Defeating Script Injection Attacks with Browser-Enforced Embedded Policies[*]

Trevor Jim
AT&T Labs Research

Nikhil Swamy and Michael Hicks
University of Maryland, College Park

November 2, 2006

Abstract

Web sites that accept and display content such as wiki articles or comments typically filter the content to prevent injected script code from running in browsers that view the site. The diversity of browser rendering algorithms and the desire to allow rich content makes filtering quite difficult, however, and sophisticated attacks (like the Samy Myspace worm) have exploited filtering weaknesses. To solve this problem, this paper proposes a mechanism called Browser-Enforced Embedded Policies (BEEP). The idea is that a web site can embed a policy inside its pages that specifies which scripts are allowed to run. The browser, which knows exactly when it will run a script, can enforce this policy perfectly. We have added BEEP support to several browsers, and built tools to simplify adding policies to web applications. We found that supporting BEEP in browsers requires only small and localized modifications, modifying web applications requires minimal effort, and enforcing policies is generally lightweight.

1 Introduction

Many web sites republish content supplied by their user communities, or by third parties such as advertising networks and search engines. If this republished content contains scripts, then visitors to the site can be exposed to attacks such as cross-site scripting (XSS) [1], and can themselves become participants in attacks on the web site and on others [9]. The standard defense is for the web site to filter or transform any content that does not originate from the site itself, to remove scripts and other potentially harmful elements [13, 19, 11].

Filtering is complicated in practice. Sites want to allow their users to provide rich content, with images, hyperlinks, typographic stylings and so on; scripts can be embedded in rich content in many ways, and it is nontrivial to disable the scripts without also disabling the rich content. One reason is that different browsers parse and render content differently: filtering that is effective for one browser can be ineffective for another. Moreover, browsers try to be forgiving, and can parse and render wildly malformed content, in unexpected ways. All of these complications have come into play in real attacks that have evaded server-side filtering (e.g., the "Samy" worm [18]).

To avoid these difficulties, we propose to apply filtering and other defense mechanisms in the browser. The principal advantage is that correct script detection is already performed by the browser, as a necessary precondition of page rendering. This makes the browser the ideal place

[*]University of Maryland, Department of Computer Science Technical Report, CS-TR-4835

to filter scripts. Indeed, for some web applications (e.g., GPokr, S3AjaxWiki), most or all of the application logic is executed in the browser, with the web site acting only as a data store. For these applications, browser-side filtering may be the only option.

If the browser is the ideal place to detect and filter scripts, it is the web site that is best able to determine which scripts are needed for the web application to function properly. We believe the ideal filtering approach, therefore, is to allow the web site to specify which scripts are approved for execution, while the browser filters any illegal scripts, allowing approved scripts and other necessary elements to run unimpeded. In short, the web site sets the policy and the browser enforces it.

In this paper, we describe one possible implementation of this idea, which we call Browser-Enforced Embedded Policies (BEEP). The security policy is expressed as a trusted JavaScript function that the web site embeds in the pages it serves. We call this function the *security hook*. A suitably-modified browser passes each script it detects to the security hook during parsing (along with other relevant information) and will only execute the script if the hook deems it legitimate.

With this technique we have implemented two kinds of embedded policies. The first is a *whitelist* policy in which the hook function includes a one-way hash of each legitimate script appearing in the page. When a script is detected in the browser and passed to the hook function, the hook function hashes the script and matches it against the whitelist; any script whose hash does not appear in the list is not executed.

The second kind of policy is a *DOM sandbox*. Here, the web application structures its pages to identify content that might include malicious scripts. The possibly-malicious user content is placed inside of a `<div>` or `<span>` element that acts as a sandbox:

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

Within the sandbox, rich content (typographic styling, etc.) is enabled, but all scripts are disabled. When invoked, the hook function will examine the document in its parsed representation, a Document Object Model (DOM) tree. Beginning at the DOM node of the script, the hook function inspects all of the nodes up to the root of the tree, looking for "noexecute" nodes. If such a node is found, the script is not executed.[1]

BEEP has several benefits. First, it requires minimal changes to existing infrastructure. Browsers must be modified to support security hooks, and web application pages must be modified to include embedded policies, but both sorts of changes are simple and localized. We have successfully modified the Konqueror and Safari browsers to support security hooks, and we have implemented partial support in the closed-source Opera browser. We have also modified some existing web applications to embed policies, and have built some simple tools to help in this process. Second, BEEP is entirely backwards compatible. Browsers that do not support hooks will still render pages that define hooks, albeit without the protection they offer. Servers can (and should) continue to filter user content, with BEEP serving as a second line of defense against scripts that escape detection. Finally, BEEP supports incremental deployment—while we intend that ultimately web sites be responsible for embedding appropriate policies, policies could also be embedded by other means. For example, a third party could generate a whitelist for an infected site, and a firewall or other proxy could insert the whitelist policy into pages served from that site.

In the next section, we review script injection attacks and standard defenses in more detail. The remainder of the paper explains our BEEP technique and policies, shows that they are effective at defeating attacks while imposing minimal overhead, and compares BEEP to related work.

[1]We must take care to prevent cleverly formatted content from escaping its confines as discussed in Section 3.4.

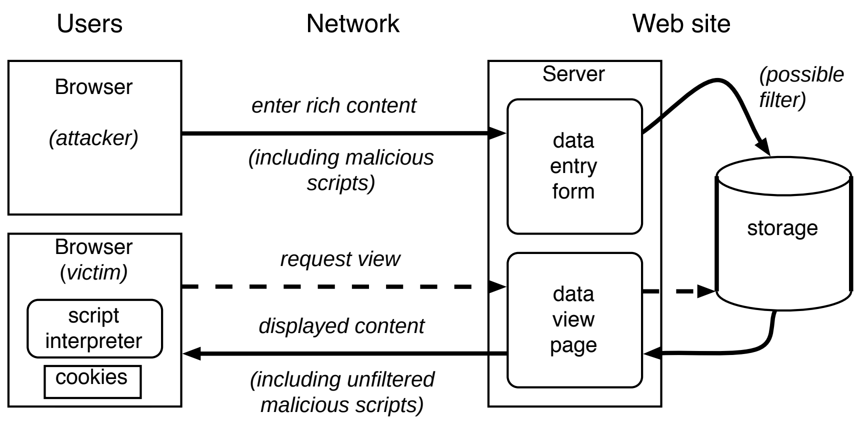

Figure 1: Script injection attack on a typical Wiki/Blog-based site, like MySpace.

2 Background

2.1 Script Injection

We are concerned with attacks that cause a malicious script, typically written in JavaScript, to be injected into the content of a trusted web site. When a visitor views a page on the site, the injected script is loaded and executed in the visitor's browser with the trusted site's privileges. The injected script can leak privileged information (cookies, browsing history, and, potentially, any private content from the site) [1]. The script can also use the visitor's browser to carry out denial of service attacks or other attacks on the web site, or on others. If the web site is very popular, the attack can be greatly amplified [9].

Script injection can be achieved in many ways. In cross-site scripting (XSS), the attacker often exploits cases where a web application inserts user-provided text into a dynamically-generated page, without first filtering the text. For example, users in on-line communities like MySpace, Blogger, and Flickr may enter their own content and add comments to the content of others. This content is stored on the site and may be viewed by anyone. Because different users' profiles are hosted by the same site, if a malicious user were able to include a script in his content, any viewers of that content would run the script with the privileges of the site. This would allow the malicious user's script to steal or modify the viewer's content, including private information stored at the site or at the browser (e.g., as a cookie). Such an attack is shown in Figure 1.

Another way of injecting a script is by "reflection." For example, when asked for a non-existent page, many sites try to produce a helpful page in response, with a "not found" message that includes the URL of the non-existent page that was requested. Therefore, if the site is not careful, an occurrence of the text `<script>...</script>` *in the URL* can be executed on the visitors's browser when it renders the "not found" page. To exploit this, an attacker can try to entice victims to follow URLs with targets that include scripts, e.g.,

```
http://trusted.site/<script>document.location='http://evil.site/?'+document.cookie</script>
```

The attacker could place the URL in a spam e-mail, in a blog comment or wiki entry on `trusted.site`, or even on another site. If a victim follows the link, the script will run in the "not found" page served by `trusted.site`, retrieve the user's `trusted.site` cookie, and send it to `evil.site`.

Ajax [3] is an increasingly popular trend in web applications, and it provides additional opportunities for script injection. An Ajax web application uses JavaScript in the browser to modify the web page seen by the user, potentially without any interaction with the server. Ajax enables rich application behavior while reducing the overhead of communication with the site. However, it also enables script injection such that *the malicious script is never seen by the site*. For example [8], a site might be constructed so that a URL of the form

```
http://www.vulnerable.site/welcome.html?name=Joe
```

would result in the same web page, regardless of the value of "name." The page can contain a script that uses features like `innerHTML` and `document.write` so that the content of the page is modified in the browser, and personalized according the value of "name." This opens the possibility that the malicious script can be constructed *entirely* in the browser, as a combination of "name" and other parameter values, as well as the text of the page itself.

Ajax can be taken to an extreme: web applications like S3AjaxWiki [12] have no server-side logic at all. The application logic consists entirely of JavaScript code that executes in the browser, and the server is used solely as a data store. In this case, clearly any measures to combat malicious scripts must be taken in the browser (and S3AjaxWiki currently provides no such measures).

2.2 Script Detection

The standard solution to script injection is for the web site to filter or transform all possibly-malicious content so that scripts are removed or made harmless, as shown in Figure 1. The simplest kind of filter is to escape the special characters in the content to prevent scripts, if any, from executing when viewed. For example, if some content contains `<script>`, the special characters '`<`' and '`>`' would be escaped as HTML entities '`&lt;`' and '`&gt;`'. When combined with technologies like tainting [11] that track potentially-malicious content, this is an excellent defense. Unfortunately, this simple approach can prevent users from creating rich content. It renders `<script>` elements harmless, but also disables features like typographic styling (`<b>`, ...), lists (`<ul>`, `<li>`, ...), etc. from appearing in user content.

Therefore many sites attempt to *detect* scripts within possibly-malicious content, and filter only those portions of the content. Unfortunately, detecting scripts is hard, for several reasons. First, scripts can be embedded in a web page in many ways; Figure 2 shows some examples. Line 2 embeds a script contained in a separate file. Line 3 is an inline script. Line 4 is an event handler that will be attached to the img element on line 8, and which will be invoked when the user moves the mouse over the element. Line 5 is an inline CSS style declaration that says that the background of elements in class ".bar" should be gotten by executing a script. The script is invoked by the browser as it renders line 9. The script is contained in a javascript:URL; such URLs can appear in a document wherever any other URL can appear. Line 7 is an inline event handler that will execute when the body of the document has finished loading. Line 10 is an element that uses an inline CSS style to invoke a script. Line 11 is a refresh directive that indicates that the page should be refreshed by loading a data:URL. The data:URL is the base64 encoding of a javascript:URL, and it is executed on page refresh. Of course, this is only a partial list of how scripts can be embedded in web pages and we are currently in a phase where browsers are actively being developed to enable more scripting.

Script detection is also complicated by the fact that the process of rendering in the browser is ill-defined. Different browsers can render pages in very different ways, so, what one browser sees

```
<html><head>
  <script src="a.js"></script>
  <script> ... </script>
  <script for=foo event=onmouseover> ... </script>
  <style>.bar{background-image:url("javascript:alert('JavaScript')");}</style>
</head>
<body onload="alert('JavaScript')">
  <img id=foo src="image.jpg">
  <a class=bar></a>
  <div style="background-image: url(javascript:alert('JavaScript'))">...</div>
  <meta http-equiv="refresh"
   content="0;url=data:text/html;base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4K">
</body></html>
```

Figure 2: Ways of embedding scripts in web pages.

as a script may not be a script to another browser. Furthermore, browsers make a best effort to render all pages, no matter how ill-formed: better to render something rather than show a blank page or an error message. This can lead to unexpected ways of embedding scripts. For example, some browsers allow newlines to appear in the "`javascript:`" portion of a javascript:URL, so that

```
<img src='java
script:alert(1)'>
```

will result in script execution.

Quotes that delimit content and encodings of special characters add more complications. There are multiple kinds of quoting and escaping (for URLs, HTML, and JavaScript), which must be stripped at multiple stages. There are multiple quote characters, plus cases in which quotes can be omitted. And malformed input that uses quotes and escapes may still be rendered by helpful browsers. All of the following examples can result in script execution in some browsers.

```
<img src=&#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;&#112;&#116;\
 &#58;&#97;&#108;&#101;&#114;&#116;&#40;&#39;&#88;&#83;&#83;&#39;&#41;>
<img src=javascript:alert(&quot;3&quot;)>
<img src=`javascript:alert("Hello 'world'")`>
<img """><script>alert("ack")</script>">
```

These issues give rise to dozens of techniques for hiding scripts from detection, available on public sites, e.g., `ha.ckers.org`[16]. The techniques are effective in practice: for example, the author of the Samy worm has a writeup [18] where he describes how he evaded the script filters on MySpace. The worm was active in October 2005, and caused over a million MySpace users to add "Samy" to their "friends" list, maintained on the site. Portions of the site had to be closed down for several days to repair the infection.

3 Browser-Enforced Embedded Policies

We have argued that it is difficult for the web site to detect and filter malicious scripts. We now present our alternative approach in detail. The idea is for the web site to specify, for each page, a security policy for allowing and disallowing script execution. The policy is embedded in the pages and enforced by the browser during page rendering. We call this approach Browser-Enforced Embedded Policies (BEEP).

3.1 Attacker Assumptions

We assume that the adversary has no special access to served content, and attempts to inject malicious scripts occur as described in the previous section, e.g., by uploading malicious content to a wiki or phishing with creatively-formed URLs. Therefore, we assume that the web site is trusted by site visitors, up to the limits of the same-origin policy [17]: visitors are willing to execute scripts in site content, but they expect that the site will not distribute private information to a malicious third party. We also assume that the attacker cannot modify content that is en route from the web site; depending on the attacker, this may require HTTPS for transport.

3.2 The Security Hook

In our implementation of BEEP, a web site specifies its policy through a *security hook* that will be used to approve scripts before execution in the browser. The hook is communicated to the browser as the definition of a JavaScript function, `afterParseHook`. A specially-modified browser invokes `afterParseHook` whenever it parses a script while rendering pages. (The necessary browser modifications will be described shortly.) If the hook function returns `true` then the script is deemed acceptable and will be executed; otherwise it will be ignored. To be effective, the hook function must be installed before any malicious scripts are parsed and executed. While the HTML standard does not specify the order of parsing and execution, we have verified that in practice the major browsers parse and execute the first `<script>` element in the head first.

When a modified browser parses a script, it invokes the `afterParseHook` function with three arguments: the text of the parsed script; the DOM element of the parsed script; and the event name (if any) for which the script is to be installed as a handler. Thus when rendering the document fragment

```
<body onload="alert('hello')"> ...  </body>
```

the browser invokes `afterParseHook` on the string `"alert('hello')"`, the DOM node of the `<body>` element, and the string `"onload"`. The policy implemented by the hook function can be any boolean function that can be programmed in JavaScript. We have experimented with two kinds of policies: *whitelists* and *DOM sandboxes*.

3.3 Whitelists

Most current web applications embed scripts in their web pages. Of course, the web application developer knows precisely which scripts belong in each page. Therefore, the developer can write a security hook that checks that every script encountered in browser is one of these known scripts. We call this the *whitelist policy*.

We implement a whitelist in JavaScript as an associative array indexed by the SHA-1 hashes of the known scripts. When `afterParseHook` is invoked on a script, it hashes the script and checks whether the hash appears in the array. For example, for a known script `<script>alert(0)</script>`, `whitelist [SHA1("alert(0)")]` should be defined, while for `<script src="aURL"/>`, `whitelist [SHA1("aURL")]` should be defined.

Here is a sample implementation:

```
if (window.JSSecurity) {
  JSSecurity.afterParseHook = function(code, elt, eventName) {
    if (whitelist[SHA1(code)]) return true;
    else return false;
  };
  whitelist = new Object();
  whitelist["478zB3KkS+UnP2xz8x62ugOxvd4="] = 1;
  whitelist["AOOq/aTVjJ7EWQIsGVeKfdg4Gdo="] = 1;
  ... etc. ...
}
```

Since the whitelist is indexed by hashes, which must change every time a script changes, whitelists clearly demand some automated support in the web development process. We have built some simple tools to help with this process, and we describe them in Section 3.6.

3.4 DOM sandboxing

Our second kind of policy, *DOM sandboxing*, takes a blacklist approach: instead of specifying the approved scripts, we specify the scripts to be rejected. The web application is written to produce web pages in which the parts that contain possibly-malicious content are clearly marked, and the security hook prevents scripts in those parts from executing.

As a first attempt, we suggest that a web application place possibly-malicious content within `<div>` or `<span>` elements that are marked as "noexecute," and which act as a sandbox.

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

The web application would then supply a security hook that receives the DOM node of a script as input, and walks the DOM tree from that node towards the root. If a "noexecute" element is found, the hook function will return `false`, preventing execution.

Unfortunately, this implementation of DOM sandboxing is flawed: malicious content of the form

`</div><script>`*malicious script*`</script><div>`

will escape the sandbox. We call this trick *node-splitting*; similar tricks are used to illegally access hidden files in web servers (using `..` in URLs) and to perform SQL injections.

A simple variation solves the problem. The web application arranges for all possibly-malicious content to be encoded as a JavaScript string, and to be inserted as HTML into the document by a script in the browser, using `innerHTML`:

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

Here the "noexecute" node is created separately from its contents, so that there is no possibility of the contents splitting the node. The assignment of the string to the `innerHTML` property of the node causes the browser to parse and render the string as HTML, producing a DOM tree with the node as parent. The rules for quoting special characters in JavaScript strings are simple, so there is no possibility of malicious content escaping from the string.

HTML frames cause an additional complication. A frame in a document introduces a child document. If an attacker injects a script included in a frame, our hook reaches the top of the frame without encountering the sandbox node, and must continue searching in the parent document. The DOM does not provide easy access from the child to its place in the parent, so our hook must do some searching in the parent document to find the frame element. We give a complete implementation in the appendix.

3.5 Browser modifications

Konqueror and Safari We have modified the Konqueror and Safari browsers to support our security hook functions. These browsers are related: Safari's rendering engine (WebKit) was forked from Konqueror's in 2002. Konqueror's engine currently consists of approximately 200,000 lines of C++, while Safari's consists of about 350,000 lines of C++. Our modifications required changing or adding 427 lines of code in Konqueror, and 415 lines of code in Safari.

In both browsers, each frame in an HTML document is handled by a single instance of the HTML parsing and rendering engine which in turn is associated with an instance of the JavaScript interpreter. As might be expected, the required changes were limited to the interface between the HTML and JavaScript engines. This interface is bi-directional—the HTML engine invokes the JavaScript interpreter to execute scripts that it encounters while parsing the document, and a JavaScript function can modify the document tree that is managed by the HTML engine.

To implement `afterParseHook`, we had to take special care to ensure that certain modifications to the document tree that occur due to the execution of JavaScript do not result in invocations of the hook function. For instance, if a JavaScript function (already authorized by the `afterParseHook`) chooses to insert a dynamically-generated script into the document we must ensure that the hook function is not called once again. The majority of changes (in terms of lines of code) in both browsers were due to a small refactoring that was necessary to handle this case.

Opera We have also implemented partial support for our hooks in a closed-source browser, Opera. Opera supports a feature called User JavaScripts intended to allow users to customize the web pages of arbitrary sites. For example, if a web site relies on non-standard behavior of Internet Explorer, an Opera user can write a User JavaScript that is invoked whenever a page from the site is rendered, and which rewrites the page content so that it renders correctly in Opera. The User JavaScript programming interface permits registering JavaScript callback functions to handle events that occur during parsing and rendering. Crucially, User JavaScript is executed before any scripts on the web page, and it can prevent any script on the web page from executing.

We have written a User JavaScript for Opera that does two things. First, it defines a `JSSecurity` object for every web page, into which a web page can register its `afterParseHook` function. Second, it registers a handler function that calls the user's `JSSecurity.afterParseHook` (if it exists) on script execution events. The Opera implementation handles `<script>` elements perfectly. Opera does not invoke callbacks when parsing a script within an event handler, but we can insert a callback just before an event is delivered to a listener. Similarly, we can insert a callback just before a javascript:URL is executed; however, in this case, Opera does not make the DOM node of the URL available, so we cannot implement DOM sandboxing for `javascript:URL`s in Opera. The complete User JavaScript is 79 commented lines of code (included in the appendix) along with 137 lines for the SHA-1 implementation in JavaScript.

Mozilla Firefox and Internet Explorer We have a partial implementation of security hooks in the Firefox browser and hope to have a full implementation shortly. We have not yet investigated Internet Explorer. It is worth mentioning that both browsers have extensions that can function something like the User JavaScript provided by Opera; the Firefox extension is called Greasemonkey and the IE extension is called Trixie. However, these extensions are not sufficient to implement BEEP, because scripts embedded in a page can execute before the extensions are triggered.

3.6 Web application modifications

Adding BEEP security policies to web application pages is fairly straightforward. For the whitelist policy, this can be done with some simple tool support, depending on how the application was written. For the DOM-based policy, the application developer must author the pages according to the required structure.

Whitelist policies For applications written directly in a mixture of HTML and JavaScript, it is straightforward for a tool to identify the scripts on each page, calculate their hashes, and insert the whitelist and security hook into the document's head. A web developer could use such a tool to add policies to his pages prior to deployment—i.e., when the pages do not contain any user content, so all scripts are legal.

We have written such a tool based on the Tidy HTML parser [20]. Currently, the tool searches for scripts where they most frequently occur: in `<script>` elements, in event handlers, and in the URLs of hyperlinks. Though parsing page content is a difficult problem in general, in this case the parsed content is non-malicious, and thus presumably non-obfuscated. Moreover, if a script is missed there is no risk to security as it is not added to the whitelist. Support for applications that use uncommon combinations of HTML and JavaScript might be better provided by adapting a sophisticated server-side filter to identify and hash all scripts in the static content of a page.

Web applications can also be developed from higher-level languages and/or specifications, in lieu of authoring HTML and JavaScript directly. For example, Links [10] compiles programs written in a special purpose language and the Google Web Toolkit (GWT) [4] compiles Java programs into web applications that, on the client-side, are implemented in JavaScript and HTML. For these applications, we would like the toolkit compilers to introduce the security hook automatically. To show that this is feasible, we modified Links to generate and insert the whitelist for the emitted scripts; the changes to the compiler were fairly small—only 60 LOC. GWT is distributed only in binary form, so we were unable to attempt a similar modification.

Finally, for applications that generate HTML dynamically (e.g., by using server-side PHP, JSP, etc.), the generated HTML must include the policy. Fortunately, emitted scripts often appear directly in the page-generating code, so it is straightforward to copy them into a document on which to run our script identification tool. One could imagine authoring language-specific tools to do this automatically.

DOM sandboxing In contrast with whitelist policies, DOM sandboxing is simple enough to apply by hand. We modified Blixlwiks, a custom blog and wiki engine that we use to run `cyclone.thelanguage.org`, to implement DOM sandboxing. This involved writing one function to escape JavaScript strings, another to output sandboxed content, and a third to output the hook function in each page. In total we added about 40 lines of code, plus the 34-line hook function shown in the appendix.

We expect this would be just as easy for a web application written in a templating language such as PHP. Templating languages make it easy to insert content into boilerplate HTML, and

1	`<SCRIPT a=">" SRC="xss.js"></SCRIPT>`
2	`<SCRIPT>document.write("<SCRI");</SCRIPT>PT SRC="xss.js"></SCRIPT>`
3	`<IMG SRC="javascript:alert('XSS');">`
4	`<DIV STYLE="background-image: url(javascript:alert('XSS'))">`
5	`<XML ID=I><X><C><![CDATA[<IMG SRC="javas]]><![CDATA[cript:alert('XSS');">]]>`

Table 1: A representative sample of attack vectors.

also provided functions for quoting content as strings.

4 Experimental Evaluation

We constructed a test suite based on 61 XSS attack vectors published by `ha.ckers.org` [16]. Each vector is a snippet of HTML and JavaScript that can form the foundation of a script injection attack. The vectors incorporate obfuscation designed to evade common server-side filters, and they have been tested to ensure execution on at least one major browser. BEEP successfully defeated each attack in the suite.

The attack vectors Table 1 contains a representative sample of the 61 vectors. The first vector in Table 1 will confuse a filter that looks for `<script src=...>` without expecting additional attributes. The second vector is more complicated and relies on the way in which a browser interprets the text written into the document by JavaScript functions. A vulnerable browser would interpret the result of executing the first part of the vector (enclosed within well-formed script tags) concatenated with the remaining part as a legal script element that includes the malicious script file "xss.js". The third vector might evade a server filter that does not expect scripts to appear as attributes within a tag intended to include images in a document. The fourth vector is a version of the primary vector used by the Samy worm. The final vector exploits a feature of some browsers that allows an XML document containing JavaScript to be inlined in an HTML document.

Experimental setup and results To model a script injection attack on a web site, we injected each of the attack vectors, one per document, into several of the pages that comprise the target application. For the whitelist policies, we used S3AjaxWiki, and for DOM sandboxing we used Blixlwiks. Each S3AjaxWiki page is derived from a common template for all wiki pages that includes a whitelist security hook function (generated by our tool) in the head of the document. A fragment of an S3AjaxWiki page, as used in our test suite, is shown in Figure 4; the script that appears in the body of the page is the injected attack vector. Our modifications to Blixlwiks were described above.

Of the 61 vectors, we found that 17 were accepted by Konqueror, 9 were accepted by Safari, and 33 were accepted by Opera as legal scripts (the remaining vectors are accepted by some other browser). Both the whitelist policy and the DOM sandbox successfully stopped all these vectors. Over all the whitelist test cases, Konqueror invoked the `JSSecurity.afterParseHook` function defined by S3AjaxWiki a total of 746 times. Of these authorization checks, 17 calls resulted in the hook rejecting the script, corresponding to one hook rejection for each of the 17 vectors that are effective for Konqueror. The corresponding numbers for Safari are 754 calls with 9 rejections; and for Opera 1020 calls and 33 rejections.

Measurements indicate that the performance overhead of executing a hook function each time a script is parsed is negligible. We tested two implementations of the whitelist hook function: one

```
<html><head>
   <script src="hook.js" type="text/javascript"></script>
   <title>StartPage</title>
   <script src="../js/S3Ajax.js" type="text/javascript"></script>
   <script src="../js/wiki.js" type="text/javascript"></script>
   ...
</head><body>
   <h1 id="page_title">StartPage</h1>
   <div id="content">
     <p>This is the default page content.  You should edit this.</p>
     <SCRIPT SRC=xss.js></SCRIPT><!-- This is the injected attack vector -->
   </div>
</body></html>
```

Figure 3: A sample from the test suite that uses a whitelist policy.

used native support provided by the `JSSecurity` library for the computation of SHA1 digests and the base64 encoding of byte arrays as strings (written in C); the other implementation performed these operations directly in JavaScript. We benchmarked Konqueror on a Pentium 4, 2.3 GHz laptop running Linux 2.6; Safari on a 1.67 GHz G4 PowerBook running MacOS X 10.4. We were unable to time hook invocations in Opera since our solution for this browser was pure JavaScript with no browser modifications.

For the version using native support, the elapsed time for each hook function invocation for Konqueror was in most cases beyond the resolution of the `ftime` millisecond timer that we used (the maximum elapsed time was 0.016 seconds; the average 0.001 seconds). For the pure JavaScript version, the average elapsed time was 0.05 seconds, with a maximum of 0.365 seconds. The performance of the native version in Safari was also beyond the resolution of the millisecond timer, however, the average elapsed time for the JavaScript version was only 0.005 seconds with a maximum time of 0.067 seconds.

The DOM sandboxing policy that we applied to Blixlwiks resulted in 197 hook calls in Konqueror, 318 calls in Safari and 220 calls in Opera. The elapsed time for each hook invocation was beyond the resolution of the timer for Konqueror and Safari. As with the whitelist policy, we are unable to report timing numbers for Opera.

Our test suite only injected relatively short XSS attack vectors. A realistic attack vector, such as the one used in the Samy MySpace worm is much larger. In such cases, the pure JavaScript approach for hashing is likely to take unacceptably long (on the order of several seconds).

5 Related Work

Server-side techniques to protect against script injection attacks have been reported extensively in the literature. A systematic approach to filtering injected attacks involves partitioning trusted and untrusted content into separate channels and subjecting all untrusted content to application defined sanitization checks [13]. Su and Wassermann [19] develop a formal model for command injection attacks and apply a syntactic criterion to filter out malicious dynamic content. Applications of taint checking to server programs that generate content to ensure that untrustworthy input does not flow to vulnerable application components have also been explored [11, 6, 22].

While insights borrowed from server-side filtering can, in principle, be brought to bear in the design of security hook functions, our work is most closely related to other client-side techniques to protect users from malicious web content.

Noxes [7] is a purely client-side method that aims to defend against cross-site scripting by disallowing the browser from contacting "bad" URLs. It has general rules for blacklisting and whitelisting websites in which links that are statically present in the page are placed in the whitelist, while dynamically-generated links are disallowed. Because Noxes policies blacklist script-generated links, they can be quite restrictive for applications with substantial client-side logic, e.g., in Ajax-enabled applications. Moreover, link blacklisting is not enough to prevent all attacks, e.g., those not in violation of the same origin principle, as was the case of the Samy worm. By contrast, our policies either permit or deny execution of entire scripts, as determined by the host site.

BrowserShield [15] proposes a two-step architecture to protect against browser vulnerabilities such as buffer overruns. Prior to being loaded in a browser, a document is rewritten (say, at a firewall or web proxy) so that certain trusted JavaScript functions mediate the access of the document tree by the untrusted scripts in the document. The policy (as embodied by the rewriting step) is then enforced in the browser by the trusted JavaScript functions. BrowserShield policies are far richer than ours, as they can mediate individual script actions, whereas we consider only whether to run the script at all. As a result, BrowserShield has a correspondingly higher implementation (and trust) burden, especially since parsing HTML and JavaScript are non-trivial when accommodating many possible browsers, as we have argued. Finally, and perhaps most importantly, in the main usage mode for BrowserShield, the policy is expected to be specified independently of the site that serves the content. In this mode, it is unclear how a policy might distinguish between malicious republished content that, say, accesses a document's cookie from a server-trusted script that does the same. Combining BEEP with BrowserShield might result in the best of both worlds: the system would accurately filter illegal scripts, but would allow client-based policing of server-provided scripts.

Jackson et al. [5] describe several unexpected repositories of private information in the browser's cache that could be stolen by XSS attacks. They advocate applying a refinement of the same-origin policy [17] to cover aspects of browser state that extend beyond cookies. By allowing the server to explicitly specify the scripts that it intentionally includes in the document, our approach can also be thought of as an extension of the same-origin policy. In particular, our policies ensure that all scripts that executed in the page are trusted by the site from which the page originated.

There are some analogies between our work and intrusion detection systems (IDS). The filtering problem arises in network intrusion detection (IDS) systems [14]. In particular, just as different browsers accept and render HTML differently, different operating systems may accept and process packets slightly differently, even packets that are ill-formed. As a result, the IDS might think a packet is harmless because it is ill-formed, but in fact a particular OS might accept it and thereby be exploited. Our solution is analogous to a host-based intrusion detection system (HBIDS) [21, 2]. In these systems, a program's correct behavior is characterized in advance in terms of actions like system calls, and an execution monitor detects when a program deviates from its allowable behavior. In BEEP, the allowable behavior is defined by the web site in terms of whitelisted (or non-sandboxed) scripts, and attempts to deviate from it are prevented by the browser.

6 Conclusions

This paper has presented Browser-Enforced Embedded Policies (BEEP), a technique for defeating script injection attacks on web applications. The broad diversity of what browsers will accept as valid HTML makes script detection/filtering difficult at the server; to the contrary, the browser has perfect knowledge of when it will execute a script. We exploit this insight to allow servers to embed a *security hook* function in their pages that will be executed in a suitably-modified browser prior to executing a script. The hook function can thereby remove malicious scripts with perfect precision when employing a server-provided whitelist or sandbox. Changes to applications and browsers are small and localized, and performance overhead is small, make possible deployment practical. We plan to further explore the possibilities of BEEP, experimenting with additional policies and greater policy language support. We plan to make all our code available on the web for public use.

References

[1] Malicious HTML tags embedded in client web requests. CERT Advisory CA–2000–02, February 2000.

[2] Henry Hanping Feng, Jonathon T. Giffin, Yong Huang, Somesh Jha, Wenke Lee, and Barton P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2004.

[3] Jesse James Garrett. Ajax: A new approach to web applications. `http://www.adaptivepath.com/publications/essays/archives/000385.php`, feb 2005.

[4] Google web toolkit. `http://code.google.com/webtoolkit/`.

[5] Collin Jackson, Andrew Bortz, Dan Boneh, and John C. Mitchell. Protecting browser state from web privacy attacks. In *Proceedings of the 15th ACM World Wide Web Conference (WWW 2006)*, 2006.

[6] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *PLAS '06: Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*, pages 27–36, New York, NY, USA, 2006. ACM Press.

[7] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: A client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 21st ACM Symposium on Applied Computing (SAC), Security Track*, April 2006.

[8] Amit Klein. DOM based cross site scripting or XSS of the third kind. `http://www.webappsec.org/projects/articles/071105.shtml`, July 2005.

[9] V. T. Lam, S. Antonatos, P. Akritidis, and K. G. Anagnostakis. Puppetnets: Misusing web browsers as a distributed attack infrastructure. In *13th ACM Conference on Computer and Communications Security*. ACM, 2006.

[10] Links: Linking theory to practice for the web. `http://groups.inf.ed.ac.uk/links/`.

[11] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *20th IFIP International Information Security Conference*, 2005.

[12] Les Orchard. S3AjaxWiki. `http://decafbad.com/trac/wiki/S3Ajax`.

[13] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection (RAID 2005), volume 3858 of Lecture Notes in Computer Science, pages 124–145*, Seattle, WA, 2005.

[14] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., January 1998.

[15] Charlie Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI)*, October 2006.

[16] RSnake. XSS (cross site scripting) cheat sheet. Esp: for filter evasion. `http://ha.ckers.org/xss.html`.

[17] Jesse Ruderman. The same origin policy. `http://www.mozilla.org/projects/security/components/same-origin.html`, August 2001.

[18] Samy. I'm popular. `http://namb.la/popular/`, October 2005. Description of the MySpace worm by the author, including a technical explanation.

[19] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *POPL '06: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–382, New York, NY, USA, 2006. ACM Press.

[20] HTML Tidy project page. `http://tidy.sourceforge.net/`.

[21] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2001.

[22] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *15th USENIX Security Symposium*, July 2006.

A DOM Sandboxing and Frames

HTML provides two kinds of frame constructs, `<FRAME>` and `<IFRAME>`, which can be used to embed sub-documents into a document. According to the DOM, these sub-documents are separate document trees only loosely coupled with the document tree of the enclosing document. This presents a complication for hook functions implementing a DOM sandboxing policy. When ascending the document tree starting at a script node that appears within a frame, the root node of the document is the root of the frame's document, not the root of the enclosing document. To continue searching upward, we must find the position of the frame in the parent document, and search from there. The DOM does not make this easy, but it can be done by searching in the parent for the sub-document.

A complete implementation of `afterParseHook` that handles these complications is given below.

```
window.JSSecurity.afterParseHook = function (elt, code, ev) {
   while(true) {
      if(!elt.parentNode) {
         var parentWindow = elt.defaultView;
         if(parentWindow.parent == parentWindow) {
            return true;
         }
         parentWindow = parentWindow.parent
         var iframesInParent = parentWindow.document.getElementsByTagName("iframe");
         var framesInParent = parentWindow.document.getElementsByTagName("frame");
         for(i=0;i<iframesInParent.length;i++) {
            if(iframesInParent[i].contentDocument == elt) {
              elt = iframesInParent[i];
            }
         }
         for(i=0;i<framesInParent.length;i++) {
            if(framesInParent[i].contentDocument == elt) {
              elt = framesInParent[i];
            }
         }
      }
      if(elt.tagName.toLowerCase() == "div") {
         var classAtt = elt.getAttribute("class");
         if(classAtt && classAtt.toLowerCase() == "noexecute") {
           return false;
         }
      }
      elt = elt.parentNode;
      if(!elt) {
        return false;
      }
   }
   return true;
}
```

B User Script for Opera

```
var JSSecurity = {
   b64sha1 : function (x) { ... },//JavaScript implementation of SHA1 omitted
   afterParseHook : function () { return true; } //Default hook
};

/* for <script> elements */
window.opera.addEventListener(
   'BeforeScript',
   function (e) {
     if (window.top.JSSecurity.afterParseHook &&
         !window.top.JSSecurity.afterParseHook(e.element, e.element.text, "")) {
       e.preventDefault();
     }
   },
   false
);

/* for <script src=...> elements */
window.opera.addEventListener(
   'BeforeExternalScript',
   function (e) {
      if (window.top.JSSecurity.afterParseHook &&
          !window.top.JSSecurity.afterParseHook(e.element, e.element.src, "")) {
         e.preventDefault();
      }
   },
   false
);

/* for events about to take place */
window.opera.addEventListener(
   'BeforeEventListener',
   function (e) {
      if (window.top.JSSecurity.afterParseHook &&
          !window.top.JSSecurity.afterParseHook(e.event.target, e.listener, e.event.type)) {
         e.preventDefault();
      }
   },
   false
);

/* for JavaScript URLs
   Partial solution -- don't have access to the DOM node */
window.opera.addEventListener(
   'BeforeJavascriptURL',
   function (e) {
      if (window.top.JSSecurity.afterParseHook &&
         !window.top.JSSecurity.afterParseHook(e.target, e.source)) {
         e.preventDefault();
      }
   },
   false
);
```