confinement to all web requests unless the server overrides with another whitelisting based policy.

7.2 Experimental Results and Analysis

7.2.1 Attack Detection

Reflected XSS. We evaluated the effectiveness against all real-world web sites with known vulnerabilities, archived at the XSSed [43] web site as of 25th July 2008, which resulted in successful attacks using Konqueror 3.5.9. In this category, there were 5,328 web sites which constituted our final test dataset. Our DSI-enforcement using the proxy web server and DSI compliant browser nullified 98.4% of these attacks as shown in Figure 10. Upon further analysis of the false negatives in this experiment, we discovered that 46 of the remaining cases were missed because the real web server modified the attack input before embedding it on the web page—our web server proxy failed to recognize this server-side modification as it performs a simple string matching between data sent by the browser and the received HTTP response. We believe that in full-deployment these would be captured with server explicitly demarcating untrusted data. We could not determine the cause of missing the remaining 39, as the sent input was not discernible in the HTTP response web page. We showed that the policy of terminal confinement, if supported in web servers as the default, is sufficient to prevent a large majority of reflected XSS attacks.

Stored XSS. We setup a vulnerable version of phpBB web blog application (version 2.0.18) on our DSI enabled web server, and injected 30 benign text and HTML based

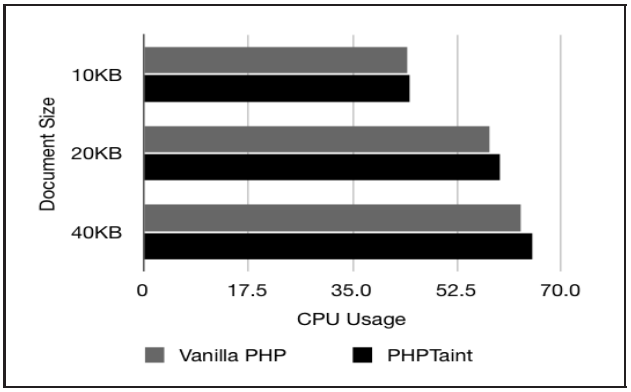


Figure 12: Increase in CPU overhead averaged over 5 runs for different page sizes for a DSI-enabled web server using PHPTaint [38].

posts, and all of the stored attack vectors taken from XSS cheat sheet [12] that worked in Konqueror 3.5.9. Of the 92 attack vectors outlined therein, only 25 worked in a vanilla Konqueror 3.5.9 browser. We configured the policy to allow only `<p>, <b> and <a>` HTML tags and and `href` attributes. No modifications were made to the phpBB application code. Our prototype nullified all 25 XSS attacks.

7.2.2 Performance

Browser Performance. To measure the browser performance overhead, we compared the page load times of our modified version of Konqueror 3.5.9 and the vanilla version of Konqueror 3.5.9. We evaluated against the test benchmark internally used at Mozilla for browser performance testing, consisting of over 350 web pages of popular web pages with common features including HTML, JavaScript, CSS, and images[24]. No data on this web pages was marked untrusted. We measured a performance overhead of 1.8% averaged over 5 runs of the benchmark.

We also measured the performance of loading all the pages from the XSSed dataset consisting of 5,328, with untrusted data marked with serialization delimiters. We observed a similar overhead of 1.85% when processing web pages with tainted data.

Web page (or code) size increase often translates to increased corporate bandwidth consumption, and is important to characterize in a cost analysis. For the XSSed dataset, our instrumentation with delimiters of length $\ell = 10$ increased the page size by less than 1.1% on average.

Server Performance. We measured the CPU overhead for the phpBB application running on a DSI compliant web server with PHPTaint enabled. This was done with `ab` (ApacheBench), a tool provided with Apache to measure performance [1]. It is configured to generate dynamic forum web pages of sizes varying from 10 KB to 40 KB. In our experiment, 64,000 requests were issued to the server with 16 concurrent requests. As shown in Figure 12, we observed average CPU overheads of 1.2%, 2.9% and 3.1% for pages of 10 KB, 20 KB, and 40 KB in size respectively. This is consistent with the performance overheads reported by the authors of PHPTaint [38]. Figure 11 shows a comparison between the vanilla web server and a DSI-compliant web server (both running phpBB) in terms of the percentage of HTTP requests completed within a certain response time frame. For 10 concurrent requests, the two servers perform nearly very similar, wheres for 30 concurrent requests the server with PHPTaint shows some degradation for completing more than 95% of the requests.

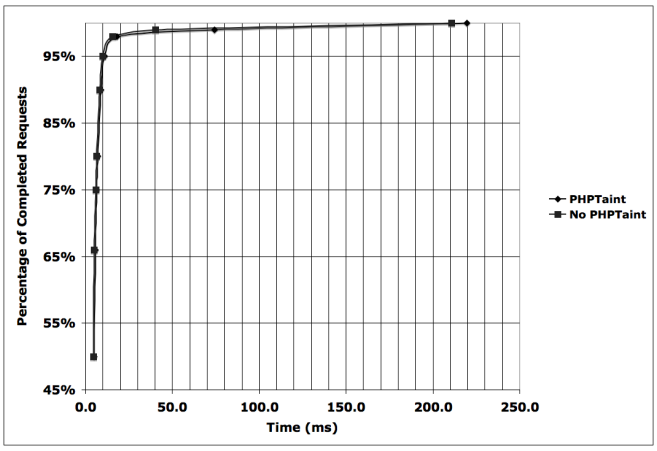
7.2.3 False Positives

We observed a fewer false positives rate in our stored XSS attacks experiment than in the reflected XSS experiment. In the stored experiment, we did not observe any false positives. In the reflected XSS experiment, we observed false positives when we deliberately provided inputs that matched existing page content. For the latter experiment, we manually browsed the Global Top 500 websites listed on Alexa [2] browsing with deliberate intent to raise false positives. For each website, we visited an average of 3 second-level pages by creating accounts, logging in with malicious inputs, performing searches for dangerous keywords, as well as clicking on links on the web pages to simulate normal user activity.

With our default policy, as expected, we were able to induce false positives on 5 of the web pages. For instance, a search query for the string "`<title>`" on Slashdot[8] caused benign data to be returned page to be marked quarantined. We confirmed that these arise because our client-side proxy server marks trusted code as untrusted which subsequently raises alarms when interpreted as code by the browser. In principle, we expect that full-implementation with a taint-aware server side component would eliminate these false positives inherent in the client-side proxy server approximation.

We also report that even with the client-side proxy server approximation, we did *not* raise false positives in certain cases where the IE 8 Beta XSS filter did. For instance, we do not raise false positives when searching for the string "javascript:" on Google search engine. This is because our DSI enforcement is parser context aware—though all occurrences of "javascript:" are marked untrusted in the HTTP response page, our browser did not raise an alert as untrusted data was not interpreted as code.

[8] http://slashdot.org



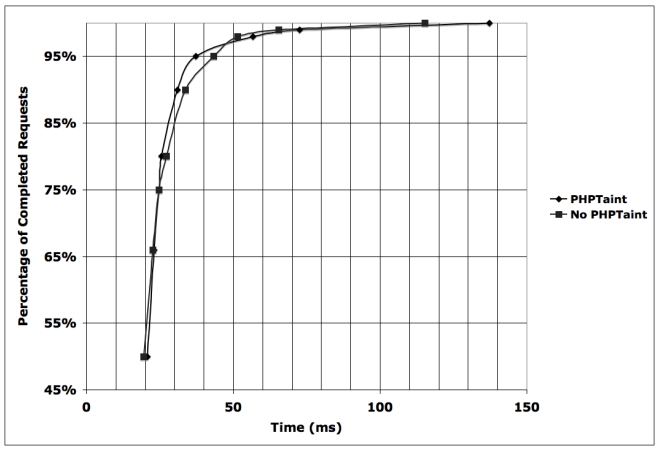


Figure 11: Percentage of responses completed within a certain timeframe. 1000 requests on a 10 KB document with (a) 10 concurrent requests and (b) 30 concurrent requests.

8 Comparison with Existing XSS Defenses

We outline the criteria for analytically comparing different XSS defenses first, and then discuss each of the existing defenses next providing a summary of the comparison in Figure 13.

8.1 Comparison Criteria

To concretely summarize the strengths and weaknesses of various XSS defense techniques, we present a defender-centric taxonomy of adaptive attacks to characterize the ability of current defenses against current attacks as well as attacks in the future that try to evade the defenses. Adaptive attackers can potentially target at least the avenues outlined below.

- *Browser inconsistency*. Inconsistency in assumptions made by the server and client lead to various attacks as outlined in the Section 1.
- *Lexical Polymorphism*. To evade lexical sanitization, attackers may find variants in lexical entities.
- *Keywords Polymorphism*. To evade keyword filters, attackers may find different syntactic constructs to bypass these. For instance, in the Samy worm [32], to inject a restricted keyword `innerHTML`, the attacker used a semantically equivalent construct "`eval('inner'+'HTML')`".
- *Multiple Injection Vectors*. Attacker can inject non-script based elements.
- *Breaking static structural integrity*. To specifically evade confinement based schemes, attacker can break out of the static confinement regions on the web page.
- *Breaking dynamic structural integrity*. Attacks may target breaking the structure of the dynamically executing client-side code, as discussed in Section 2.

Defense against each of the above adaptive attack categories serves a point of comparing existing defenses. In addition to these, we analytically compare the potential effectiveness of techniques to defend against stored XSS attacks. We also characterize whether a defense mechanism enables flexible server-side specification of policies or not. This is important because fixation of policies often results in false positives, especially for content-rich untrusted data, which can be a serious impediment to the eventual deployability of an approach.

8.2 Existing Techniques

Figure 13 shows the comparative capabilities of existing defense techniques at a glance on the basis of criteria outlined earlier in this section. We describe current XSS defenses and discuss some of their weaknesses.

8.2.1 Purely server-side defenses

Input Validation and sanitization. Popular server side languages such as PHP provide standard sanitization functions, such as `htmlspecialchars`. However, the code logic to check validity is often concentrated at the input interface of the server, and also distributed based on the context where untrusted data gets embedded. This mechanism serves as a first line of defense in practice, but is not robust as it places excessive burden on the web developer for its correctness. Prevalence of XSS attacks today shows that these mechanisms fail to safeguard against both static and dynamic DSI attacks.

Techniques	BI	P	MV	S DSI	D DSI	ST	FP
Purely Server-side							
Input Validation & Sanitization			✓			✓	✓
Server Output browser-independent policies (using taint-tracking)		✓	✓	✓		✓	✓
Server Output Validation browser-based policies (XSS-GUARD [5])		✓	✓	✓	✓	✓	✓
Purely Browser Side							
Sensitive Information Flow Tracking	✓	✓		✓	✓	✓	
Global Script Disabling	✓	✓		✓	✓	✓	
Personal Firewalls with URL Blocking	✓	✓			✓		
GET/POST Request content based URL blocking	✓	✓		✓		✓	
Browser-Server Cooperation Based							
Script Content Whitelisting (BEEP)	✓	✓		✓		✓	✓
Region Confinement Script Disabling (BEEP)	✓	✓		✓		✓	✓
PLI with Server-specified policy enforcement	✓	✓	✓	✓	✓	✓	✓

BI Not susceptible to browser-server inconsistency bugs
P Designed to easily defeats lexical and keyword polymorphism based attacks
MV Designed for comprehensiveness against multiple vectors and attack goals (Flash objects as scripting vectors, `iframe`s insertion for phishing, click fraud).
S DSI Designed to easily defeat evasion attacks that break static DSI (attacks such as 1,2 in Section 2).
D DSI Designed to easily defeat evasion attacks that break dynamic DSI (attacks such as 3,4 in Section 2).
ST Can potentially deal with stored XSS attacks.
FP Allows flexible server configurable policies (important to eliminate false positives for content-rich untrusted data)

Figure 13: Various XSS Mitigation Techniques Capabilities at a glance. Columns 2 - 6 represent security properties, and columns 7-9 represent other practical issues. A '✓' denotes that the mechanism demonstrates the property.

Browser-independent Policy Checking at Output. Taint-tracking [44, 25, 27, 30] on the server-side aims to centralize sanitization checks at the output interface with the use of taint metadata. Since the context of where untrusted data are being embedded can be arbitrary, the policy checking becomes complicated especially when dealing with attacks that affect dynamic DSI. The primary reason is the lack of semantics of client side behavior in the policy checking engine at the interface. Another problem with this approach is that the policy checks are not specific to the browser that the client uses and can be susceptible to browser-server inconsistency bugs.

Browser-based Policy Checking at Output. To mitigate the lack of client-side language semantics at the server output interface, XSS-GUARD [5] employs a complete browser implementation on the server output. In principle, this enables XSS-GUARD to deal with both static and dynamic DSI attacks, at the expense of significant performance overheads. However, this scheme conceptually still suffers from browser inconsistency bugs as a different target browser may be used by the client than the one checked against. Our technique enables the primary benefits of XSS-GUARD without high performance overheads and making the policy enforcement consistent with the client browser.

8.2.2 Purely client-side defenses

Sensitive information flow tracking. Vogt et. al. propose sensitive information flow tracking [39] in the browser to identify spurious cross-domain sensitive information transfer as a XSS attack. This approach is symptom targeted and limited in its goal, and hence does not lend easily to other attack targets outlined in the introduction. It also requires moderately high false positives in normal usage. This stems from the lack of specification of the intended policy by the web server.

Script Injection Blocking. Several techniques are focused on stopping script injection attacks. For instance, the Firefox NoScript extension block scripts globally on web sites the user does not explicitly state as trusted. Many web sites do not render well with this extension turned on, and this requires user intervention. Once allowed, all scripts (including those from attacks) can run in the browser.

Personal Firewalls with URL blocking. Noxes [18] is a client-side rule based proxy to disallow users visiting potentially unsafe URL using heuristics. First, such solutions are not designed to distinguish trusted data generated by the server from user-generated data. As a result, they can have high false negatives (Noxes treats static links in the page as safe) and have false positives [18] due to lack of server-side configuration of policy to be enforced. Second, they

are largely targeted towards sensitive information stealing attacks.

GET/POST Request content based URL blocking. Several proposals aim to augment the web browser (or a local proxy) to block URL that contain GET/POST data with known attack characters or patterns. The most recent is an implementation of this is the XSS filter in Internet Explorer (IE) 8 Beta [14]. First, from our limited experiments with the current implementation, this approach does not seem to detect XSS attacks based on the parsing context. This raises numerous false positives, one instance of which we describe in section 7. Second, their design does not allow configurable server specified policies, which may disallow content-rich untrusted data. In general, fixed policies on client-side with no server-side specification either raise false positives or tend to be too specific to certain attack vectors (thus resulting in false negatives). Finally, our preliminary investigation reveals that they currently do not defend against integrity attacks, as they allow certain non-script based attack vectors (such as forms) to be injected in the web page. We believe this is an interesting avenue and a detailed study of the IE 8 mechanism would be worthwhile to understand capabilities of such defenses completely.

8.2.3 Client-server cooperative defenses

This paradigm for XSS defense has emerged to deal with the inefficiencies of purely client and server based mechanisms. Jim et al. have recently proposed two approaches in BEEP [15]—whitelisting legitimate scripts and defining regions that should not contain any scripting code.

Whitelisting of legitimate scripts. First, they target only script-injection based vectors and hence are not designed to comprehensively defend against other XSS vectors. Second, this mechanism does not thwart attacks (such as attack 4 in Figure 3)violating dynamic DSI that target unsafe usage of data by client-side code. Their mechanism checks the integrity and authenticity of the script code before it executes, but does not directly extend to attacks that deal with the safety of data usage. Our technique enforces a dynamic parser-level confinement to ensure that data is not interpreted as code in client-side scripting code.

Region-based Script Disabling. BEEP outlined a technique to define regions of the web page that can not contain script code, which allows finer-grained region-based script disabling than those possible by already supported browser mechanisms [28]. First, their isolation mechanism using JavaScript string quoting to prevent static DSI attacks against itself. As discussed in Section 4.1, this mechanism can be somewhat tricky to enforce for content-rich untrusted data which allows HTML entities in untrusted data. Second, this mechanism does not deal with dynamic DSI attacks by itself, because region based script blocking can not be applied to script code regions.

9 Discussion

DSI enforcement using a client-server architecture offers a strong basis for XSS defense in principle. However, we discuss some practical concerns for a full deployment of this scheme. First, our approach requires both client and server participation in implementing our enhancements. Though we can minimize the developer effort for such changes, our technique requires both web servers and clients to collectively upgrade to enable any protection.

Second, a DSI-compliant browser requires quarantine bit tracking across operations of several languages. If implemented for Javascript, this would prevent attacks vectors using Javascript, but not against attacks that using other languages. Uniform cross-component quarantine bit tracking is possible in practice, but it would require vendors of multiple popular third party web plugins (Flash, Flex, Silverlight, and so on) to cooperate and enhance their language interpreters or parsers. Automatic techniques to facilitate such propagation and cross-component dynamic quarantine bit propagation at the binary level for DSI enforcement are interesting research directions for future work that may help address this concern.

Third, it is important to account for end-user usability. Our techniques aim to minimize the impact of rendering DSI compliant web pages on existing web browsers for ease of transition to DSI compliance; however, investigation of schemes that integrate DSI seamlessly while ensuring static DSI are important. Recent work but Louw et. al. formulates the problem of isolation of untrusted content in static HTML markup [21]; they present a comparison of prevalent isolation mechanisms in HTML and show that there is no single silver bullet. In contrast, we outline techniques that address static as well as dynamic isolation of untrusted data. We hope that our work provides additional insight for development of newer language primitives for isolation. Finally, false positives are another concern for usability. We did not encounter false positives in our preliminary evaluation and testing, but this not sufficient to rule out its possibility in a full-deployment of this scheme.

10 Related Work

XSS defense techniques can be largely classified into detection techniques and prevention techniques. The latter has been directly discussed in Section 8; in this section, we discuss detection techniques and other work that relates to ours.

XSS detection techniques focus on identifying holes in web application code that could result in vulnerabilities. Most of the vulnerability detection techniques have focused on server-side application code. We classify them based on the nature of the analysis, below.

- *Static and Quasi-static techniques*. Static analysis [13, 16, 23] and model checking techniques [22] aim to identify cases where the web application code fails to sanitize the input before output. Most static analysis tools are equipped with the policy that once data is passed through a custom sanity check, such as `htmpspecialchars` PHP function, then the input is safe. Balzarotti et al. [3] show that often XSS attacks are possible even if the develop performs certain sanitization on input data due to deficiencies in sanitization routines. They also describe a combined static and dynamic analysis to find such security bugs.

- *Server-side dynamic detection* techniques have been proposed to deal with the distributed nature of the server side checks. Taint-tracking [44, 5, 27, 30] on the server-side aims to centralize sanitization checks at the output interface with the use of taint metadata. These have relied on the assumption that server side processing is consistent with client side rendering, which is a significant design difference. These can be used as prevention techniques as well. Our work extends the foundation of taint-tracking to client-side tracking to eliminate difficulties of server-browser inconsistencies and to safeguard client-side code as well. Some of the practical challenges that we share with previous work on taint-tracking are related to tracking taint correctly through multiple components of the web server platform efficiently. Cross-component taint tracking [25] and efficient designs of taint-tracking [33, 31, 19] for server-side mitigation are an active area of research which our architecture would readily benefit from.

Several other works have targeted fortification of web browser's same-origin policy enforcement mechanisms to isolate entities from different domains. Browser-side taint tracking is also used to fortify domain isolation [8], as well as tightening the sharing mechanisms such as iframe communication[4] and navigation. These address a class of XSS attacks that arise out of purely browser-side bugs or weak enforcement policies in isolating web content across different web page, whereas in this paper, we have analyzed the class of reflected and stored XSS attacks only. MashupOS[41] discussed isolation and communication primitives for web applications to specify trust associated with external code available from untrusted source. Our work introduces primitives for isolation and confinement of inline untrusted data that is embedded in the web page.

Finally, the idea of parser-level isolation is a pervasively used mechanism. Prepared statements [9] in SQL are built on this principle, and Su et al. demonstrated a parser-level defense technique against SQL injection attacks[35]. As we show, for today's web applications the problem is significantly different than dealing with SQL, as untrusted data is processed dynamically both on the client browser and in the web server. The approach of using randomization techniques has been proposed for SQL injection attacks [6], control hijacking in binary code [17], and even in informal proposals for confinement in HTML using `<jail>` tag [7, 21]. Our work offers a comprehensive framework that improves on the security properties of `<jail>` element for static DSI (as explained in Section 4), and provides dynamic integrity as well.

11 Conclusion

We proposed a new approach that models XSS as a privilege escalation vulnerability, as opposed to a sanitization problem. It employs parser-level isolation for confinement of user-generated data through out the lifetime of the web application. We showed this scheme is practically possible in an architecture that is backwards compatible with current browsers. Our empirical evaluation over 5,328 real-world vulnerable web sites shows that our default policy thwarts over 98% of the attacks, and we explained how flexible server-side policies could be used in conjunction, to provide robust XSS defense with no false positives.

12 Acknowledgments

We are thankful to Adam Barth, Chris Karloff and David Wagner for helpful feedback and insightful discussions during our design. We also thank Robert O'Callahan for providing us with the Mozilla Firefox test suite and Nikhil Swamy for discussions during writing. We are grateful to our anonymous reviewers for useful feedback on experiments and suggestions for improving our work. This work is supported by the NSF TRUST grant number CCF-0424422, NSF TC grant number 0311808, NSF CAREER grant number 0448452, and the NSF Detection grant number 0627511.

References

[1] ab. Apache HTTP server benchmarking tool. `http://httpd.apache.org/docs/2.0/programs/ab.html`.

[2] alexa.com. Alexa top 500 sites. `http://www.alexa.com/site/ds/top_sites?ts_mode=global&lang=none`, 2008.

[3] 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.

[4] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. In *Proceedings of the 17th USENIX Security Symposium (USENIX Security 2008)*, 2008.

[5] P. Bisht and V. N. Venkatakrishnan. XSS-GUARD: precise dynamic prevention of cross-site scripting attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008.

[6] S. W. Boyd and A. D. Keromytis. Sqlrand: Preventing sql injection attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, pages 292–302, 2004.

[7] C. M. C. Brendan Eich. Javascript: Mobility & ubiquity. Presentation. `http://kathrin.dagstuhl.de/files/Materials/07/07091/07091.EichBrendan.Slides.pdf`.

[8] S. Chen, D. Ross, and Y.-M. Wang. An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 2–11, New York, NY, USA, 2007. ACM.

[9] H. Fisk. Prepared statements. `http://dev.mysql.com/tech-resources/articles/4.1/prepared-statements.html`, 2004.

[10] M. V. Gundy and H. Chen. Noncespaces: using randomization to enforce information flow tracking and thwart cross-site scripting attacks. *16th Annual Network & Distributed System Security Symposium*, 2009.

[11] R. Hansen. Clickjacking. `http://ha.ckers.org/blog/20081007/clickjacking-details/`.

[12] R. Hansen. Xss cheat sheet. `http://ha.ckers.org/xss.html`.

[13] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo. Securing web application code by static analysis and runtime protection. *DSN*, 2004.

[14] IE 8 Blog: Security Vulnerability Research & Defense. IE 8 XSS filter architecture and implementation. `http://blogs.technet.com/swi/archive/2008/08/18/ie-8-xss-filter-architecture-implementation.aspx`, 2008.

[15] T. Jim, N. Swamy, and M. Hicks. Beep: Browser-enforced embedded policies. *16th International World World Web Conference*, 2007.

[16] 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.

[17] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, 2003.

[18] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 2006 ACM symposium on Applied computing*, 2006.

[19] L. C. Lam and T. Chiueh. A general dynamic information flow tracking framework for security applications. In *Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, 2006.

[20] J. Lavoie. Myspace.com - intricate script injection. www.derkeiler.com/pdf/Mailing-Lists/securityfocus/vuln-dev/2006-04/msg00016.pdf.

[21] M. T. Louw, P. Bisht, and V. Venkatakrishnan. Analysis of hypertext isolation techniques for XSS prevention. *Workshop on Web 2.0 Security and Privacy (W2SP)*, 2008.

[22] 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.

[23] M. C. Martin, V. B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

[24] Mozilla Foundation. Tp2 pageloader framecycle test. `http://mxr.mozilla.org/mozilla/source/tools/performance/pageload/`.

[25] S. Nanda, L.-C. Lam, and T. Chiueh. Dynamic multi-process information flow tracking for web application security. In *Proceedings of the 8th ACM/IFIP/USENIX international conference on Middleware*, 2007.

[26] Netcraft. Banks hit by cross-frame phishing attacks. `http://news.netcraft.com/archives/2005/03/17/banks_hit_by_crossframe_phishing_attacks.html`, 2005.

[27] 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.

[28] NoScript. Noscript. `http://noscript.net/`, 2008.

[29] S. D. Paola and G. Fedon. Subverting ajax. In *CCC Conference*, 2006.

[30] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *RAID*, 2004.

[31] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[32] Samy. I'm popular. Description of the MySpace worm by the author, including a technical explanation., Oct 2005.

[33] P. Saxena, R. Sekar, and V. Puranik. Efficient fine-grained binary instrumentationwith applications to taint-tracking. In *Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, 2008.

[34] A. Sotirov. Blackbox reversing of XSS filters. *RECON*, 2008.

[35] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. 2006.

[36] Symantec Corp. Symantec internet security threat report. Technical report, Symantec Corp., April 2008.

[37] Unicode, Inc. Unicode character database. `http://unicode.org/Public/UNIDATA/PropList.txt`, 2008.

[38] W. Venema. Taint support for PHP. ftp://ftp.porcupine.org/pub/php/php-5.2.3-taint-20071103.README.html, 2007.

[39] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2007.

[40] W3C. HTML 5 specification. `http://www.w3.org/TR/html5/`.

[41] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in mashupos. In *SOSP*, 2007.

[42] Web Application Security Consortium. Web application security statistics project 2007. `http://www.webappsec.org/projects/statistics/wasc_wass_2007.pdf`.

[43] XSSed.com. Famous XSS exploits. `http://xssed.com/archive/special=1`, 2008.

[44] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. *USENIX Security Symposium*, 2006.

A Unicode Whitespace Points

Table 1 contains the Unicode points used to implement the delimiters as discussed in Section 6.

U+0009	U+000A	U+000B	U+000C	U+000D
U+0020	U+00A0	U+2000	U+2001	U+2002
U+2003	U+2004	U+2005	U+2006	U+2007
U+2008	U+2009	U+200A	U+2028	U+2029

Table 1: Unicode Whitespace Points