Using Web Application Construction Frameworks to Protect Against Code Injection Attacks

Benjamin Livshits and Úlfar Erlingsson
Microsoft Research

Abstract

In recent years, the security landscape has changed, with Web applications vulnerabilities becoming more prominent that vulnerabilities stemming from the lack of type safety, such as buffer overruns. Many reports point to *code injection attacks* such as cross-site scripting and RSS injection as being the most common attacks against Web applications to date. With Web 2.0 existing security problems are further exacerbated by the advent of Ajax technology that allows one to create and compose HTML content from different sources within the browser at runtime, as exemplified by customizable mashup pages like My Yahoo! or Live.com.

This paper proposes a simple to support, yet a powerful scheme for eliminating a wide range of script injection vulnerabilities in applications built on top of popular Ajax development frameworks such as the Dojo Toolkit, `prototype.js`, and AJAX.NET. Unlike other client-side runtime enforcement proposals, the approach we are advocating requires only minor browser modifications. This is because our proposal can be viewed as a natural finer-grained extension of the same-origin policy for JavaScript already supported by the majority of mainstream browsers, in which we treat individual user interface widgets as belonging to separate domains.

Fortunately, in many cases no changes to the development process need to take place: for applications that are built on top of frameworks described above, a slight framework modification will result in appropriate changes in the generated HTML, completely obviating the need for manual code annotation. In this paper we demonstrate how these changes can prevent cross-site scripting and RSS injection attacks using the Dojo Toolkit, a popular Ajax library, as an example.

PLAS07 June 14, 2007, San Diego, California, USA.

Categories and Subject Descriptors D.2.4 [*Software engineering*]: Software/Program Verification

General Terms Security, Languages, Verification

Keywords software security, software construction frameworks, same-origin policy, code injection attacks

1. Introduction

In recent years, the battle for software security has largely moved into the area of Web applications, with vulnerabilities such as SQL injection and cross-site scripting dominating mailing lists and bulletin boards, the space once populated by buffer overruns and format string attacks. Web applications present an attractive attack target because of their wide attack surface and the potential to gain access to sensitive credentials such as passwords and credit card numbers. To make matters worse, unlike server-side software, Web applications are often developed by programmers with less security sophistication.

While security experts routinely bemoan the current state of the art in software security, from the standpoint of the application developer, application security requirements present yet another hurdle to overcome. Given the management pressure for extra functionality, "lesser" concerns such as performance and security often do not get the time they deserve. While it is common to blame this on developer education, a big part of the problem is that it is extremely easy to write unsecure code.

By was of illustration, consider an application that prompts the user for her name and sends a greeting back to the browser. The following example illustrates how one can accomplish this task in a Java/J2EE application:

```
ServletResponseStream out = resp.getOutputStream();
out.println("<p>Hello, " + username + ".</p>");
```

However, the apparent simplicity of this example is deceptive: assuming `username` is supplied as application input, this piece of code is vulnerable to a cross-site scripting attack [3]. This is because executable JavaScript can be embedded into `username`. When the request is processed within the Web application, this JavaScript will be passed to the client's browser for execution, enabling cookie theft.

Most recently, cross-site scripting issues have lead to the development of JavaScript worms such as the Samy worm that took down the MySpace.com site in October 2005 [10, 31].

In summary, the most natural way to achieve the task of printing the user's name is broken: *the default is unsafe*. To make this secure, the developer has to apply input sanitization: she needs to exclude the myriad different ways to pass JavaScript into the application [30], often a tedious and error-prone task. Even after the issue of data sanitization has been dealt with, the developer still needs to consider *all the ways* in which tainted input can propagate through the application to make sure it is sanitized on all paths, a problem that has been shown amendable to static analysis [18, 22, 33]. It is, however, very rare indeed that there is a compelling reason to have previously unseen JavaScript code passed to the browser. We believe that it is time to turn the situation around, ensuring that secure software is the default.

We are rapidly approaching a world in which much of the dynamically generated Web content is produced by *frameworks*, such as AJAX.NET [26], the Dojo Toolkit [5], and numerous other libraries, all of which allow the developer to lay out richly functional GUI controls the way they would regular HTML. Increasingly, JavaScript and HTML is being generated from other languages, as exemplified by the Google Web Toolkit [9] and OpenLaszlo [21].

Such frameworks and code generation tools provide ample opportunities to produce code that uses *safe defaults*. These defaults would be enforced on the client side for the majority of commonly used GUI controls or building blocks as a result of using the framework without requiring the developer to provide explicit annotations. As a result of this framework-based approach, the developer would have to go out of her way to make the resulting code insecure.

1.1 Problem Scope

In this paper we demonstrate our vision for more secure software by construction as applied to *code injection* attacks. These attacks, the most notable of which is cross-site scripting, account for the lion's share of all security vulnerabilities reported in Web applications [3]. In addition to cross-site scripting we also focus on or feed injection [1], another JavaScript code injection issue commonly found in aggregation mashup pages such as My Yahoo [25], or feed readers such as Sage [19]. Feed injection attacks take advantage of downloaded feed contents to embed malicious executable JavaScript. Just as with sophisticated Ajax applications, feed data is represented in XML, only being assembled into its final HTML form on the client, thus making on-the-wire rewriting an ineffective defense strategy [29, 34].

In this paper we propose a scheme in which every user interface widget, such as a drop-down list, tree, rich-text content pane, etc. acts as a *principal*. The code associated with the widget would correspond to the principal so that it can only access DOM elements *within the widget itself*. This scheme represents a natural extension of the same-origin code policy already present in JavaScript. Just like with the same-origin policy, the browser needs only to walk the DOM tree to find out the origin of a document containing a piece of embedded script. However, in this case, the DOM traversal within the browser would also record principals attached to DOM elements.

Today's Ajax applications are often constructed on top of toolkits or programming frameworks that provide a set of user interface widgets. We propose that frameworks be modified to support principal generation. This way, every application developed on top of an appropriately modified framework will be able to take advantage of better security caused by these framework modifications by default. Note that these principals are inserted wherever the HTML page is generated: for AJAX.NET applications, that may be the server, for more dynamic frameworks such as the Dojo Toolkit, this composition may occur entirely on the client side. With this approach, we get the desired sandboxing properties [13] with the burden of HTML annotation shifted to the framework instead of the developer.

This isolation-based approach is also more flexible than script whitelisting techniques recently proposed in the BEEP framework [17], as it does not require that all executing script be known in advance and therefore nicely supports dynamic script loading. Although both our isolation policies and our proposed enforcement mechanism are less general compared to other enforcement schemes [6, 34], they are also easier to implement given the current state of mainstream browsers. We explore more sophisticated security policies and enforcement tactics required by compound user interface widgets in Section 4.

1.2 Contributions

This paper makes the following contributions:

- We propose a natural refinement of the same-origin policy already supported by most mainstream browsers that extends the notion of same origin to the level of individual user interface elements to provide finer-grain isolation within the page.
- We describe how this approach solves a variety of code injection problems including cross-site scripting and RSS injection without requiring drastic changes to the existing browser infrastructure.
- We outline how appropriate annotations can be automatically generated by the framework and embedded into produced HTML. The majority of developers will be able to take advantage of framework modifications without changing their code at all. We provide a case study of augmenting the Dojo Toolkit to output additional principal information and describe how the resulting augmented widgets can be used to protect against various injection attacks in the context of mashup pages.

1.3 Paper Organization

The rest of the paper is organized as follows. Section 2 summarizes the technique we propose, describing how principals interact with runtime enforcement within the browser. Section 3 describes a case study that shows how our techniques are effective at stopping cross-site scripting and RSS injection attacks. Section 4 describes more complex security enforcement schemes. Finally, Section 5 summarizes related work and Section 6 concludes.

2. Overview

The same-origin policy of JavaScript limits access to DOM and other browser resources such as cookies by carefully keeping track of the origin of a particular piece of code. The principal in the default same-origin scheme is a triple $(host, protocol, port)$ and the origin of a piece of JavaScript is determined by where it is embedded in the HTML DOM. In particular, the origins of of both a DOM element and a piece of script are determined by walking to the top of the DOM tree. JavaScript access to DOM is only allowed if the origin, as encoded by the triple above, is the same.

In this paper we propose that application-specific principals that can be attached to an arbitrary DOM element be allowed *in addition* to the default same-origin scheme. The existing enforcement mechanism will need to be augmented: as the browser walks up the DOM tree, the *principal list* will be collected. For instance, both today's and yesterday's entries in the example in Figure 1 will have the principal list `blog-body`; `blog-entry`. As such, JavaScript code embedded within either of these entries will be able to access the other entry. Of course, generating unique principals for individual entries such as `blog-entry-1`, `blog-entry-2`, ... will provide stronger isolation if it is desired[1]. Note that the order of principals is important: code annotated with principal list `blog-body`; `blog-entry` will not be able to access DOM with principal list `blog-entry`; `blog-body`. It is possible to use a set of principals, which would make the order unimportant, however, in our design we chose to be more conservative.

```
<div principal='blog-body'>
    <b>Blog entries</b>
    <div principal='blog-entry'>
      today's entry
     </div>
     <div principal='blog-entry'>
        yesterday's entry
    </div>
</div>
```

Figure 1: Example of a blog with principal annotations.

[1] Our examples in Section 3 follow the principle of *maximum isolation* by default. However, if desired, sharing can be achieved if the Web application developer annotates her application by hand.

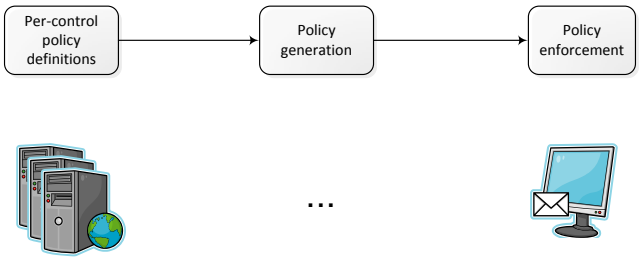

Figure 2: System architecture.

A crucial feature of our approach is that principal annotations need not be written by hand by the Web application developer: in most cases, it is enough to have the application framework generate default isolation policies, which will be extended to all applications written on top of the framework. As shown in Figure 2, individual frameworks define a system of principals associated with GUI widgets that they provide. For instance, AJAX.NET provides a `ajaxToolkit` : `TabContainer` element, which is used for creating tabbed user interfaces. AJAX.NET may by default generate unique principals for each tab, ensuring that neither can access each other. When the final HTML is produced, either on the server or the client side, these principals are injected in it. The isolation is accomplished through the changes to the same-origin policy described above.

As further illustrated through examples in Section 4, principals can also be manipulated programmatically, through changing the attributes of the surrounding DOM element, by calls to `getAttribute` and `setAttribute`. It is our goal that an explicitly defined principal be functionally equivalent to one that was assigned programmatically. Notice that only JavaScript that has the permission to access a DOM element will be able to call `setAttribute` on it, which is a necessary condition for changing principals. This helps prevent malicious JavaScript from re-assigning principals. Programmatic access to principals is useful for principal delegation: a piece of JavaScript code that has access to a DOM element may enable another piece of JavaScript code with the same principal to access the DOM element, as explained in Section 4.1. Similarly to DOM elements, since functions are objects in JavaScript, principals on a piece of JavaScript code, such as an anonymous function, can be easily changed as well by assigning to the field `principal`. For instance,

```
button.onclick = function(){...};
```

can be rewritten as

```
button.onclick = function(){...};
button.onclick.principal = 'blog-entry';
```

Finally, JavaScript code that is generated dynamically, for example, though a call to `eval` receives the same principal as the code containing the `eval`.

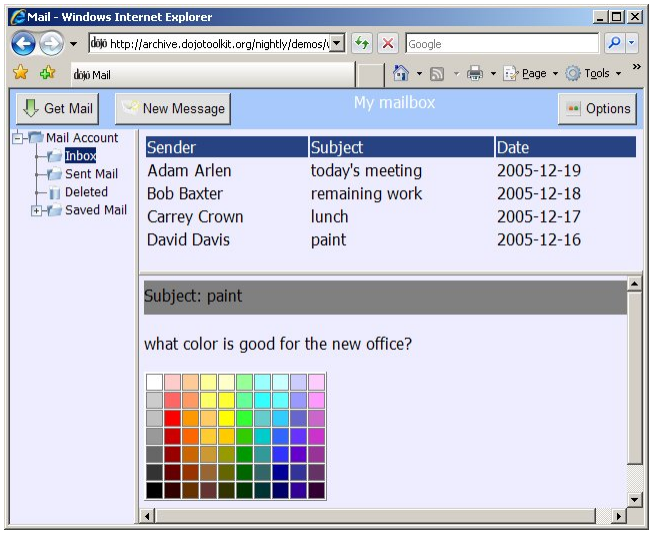

Figure 3: Mail reader application constructed using Dojo Toolkit.

3. Case Studies

This section describes in detail how our approach addresses cross-site scripting and RSS injection vulnerabilities. To make our discussion concrete, we will use the Dojo Toolkit, a popular suite of libraries that simplify the development of Ajax applications [5]. Dojo provides a range of user interface widgets that simplify the task of GUI construction. Similar arguments could be made for the majority of other such libraries. This section describes how to augment Dojo Toolkit widgets to support simple sandboxing described above. Section 3.1 describes cross-site scripting prevention, while Section 3.2 deals with RSS injection.

3.1 Protecting Against Cross-Site Scripting

The Dojo Toolkit makes it easy to construct a rich-text email client such as the one shown in Figure 3 by laying out several interface components of predefined Dojo types. The mail pane declaration shown in Figure 4 allows the pane contents to be loaded from `Mail/MailAccount.html`, an HTML file that can is changed at runtime, depending on the message being selected. One way to proceed is by completely disallowing all executable JavaScript within the message pane. This functionality could also be achieved with a `<noexecute>` block in BEEP [16, 17].

However, this approach is unnecessarily restrictive. Indeed, we primarily care about JavaScript execution affecting other portions of the page, so we may allow some JavaScript

```
<div id="contentPane" dojoType="ContentPane"
  sizeMin="20" sizeShare="80" principal="contentPane"
  href="Mail/MailAccount.html" style="padding: 5px">
</div>
```

Figure 4: Augmented mail pane HTML code generated by Dojo.

Principal/Resource id or type	contentPane1	contentPane2	XmlHttpRequest	Cookies
Code in pane with id `contentPane1`	✓	✗	✗	✗
Code in pane with id `contentPane2`	✗	✓	✗	✗

Figure 5: Default access control matrix for Dojo content panes.

within the message to run as long as it does not affect areas of the page with a different principal. In fact, the mail client shown in the figure does support displaying rich HTML content and JavaScript `onmouseover` handlers run when the user mouses over a color swatch in the message pane; this may be useful for showing the color name or RGB value in a tooltip. This technique disallows other portions of the page from snooping on sensitive email content. However, since messages act as the same principal, it may be possible for malicious JavaScript code from one email message to affect the contents of another message.

This isolation technique alone will also go a long way toward preventing cross-site scripting and JavaScript worms, such as Yamanner [4] that propagated through Yahoo! Mail each time a user opened a cleverly crafted email message. Browser history, cookies, the `XmlHttpRequest` object, etc. have no principals associated with them. Since worm functionality requires Ajax RPCs to propagate, making the `XmlHttpRequest` object inaccessible to the content pane principal by default, as shown in the access control matrix in Figure 5 will disallow worm propagation. Moreover, sensitive cookie data can be be made inaccessible with the same mechanism, without requiring specialized browser extensions such as Noxes [20]. All of these important benefits can be achieved by just associating a principal with the message pane as shown in Figure 4.

3.2 Protecting Against RSS Injection

Similar default policies can be produced for other widgets. Consider the `Tree` widget in the Dojo toolkit that allows one to create multi-level trees in the browser. `<div class = "dojoTree">`. Node labels are explicitly listed within the tree. Labels have event-processing code attached to them to support mouse-click events, etc. Moreover, label text supports HTML and may have some embedded JavaScript.

Consider a tree widget that is used for displaying news items in an RSS feed. A safe default for the tree widget is to assert that code within the tree declaration cannot affect anything outside of the declaration as shown in Figure 6. As

```
<div class="dojoTree" style="-moz-user-select: none;"
    id="myTree" principal="myTree">
  <div class="dojoTreeNode">
    <span class="dojoTreeNodeLabel"
        dojoattachpoint="labelNode"
        treenode="2.2"
        dojodragsource="dojoDragSourceIdx_21">
      <span class="dojoTreeNodeLabelTitle"
          dojoattachevent="onClick: onTitleClick"
          dojoattachpoint="titleNode">
        <b>HTML label</b>
      </span>
    </span>
    ...
  </div>
</div>
```

Figure 6: Augmented tree declaration in Dojo; the principal is added to the top-level tree.

a result, the tree widget charged with displaying an RSS feed will enforce the following two properties:

1. RSS feed injections cannot affect other portions of the page, which is especially important for mashup pages containing sensitive data such as email.
2. Other portions of the page cannot get at private RSS feed contents, which is especially important for private RSS feeds carrying sensitive data [8].

However, note that this default would not protect RSS messages from modifications by other malicious messages *within* the same feed. This is because the HTML node that is annotated with a principal is the surrounding tree widget.

If finer-grained protection is desired, we can have individual tree nodes be declared with their own principals. One advantage of only maintaining principals at the tree level and not persisting them at the level of individual tree nodes is that tree manipulation functions need not be changed to support principals as well. Tree nodes support addition and removal, however, neither method needs to be augmented since principal information is stored at the level of the surrounding tree and not persisted at nodes. Similarly, there is no problem with drag-and-drop: when the tree node move operation

```
 1  var DemoTreeManager = {
 2    djWdgt: null, myTreeWidget: null, ctxMenu = null,
 3
 4    addTreeContextMenu: function(){
 5      ctxMenu = this.djWdgt.createWidget("TreeContextMenu",{});
 6      ctxMenu.addChild(this.djWdgt.createWidget(
 7        "TreeMenuItem",{caption:"Add Menu Item", widgetId:"ctxAdd"}));
 8      document.body.appendChild(ctxMenu.domNode);
 9
10      /* Bind the context menu to the tree */
11      ctxMenu.listenTree(this.myTreeWidget);
12    },
13
14    addController: function(){
15      this.djWdgt.createWidget("TreeBasicController",
16        {widgetId:"myTreeController", DNDController:"create"}
17      );
18    },
19
20    bindEvents: function(){
21      dojo.event.topic.subscribe("ctxAdd/engage",
22        function (menuItem) { addNode(menuItem.getTreeNode(), "myTreeController"); }
23      );
24    },
25
26    addNode: function(parent,controllerId){
27      this.controller = dojo.widget.manager.getWidgetById(controllerId);
28      var res = this.controller.createChild(parent, 0, { title: "New node" });
29    },
30
31    init: function(){
32      /* Initialize this object */
33      this.djWdgt = dojo.widget;
34      this.myTreeWidget = this.djWdgt.manager.getWidgetById("myTreeWidget");
35      this.addTreeContextMenu(); this.addController(); this.bindEvents();
36    }
37  };
```

Figure 7: Context menu creation code.

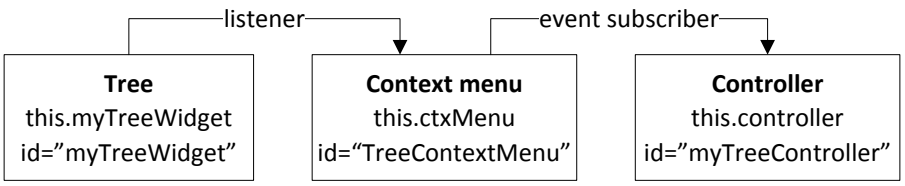

Figure 8: Tree widget-menu-controller architecture in Dojo.

is complete, the principal will be available through a DOM traversal of the updated tree.

Note that because of code reuse, the same tree widget may be used within the same HTML page multiple times. The tree instances will be isolated from each other as long as they are assigned different principals.

4. Beyond Simple Sandboxing

So far, our discussion has only concerned relatively simple self-contained widgets, for which isolation policies sufficed. However, it is not uncommon to have two or more elements that interact using a common development pattern. As an example, consider the relationship between Dojo trees described above and associated context menus, accessible with a right-click.

Since the tree and the menu are represented by different Dojo widget types, they would therefore correspond to distinct principals. However, it is quite natural to allow the associated context menu to have access to the underlying tree. In this section, we use this example to motivate extensions to both modifications of Dojo and also more sophisticated runtime enforcement scheme.

Context menu creation code adopted from a Dojo tutorial [32], which we shall use throughout this section is shown in Figure 7. In addition to the tree and the associated context menu, a *tree controller*, which is also a Dojo widget, is used to perform actions on the tree. The relationship between these three widgets is shown in Figure 8. In this case, the context menu items manipulate nodes of the the underlying tree. The goals we want to accomplish are twofold:

1. We want to to make sure that the controller can properly access underlying tree nodes (in other words, the call to `createChild` on line 28 should succeed).
2. We want to to make sure that menu's actions are not allowed to manipulate DOM elements outside the tree.

4.1 Simple Principal Delegation

In order to give the controller the permission to manipulate tree nodes, we use *principal delegation*:

1. The principal of the tree is first delegated to the context menu (as part of the call to `listenTree` on line 11).
2. Next, this principal is delegated to the controller via a call to `subscribe` on line 21.

The first delegation requires augmenting the code of `listenTree` on line 12 to explicitly associate the principal of the tree with the context menu, as shown in Figure 9. Because principals are represented as HTML attributes, they can also be manipulated programmatically in JavaScript by calls to `getAttribute` and `setAttribute`.

4.2 Attaching Principals to Code

The second delegation on line 21 is a little more tricky, though. We need to parse the first argument to `subscribe` to extract the widget whose principal needs to be delegated. The second argument of the call to `subscribe` is an anonymous function which would gain access to DOM elements that have the principal associated with the "Add" menu item (widget `ctxAdd`).

Unfortunately, the relationship between the controller and the the anonymous function in the second case is much more difficult to encode. This is why we have to resort to assigning principals to individual pieces of code.

Previously, in our fine-grained modification of the same-origin policy we were able to determine the principal of a code snippet by looking up the element it is attached to in the DOM. Fortunately, principal delegation can be performed at the level of JavaScript, because functions in JavaScript are objects that can be assigned new properties. Figure 10 shows a modified version of the `subscribe`. Added code that performs principal attachment is shown on lines 4–9. Alternatively, we could explicitly assert the principal with a call such as `enablePrivilege`, as it is done in the JVM.

In its most general form, this type of delegation will require mechanisms similar to Java stack inspection. It has been shown in the past that such access control schemes can be implemented using inline reference monitors [7], which can be supported using recently proposed browser-side security extensions [6]. However, there is much to gain from the simpler mechanisms we describe above, even if they may only allow for coarser-grained protection.

5. Related Work

There has been a great deal of interest in static and runtime protection techniques to improve the security posture of traditional "Web 1.0" applications. Static analysis allows the developer to avoid issues such as cross-site scripting before the application goes into production. Runtime analysis allows exploit prevention and recovery.

```
1   listenTree : function (tree) {
2       var nodes = tree.getDescendants();
3       for (var i = 0; i < nodes.length; i++) {
4           if (!nodes[i].isTreeNode) {
5               continue;
6           }
7           this.bindDomNode(nodes[i].labelNode);
8       }
9       ...
10      this.listenedTrees.push(tree);
11      /* Perform principal delegation */
12      this.setAttribute('principal', tree.getAttribute('principal'));
13  }
```

Figure 9: Augmented method `listenTree`.

The WebSSARI project pioneered this line of research. WebSSARI uses combined unsound static and dynamic analysis in the context of analyzing PHP programs [14]. WebSSARI has successfully been applied to find many SQL injection and cross-site scripting vulnerabilities in PHP code. Several projects that came after WebSSARI improve on the quality of static analysis for PHP [18, 33]. The Griffin project proposes a scalable and precise sound static and runtime analysis techniques for finding security vulnerabilities in large Java applications [22, 24]. Based on a vulnerability description, both a static checker and a runtime instrumentation is generated. Static analysis is also used to drastically the runtime overhead in most cases. The runtime system allows vulnerability recovery by applying user-provided sanitizers on execution paths that lack them. Several other runtime systems for taint tracking have been proposed as well, including Haldar et al. for Java [11] and Pietraszek et al. [28] and Nguyen-Tuong et al. for PHP [27].

While server-side enforcement mechanisms are applicable for traditional Web applications that are composed entirely on the server side [18, 22, 33], Web 2.0 applications that make use of Ajax often fetch both data and JavaScript code from many sources, with the entire final HTML only available within the browser, making runtime client-side enforcement a natural choice. Recently, there has been a number of proposals for runtime enforcement mechanisms to ensure that security properties of interest hold for rich-client applications executing within the browser [6, 13, 16, 17, 34]. This effectively gives the developers isolation mechanisms similar to processes in operating systems.

A common example of one such enforcement strategy is sandboxing applied to portions of a mashup page [13]. While a step in the right direction, unfortunately, such proposals often rely on the developer to carefully annotate their HTML for the browser to perform proper enforcement. Subspace [15] focuses on mechanisms for communication between data and code from different domains, whereas the focus of our work is primarily on isolation.

The BEEP project proposes server-generated whitelisting policies as well as client-side support to prevent cross-site scripting attacks [17]. For all known pieces of JavaScript, their hash values are computed and passed to the browser. For every piece of JavaScript code it is about to execute, the browser first makes sure that its hash value is in the whitelist. While a powerful and a simple approach against code injection attacks, the adoption of BEEP poses some challenges. The server-side application must be suitably examined or modified to identify all places where script is generated. This is especially challenging if there are either many small pieces of script embedded into HTML or if script is generated at runtime and not known in advance. A single missed piece of JavaScript will lead to false positives resulting in undesirable end-user behavior, so relegating the task of producing annotations to the development framework is a natural choice.

Erlingsson et al. make an end-to-end argument for the client-side enforcement of security policies that apply to client behavior [6]. Their proposed mechanisms use server-specified, programmatic security policies that allow for flexible client-side enforcement, even to the point of runtime data tainting. In contrast, the techniques in this paper are a simpler modification of the same-origin policy already supported by the majority of browsers, and may be simpler to implement and adopt.

6. Conclusions

Recently we have seen a strong trend towards providing rich APIs for Ajax application development. Just as in the context of traditional desktop applications, this shift towards rich frameworks and APIs exposes ample opportunities for both runtime enforcement and bug finding [2, 12, 23]. Combined with runtime enforcement in the context of the browser [6, 16, 17, 34], the use of frameworks can automatically result in significantly more secure applications without additional code annotation burden placed on the developer.

In this paper we explored how the use of toolkits allows for secure by construction Web applications. We have described how simple isolation of user interface widgets goes a long way towards rendering cross-site scripting and RSS

```
1   this.subscribe = function (listenerObject, listenerMethod) {
2       var tf = listenerMethod || listenerObject;
3       var to = (!listenerMethod) ? dj_global : listenerObject;
4       if(typeof(listenerObject) === 'string' && listenerObject.contains('/')){
5           /* Perform principal delegation */
6           var widgetId = listenerObject.substr(0, listenerObject.indexOf('/'));
7           var widget = dojo.widget.manager.getWidgetById(widgetId);
8           tf.principal = widget.getAttribute('principal');
9       }
10
11      return dojo.event.kwConnect({srcObj:this, srcFunc:"sendMessage",
12                                                  adviceObj:to, adviceFunc:tf});
13  };
```

Figure 10: Augmented `subscribe` function in Dojo.

injection attacks ineffective, all with a small modification to the browser's same-origin policy.

References

[1] Robert Auger. Feed injection in Web 2.0. `www.spidynamics.com/assets/documents/HackingFeeds.pdf`, 2006.

[2] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *Proceedings of the European Systems Conference*, 2006.

[3] CGI Security. The cross-site scripting FAQ. `http://www.cgisecurity.net/articles/xss-faq.shtml`.

[4] Eric Chien. Malicious Yahooligans. `http://www.symantec.com/avcenter/reference/malicious.yahooligans.pdf`, August 2006.

[5] Dojo Foundation. Dojo, the JavaScript toolkit. `http://dojotoolkit.org`, 2007.

[6] Úlfar Erlingsson, Benjamin Livshits, and Yinglian Xie. End-to-end Web application security. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 2007.

[7] Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, 2000.

[8] Steven Garrity. Private RSS feeds: Support for security in aggregators. `http://labs.silverorange.com/archives/2003/july/privaterss`, July 2003.

[9] Google Web toolkit. `http://code.google.com/webtoolkit`.

[10] Jeremiah Grossman. Cross-site scripting worms and viruses: the impending threat and the best defense. `http://www.whitehatsec.com/downloads/WHXSSThreats.pdf`, April 2006.

[11] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 303–311, December 2005.

[12] Seth Hallem, Ben Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 69–82, June 2002.

[13] Jon Howell, Collin Jackson, Helen J. Wang, and Xiaofeng Fan. MashupOS: Operating system abstractions for client mashups. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 2007.

[14] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing Web application code by static analysis and runtime protection. In *Proceedings of the Conference on World Wide Web*, pages 40–52, May 2004.

[15] Collin Jackson and Helen J. Wang. Subspace: Secure cross-domain communication for Web mashups. In *Proceedings of the World Wide Web Conference*, May 2007.

[16] Trevor Jim, Nikhil Swamy, and Michael Hicks. BEEP: Browser-enforced embedded policies. Technical report, Department of Computer Science, University of Maryland, 2006.

[17] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the International World Wide Web Conference*, 2007.

[18] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: a static analysis tool for detecting Web application vulnerabilities (short paper). In *Proceedings of the Symposium on Security and Privacy*, May 2006.

[19] David Kierznowski. Cross context scripting with sage. `http://michaeldaw.org/md-hacks/rss-injection-in-sage-part-2`, September 2006.

[20] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the Symposium on Applied Computing*, April 2006.

[21] Laszlo Systems, Inc. OpenLaszlo: the premier open-source platform for rich Internet applications. `http://www.openlaszlo.org`, 2007.

[22] Benjamin Livshits and Monica S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings*

of the Usenix Security Symposium, pages 271–286, August 2005.

[23] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security vulnerabilities using PQL: a program query language. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2005.

[24] Michael Martin, Benjamin Livshits, and Monica S. Lam. SecuriFly: Runtime vulnerability protection for Web applications. Technical report, Stanford University, October 2006.

[25] Jeremy Moeder. Yahoo RSS XSS vulnerability. `http://www.securityfocus.com/archive/1/413594`, October 2005.

[26] Laurence Moroney. *Foundations of Atlas: Rapid Ajax Development with ASP.NET 2.0*. Apress, 2006.

[27] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening Web applications using precise tainting. In *Proceedings of the IFIP International Information Security Conference*, June 2005.

[28] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the Recent Advances in Intrusion Detection*, September 2005.

[29] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proceedings of Operating Systems Design and Implementation*, 2006.

[30] RSnake. XSS cheat sheet for filter evasion. `http://ha.ckers.org/xss.html`.

[31] Samy. The Samy worm. `http://namb.la/popular`, October 2005.

[32] willCode4Beer. Introducing the Dojo tree widget. `http://willcode4beer.com/ware.jsp?set=dojoTreeWidget`, January 2007.

[33] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the Usenix Security Symposium*, pages 271–286, August 2006.

[34] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript instrumentation for browser security. In *Proceedings of the Conference on the Principle of Programming Languages*, January 2007.