Defeating Script Injection Attacks with Browser-Enforced Embedded Policies

Trevor Jim
AT&T Labs Research

Nikhil Swamy
University of Maryland, College Park

Michael Hicks
University of Maryland, College Park

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 make filtering quite difficult, however, and attacks such as the Samy and Yamanner worms have exploited filtering weaknesses. This paper proposes a simple alternative mechanism for preventing script injection called Browser-Enforced Embedded Policies (BEEP). The idea is that a web site can embed a policy in 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.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*unauthorized access, invasive software*

General Terms

Security

Keywords

Script injection, cross-site scripting, web application security

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) [2], and can themselves become participants in attacks on the web site and on others [16]. 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 [23, 32, 21].

WWW 2007, May 8–12, 2007, Banff, Alberta, Canada.
ACM 978-1-59593-654-7/07/0005.

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 [30] and Yamanner [3] worms).

We propose a new technique to prevent script injection attacks, based on the following two observations:

Observation 1: Browsers perform perfect script detection. If a browser does not parse content as a script while it renders a web page, that content will not be executed.

Observation 2: The web application developer knows exactly what scripts should be executed for the application to function properly.

The first observation implies that the browser is the ideal place 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.

The second observation implies that the web site should supply the filtering policy to the browser—it can specify which scripts are approved for execution and the browser will filter the rest. In short, the web site sets the policy and the browser enforces it. We call this strategy *Browser-Enforced Embedded Policies (BEEP)*.

There are many possible ways to implement BEEP. In this paper, we have used a method that is easy to implement while still permitting very general policies. In our implementation, 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 approves it.

Our implementation of BEEP has several advantages.

Flexible policies. The security hook can be any function that can be implemented in JavaScript. So far we have

implemented two simple kinds of policies (but we are not restricted to these policies).

Our first policy is a *whitelist*, 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 is not in the list is rejected.

Our second 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]

While these policies are sufficient to stop injected scripts, other policies are also possible. For example, the hook function could also notify the web site when an injected script is found. A hook function could even analyze scripts and permit only a restricted class of scripts to execute. Policies can be easily modified over time: the new policy is simply embedded in the site's pages and will be enforced by browsers from then on.

Complete coverage. With policies like the whitelist and DOM sandbox, BEEP detects and filters all injected scripts, under two conditions. First, to use these policies, all approved scripts must be identified by the web site in advance either directly (by enumerating them) or indirectly (by identifying where scripts cannot occur); this is straightforward for most applications (and is discussed in more detail in Sections 3.5 and 4.2). Second, the browser must install the security hook before any other scripts on the page are executed, to ensure complete mediation. This is easily accomplished: defining the hook as the first script in the document head ensures it will be parsed first. Together, these conditions imply that any non-approved script will be rejected before it has a chance to run.

Ease of deployment. Our method requires browser modifications, but has been chosen to minimize them. For example, the places where browsers need to be modified are easily identified: we simply locate places in the source code where the browser invokes the JavaScript interpreter. These are the points where the browser has identified a script in a web page. At this point in the source code, the browser has gathered together all of the information needed to invoke the JavaScript interpreter, and we only need to insert code to invoke the interpreter on our security hook function first. Depending on the result of this first invocation, we will either execute the script from the page, or skip it. We successfully modified the Konqueror and Safari browsers to support security hooks, and we implemented partial support in the closed-source Opera browser. These changes required just over 650 lines of code in the first two cases (compared to several hundred thousand for the browsers' rendering engines), and just over 100 lines of JavaScript for Opera.

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

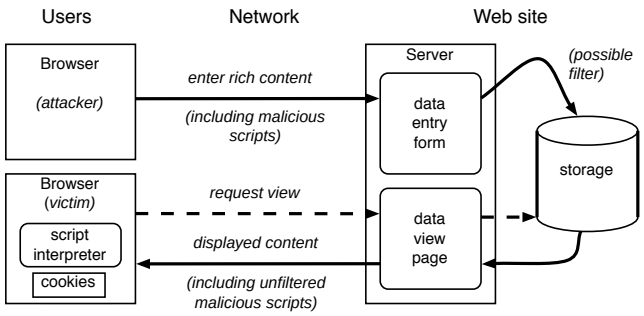

Figure 1: Script injection attack on a typical Wiki/Blog-based site, like MySpace.

Web applications must also be modified to use BEEP, but the changes are simple and localized. We will show how we modified some existing web applications to embed policies, and describe some simple tools we built to help in this process.

Finally, deployment can proceed incrementally. 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. Moreover, while we intend that 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.

Moderate overhead. When a browser renders a BEEP-enabled web page, there is some additional overhead for parsing the security hook function and executing the hook function whenever a script is parsed. After running some simple experiments we found that rendering overheads averaged 14.4% for whitelist policies and 6.6% for sandbox policies, typically amounting to a fraction of a second. These percentages do not include network time, which would further reduce overhead if accounted for.

The next section presents some background, and the remainder of the paper explains our BEEP technique and policies, describes our implementation, and presents experimental results. The paper concludes by comparing BEEP to related work.

2. BACKGROUND

Script injection, or cross-site scripting, is a very common vulnerability: according to MITRE's CVE list [20], it is the most common class of reported vulnerabilities, surpassing buffer overflows starting in 2005. Here we review script injection attacks and illustrate why it is difficult to filter scripts using standard server-side techniques.

2.1 Script Injection

We are concerned with attacks that cause a malicious

```
 1.  <html><head>
 2.    <script src="a.js"></script>
 3.    <script> ... </script>
 4.    <script for=foo event=onmouseover> ... </script>
 5.    <style>.bar{background-image:url("javascript:alert('JavaScript')");}</style>
 6.  </head>
 7.  <body onload="alert('JavaScript')">
 8.    <img id=foo src="image.jpg">
 9.    <a class=bar></a>
10.    <div style="background-image: url(javascript:alert('JavaScript'))">...</div>
11.    <XML ID=I><X><C><![CDATA[<IMG SRC="javas]]><![CDATA[cript:alert('XSS');">]]>
12.    <meta http-equiv="refresh"
        content="0;url=data:text/html;base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4K">
13.    <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;>
14.    <img src=javascript:alert(&quot;3&quot;)>
15.  </body></html>
```

Figure 2: Ways of embedding scripts in web pages.

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) [2]. 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 [16].

Script injection can be achieved in many ways. In cross-site scripting (XSS), the attacker often exploits web sites that insert user-provided text into pages without properly filtering the text. For example, members of on-line communities like MySpace, Blogger, and Flickr can 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. If a malicious member manages to include a script in his content, any viewers of that content would run the script with the privileges of the site. And if a viewer were also a member of the site, the script could access 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 "not found" response 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 in the visitor'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`.

Another possible attack scenario [14] exploits the dynamic nature of JavaScript-enabled web pages, where the HTML content served from the web server is altered in the browser by the execution of scripts. For instance, a site might be constructed so that a URL of the form

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

produces personalized content using a static HTML page in combination with an embedded script. In particular, the script can use features like `innerHTML` and `document.write` to modify the content of the page at the browser, personalized according the value of "name." This opens the possibility that a 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.

This can be taken to an extreme: web applications like S3AjaxWiki [22] 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 in the browser. For example, if some content contains "`<script>`", the special characters '`<`' and '`>`' can be escaped as HTML entities '`<`' and '`>`.' This will cause the browser to display the text "`<script>`" instead of executing the script. When combined with technologies like tainting [21] 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:

Multiple vectors. 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 obtained 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 embeds script in XML appearing in HTML; note that the script can be broken across multiple CDATA sections. Line 12 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.

Encodings and quoting. Quotes that delimit content and encodings of special characters add further 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. The base64 encoding of line 12 in Figure 2 is one example; others are line 13, which uses a javascript:URL that has been character encoded, and line 14, which uses HTML entity encoding to hide quote characters in a script (this can confuse filters that look for literal quote characters).

Browser quirks. 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 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 than show a blank page or an error message. This leads to some surprising script embeddings. For example, some browsers allow newlines or other non-printing characters to appear in the "`javascript:`" portion of a javascript:URL, so that

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

will result in script execution. For another example,

```
<img src=‘javascript:alert("Hello 'world'")‘>
```

can execute in some browsers, even though backquote (‘) is not a standard quote character in HTML or JavaScript. Even something completely malformed such as

```
<img """><script>alert("ack")</script>">
```

executes in some browsers.

2.3 Real world examples

All of these issues—multiple vectors, encodings, and quirks in browsers—make script detection a hard problem, and give rise to dozens of techniques for hiding scripts from detection, available on public sites, e.g., `ha.ckers.org` [26]. These techniques are effective in practice. For example, the Samy worm defeated script filtering on MySpace in October 2005. The worm caused over a million users to add "Samy" to their MySpace "friends" list, and portions of the site had to be closed down for several hours to repair the infection. The worm's author has written a nice description of how he developed the worm and got it past the filters [30].

The Yamanner worm is another example. Yamanner attacked Yahoo! Mail in June 2006 and infected almost 200,000 users [3]. It injected a script into HTML email, and propagated when users read their mail. Yahoo! had filtering in place, but Yamanner defeated it with input looking something like this:

`<img src="...` `target="onload="`*malicious script*`">`

This input is completely harmless as-is, but Yahoo!'s filter deleted the target attribute (which can be used for certain information-disclosure attacks). This produced an injected script in an onload event handler:

`<img src="..." onload="`*malicious script*`">`

So, an effective filter must not only detect scripts, but also ensure that it does not *introduce* scripts.

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 to allow or disallow 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). In the rest of this section, we describe one possible implementation of BEEP that provides complete coverage against script injection attacks under typical assumptions.

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 [28]: visitors are willing to execute scripts in site content, since they assume scripts to be tacitly endorsed by the site. Likewise, visitors 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.

The security hook must implement complete mediation to be an effective defense: no script may escape scrutiny by the security hook before the script runs. This implies that 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. We rely on this behavior of the browser by defining the hook function as the first script in the `<head>` element of the document. (It is straightforward to structure a web application so that no dynamic content is ever included prior to the security hook definition in the `<head>` of each web page.) Note that putting the security hook function first also ensures that it is tamper-proof: any malicious scripts that would modify the hook will be parsed after the hook is installed, and so be filtered by the hook and prevented from running.

When a modified browser parses a script, it invokes the `afterParseHook` function with two arguments: the text of the parsed script and the DOM element of the parsed script. Thus when rendering the document fragment

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

the browser invokes `afterParseHook` on the text of the script, i.e., `"alert('hello')"` and the DOM node of the `<body>` element. 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*. We discuss these next.

3.3 Whitelists

Most current web applications embed scripts in their web pages. Typically, the web application developer knows precisely which scripts belong in each page (but see Section 3.5). Therefore, the developer can write a security hook that checks that every script encountered by the browser is one of these known scripts; in other words, a 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, if the script `<script>alert(0)</script>` is known, then `whitelist [SHA1("alert(0)")]` should be defined; if an included script such as `<script src="aURL"/>` is known, then `whitelist [SHA1("aURL")]` should be defined.

Here is a sample implementation:

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

The `SHA1` function could be defined as part of the script in which the above code appears, or it could be part of library provided by the browser to security hooks. The latter is clearly preferable: while JavaScript versions of cryptographic functions exist [12], they perform far worse than native implementations (cf. Section 5.2).

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 4.2.

We have also experimented with an alternative whitelist implementation in which the array is indexed not by cryptographic hashes, but by the entire content of the approved script. This results in larger pages and better security (by avoiding collisions) but eliminates the overhead of hashing.

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. This is useful if some parts of the page should be allowed to contain unknown scripts, e.g., for third-party ads.

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 too simplistic. An attacker can cause a malicious script to break out of the sandbox by injecting content of the form:

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

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, 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, even when the string contains a `</div>` that attempts to prematurely close the `<div>` tag of the "noexecute" node. 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. The complete implementation is available at the BEEP web site [10].

3.5 Discussion

Using BEEP policies, the web site and the browser can cooperate to provide complete coverage against injected scripts. This is because (1) the hook function implements complete mediation, scrutinizing all scripts before execution, and (2) the unapproved (and possibly injected) scripts are clearly distinguished from approved scripts by the policy and will therefore be rejected.

There may be cases in which a web site wishes to approve a script provided by a third party. For example, web sites may use ad networks like AdBrite [1] to display advertisements on their pages. Typically, the ad network will provide a snippet of JavaScript for the web site to include in its pages. When the page is displayed in the user's browser, the JavaScript will then retrieve content to display the ad, overwriting the original JavaScript. This can be accommodated by BEEP; the web site simply needs to approve this "ad-retrieving" script along with its own—it would either place it in the whitelist, or outside any sandbox. However, blindly trusting this third-party script is not without risk: the script may actually be malicious, e.g., part of a scheme to perform click-fraud [5]. BEEP does not provide any guidance on whether to trust a third party.

Scripts can also dynamically create new scripts and insert them into the page using DOM operations; e.g., some ads are implemented this way. If a script is trusted, BEEP implicitly trusts the scripts it installs. Given that trusted scripts are already quite powerful (e.g., the script can modify any part of the document, including the security hook) this is not an additional risk. Indeed, BEEP does not suggest whether a user should trust a particular site or the scripts it provides. Rather, BEEP ensures that a browser only runs those scripts actually endorsed by a given trusted site.

4. IMPLEMENTATION

Our implementation of BEEP requires changes in browsers to support security hooks, and changes in web applications to embed policies as hooks in web pages. We successfully modified several browsers and web applications and found the changes to be small and straightforward, presenting little barrier to adoption. Code for our browser modifications, benchmarks, and test cases is available online [10].

4.1 Browser modifications

Konqueror and Safari. We modified the Konqueror and Safari browsers to support security hooks. The browsers are related: Safari's rendering engine 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++. We changed or added roughly 650 lines of code in both cases, and we added another 650 lines for a standard SHA-1 implementation.

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, most of the required changes were limited to the interface between the HTML and JavaScript engines. This interface is bidirectional—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 `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.

DOM sandboxing required some additional changes. To enforce DOM sandboxing, the `afterParseHook` must traverse the document tree from the location of the script towards the root of the tree. However, in a few cases the HTML parsers in Safari and Konqueror do not maintain a well-formed document tree when parsing JavaScript. This occurs, for instance, when parsing scripts that appear in the attributes of HTML elements, and this prevents the hook from determining whether or not a script is contained within a "noexecute" node. Therefore, we changed the HTML parsers to make the DOM tree well-formed in these cases. Note that the ECMA and DOM standards for JavaScript and HTML do not require the document tree to be well formed during parsing.

Opera. We 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, within 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:URLs in Opera. The complete User JavaScript is 79 commented lines of code, plus 137 lines for a SHA-1 implementation in JavaScript.

Mozilla Firefox and Internet Explorer. We have begun to explore an implementation of security hooks in the Firefox browser. We have not yet investigated Internet Explorer. Both browsers have extensions (Greasemonkey for Firefox and Trixie for IE) that can function something like

the User JavaScript provided by Opera. However, these extensions are not sufficient to implement BEEP because scripts embedded in a page can execute before the extensions are triggered.

4.2 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, a simple tool can 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 pages prior to deployment—i.e., when the pages do not contain any user content all scripts are legal.

We have written such a tool based on the Tidy HTML parser [33]. 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. Any legitimate scripts that are missed by this process will cause missing functionality, and hence should be quickly discovered; and illegitimate scripts will not be added to the whitelist through this process. 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 [18], Hop [8], and Google Web Toolkit (GWT) [6] are all systems that compile web applications into server- and client-side programs, where the client-side programs consist of HTML and JavaScript. To show that systems like these can automatically compile security hooks into each web page, we modified Links to generate and insert the whitelist for the emitted scripts; the changes to the compiler were fairly small—only 60 LOC. We did not attempt similar modifications to Hop or GWT due to time constraints.

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. To apply DOM sandboxing to a web application we first need to identify all portions of web pages where user-provided content can appear. This can be solved by known techniques (such as tainting) and in any case is routine for any web application that already applies filtering. Next, we need to escape the content as a JavaScript string and insert boilerplate code from Section 3.4. This just requires some minimal support from web authoring tools.

For example, 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. We also enlisted the aid of the authors of Continue [15], a web application for managing conferences; they added DOM sandboxing to Continue in 10 lines of code. It should be just as easy to modify a web application written in a templating language such as PHP. Templating languages make it easy to insert content into boilerplate HTML, and also provide functions for quoting content as strings.

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

Figure 3: Page of a web application containing a BEEP policy (hook.js) and an injected script.

5. EXPERIMENTAL EVALUATION

We conducted experiments to verify the effectiveness of our BEEP implementation and to measure its overhead. We found that, as expected, our implementation defeated a wide array of known attacks, and imposed a low to moderate overhead for typical web sites.

5.1 Effectiveness

BEEP as described in Section 3 should provide complete protection against injected scripts, assuming we have intercepted all invocations of the JavaScript interpreter in the browser source code. To verify this, we constructed a test suite of attack vectors, each of which is a snippet of HTML and JavaScript which might be injected into user-provided content as the first step of an attack. BEEP-enhanced versions of S3AjaxWiki, Blixlwiks and Continue defeated all of the attacks.

Our test suite is based on 61 XSS attack vectors published by `ha.ckers.org` [26]. The vectors incorporate obfuscation to evade common server-side filters, and they have been tested to ensure execution on at least one major browser. Of the 61 vectors, 17 can be used to mount a successful attack against Konqueror, 9 against Safari, and 33 against Opera. The remaining vectors can be used to attack other browsers.

To model a script injection attack on a web site, we manually inserted each attack vector into the user-content of the target application. In addition, we added whitelist policies to S3AjaxWiki and static pages of Continue, and we modified Blixlwiks and Continue to insert DOM sandboxing policies. Then we verified that BEEP detected and nullified every injected attack vector.

A fragment of an S3AjaxWiki page with a whitelist policy, as used in our test suite, is shown in Figure 3. Each

S3AjaxWiki page is derived from a common template for all wiki pages. The template includes a whitelist security hook function (generated by our tool) that appears as the first script in the head of the document. The script that appears in the body of the page is the injected attack vector. A similar scheme was used to specify whitelist policies in web pages of Continue.

The changes to support DOM sandboxing policies in Blixlwiks and Continue were described in Section 3.4.

5.2 Overhead

We used our BEEP-enabled Safari browser to measure the page load time for the thirty most popular US web sites (according to `alexa.com`), both with and without our policies installed. Repeated timings for four of the sites varied considerably, in both the modified and unmodified Safari, so we eliminated them from consideration; the statistics we report here are for the remaining twenty-six sites. All claims of statistical significance are made here with probability of the null-hypothesis $p < 0.05$. Overheads were low to moderate, and indicate that, on average, DOM sandboxing policies are less expensive than whitelist policies.

Whitelist policies. For our whitelist experiments we automated the following process. First, we retrieved the front page of each of the 26 sites, along with all elements needed for proper rendering, and stored them locally, to remove the variability of the network from our measurements. Next, we used our script identification tool (cf. Section 4.2) to compute a SHA-1 hash of each script that appeared within a page. We then inserted a script defining an `afterParseHook` function as the first element in the `<head>` of each document.

We benchmarked Safari version 2.0.4 linked with our modified version of the WebKit HTML engine, revision 16269 running on MacOS X 10.4. All experiments were conducted on a 1.67 GHz G4 PowerBook laptop with 1.5GB of main memory. Each of the 26 web pages was loaded in the browser twenty times, and we measured the total time taken by the browser to load each document, using Safari's loading-time measurement feature [29]. The total time to load the 520 unmodified pages was 698.2 seconds, compared to 798.6 seconds to load the pages that included whitelist policies, for an average overhead of 14.4%. The slowdown due to whitelist policies was statistically significant in 19 of the 26 pages. The greatest overhead was 40% for the `megaupload.com` front page, which includes several large scripts inline. Recall that these measurements do not include network latency, which would tend to mask the overhead of BEEP. Furthermore, the average increase in mean page load time was only 0.2 seconds per page.

We also measured the performance of policies in which the whitelist array is indexed using the content of an approved script rather than its hash. We found this approach to increase the size of each web page by 8KB or 13% on average. We found no clear difference in the load times of pages that used content-based whitelists and hash-based whitelists. Of the 26 web pages, the difference in load times was not statistically significant in 13 cases; 7 showed a statistically significant speedup when using content-based whitelists, while 6 showed a statistically significant speedup when using hash-based whitelists. Note, however, that native support for hashing is critical for acceptable performance of a hash-based whitelist policy. For example, using a purely JavaScript implementation of SHA-1 caused `megaupload.com` to take nearly 10 times longer to load than when using a native implementation.

DOM Sandbox policies. We also modified the 26 web pages to include a trivial DOM sandboxing policy where no `<div>` or `<span>` in the page is marked with a "noexecute" tag. The purpose here is to measure the cost of traversing the DOM to validate scripts for the common case in which no malicious script has been injected in a page. It took 744.0 seconds to load the 520 DOM-sandbox enabled web pages in Safari, a 6.6% average overhead compared to the unprotected case. The maximum overhead of 25.9% was observed for `youtube.com`. The slowdown due to DOM policies was statistically significant in only 13 of the 26 cases.

Our measurements showed that the difference in load times between whitelist policies and DOM sandboxing policies are statistically significant in only 17 of 26 cases. On average whitelist policies are 7.3% slower than DOM sandboxing policies, ranging from a maximum of a 28% slowdown to a 7.8% improvement in load time performance. Over all the top 30 web pages, we found that the average depth of a script in the DOM is only about 11. This indicates that traversing the DOM to authorize a script is often likely to be cheaper than computing cryptographic hash functions over the text of a script.

6. RELATED WORK

We are not aware of any other implementation of BEEP, but we have seen discussion of a related idea in the Mozilla forums: Markham has proposed communicating some policies on scripts from web site to browser in an HTTP header [19]. In comparison to our work, his selection of policies is fixed, e.g., "only scripts in the header are allowed to execute," and do not seem to include policies that could, for example, allow some event handlers in a page to execute, while preventing others. Our implementation is more flexible and can accommodate such policies, which appear needed to handle common practice in web applications. Using HTTP headers also seems to require more intrusive changes to web applications and browsers than our work. Relative to that proposal, Schmidt suggested the idea of using a DIV element as a DOM sandbox, but did not address node-splitting [31].

Mozilla has a feature called *signed scripts* [27]. Digital signatures could be used as a basis for BEEP, providing a way to distinguish between approved and non-approved scripts, but Mozilla's signed scripts cannot be used this way. Instead, scripts are signed when they require *additional* privileges, such as writing to local files, and the absence of a signature does not constrain scripts.

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 [23]. Su and Wassermann [32] 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 [21, 11, 35].

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 [13] 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 web sites 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 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 [25] and CoreScript [36] propose to defeat JavaScript-based attacks by rewriting scripts according to a security policy prior to executing them in the browser. In BrowserShield, the rewriting process inserts trusted JavaScript functions to mediate access to the document tree by untrusted scripts. CoreScript policies are specified as a kind of edit automata [17]. BrowserShield and CoreScript 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, these systems have 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, and since rewriting may need to be applied during script execution [36]. Finally, and perhaps most importantly, in the main usage mode for these systems, 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 client-side rewriting policies might result in the best of both worlds: BEEP would accurately filter illegal scripts, while client-side rewriting could police server-provided scripts for less-trusted sites.

Hallaraker and Vigna [7] modified Mozilla to monitor the JavaScript operations of a web page and invoke countermeasures against malicious behavior. This permits fine-grained policies on JavaScript execution in the browser. However, the work does not address communicating policies from the web site to the browser.

Jackson et al. [9] 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 [28] 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; we believe this is actually the assumption of most users.

There are some analogies between our BEEP policies and intrusion detection systems (IDS). Filtering is a problem in network intrusion detection (IDS) systems [24] too. 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) [34, 4]. 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.

7. CONCLUSIONS

This paper has presented Browser-Enforced Embedded Policies (BEEP), a simple technique for defeating script injection attacks on web applications. The broad diversity of browser rendering algorithms makes it difficult for server-side techniques to detect potential scripts within rich user-provided content. In contrast, any web browser knows perfectly well what content it considers a script. We exploit this insight by having web applications embed a *security hook* function in their pages that will be executed in a suitably-modified browser before executing any other script. When instantiated with a suitable server-provided whitelist or sandbox policy, the hook function can remove malicious scripts with perfect precision. We found that the required changes to web applications and browsers are small and localized, and performance overhead is low, making deployment practical. We plan to further explore the possibilities of BEEP, experimenting with additional policies and greater policy language support. Code, patches, and experimental data are available from our web site [10].

Acknowledgments. This work was supported in part by NSF grant CCF–0524036. We thank Nick Petroni and Jeff Foster for comments that helped us improve this paper; Jay McCarthy and Shriram Krishnamurthi their help with Continue; and Tamer Elsayed for providing last-minute battery power when one of our laptops' power supply failed.

8. REFERENCES

[1] Adbrite. `http://www.adbrite.com`.

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

[3] Eric Chien. Malicious Yahooligans. Virus Bulletin, August 2006.

[4] 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.

[5] Mona Gandhi, Markus Jakobsson, and Jacob Ratkiewicz. Badvertisements: Stealthy click-fraud with unwitting accessories. *Journal of Digital Forensic Practice*, 1(2), November 2006. Special Issue on Anti-Phishing and Online Fraud, Part I.

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

[7] O. Hallaraker and G. Vigna. Detecting malicious JavaScript code in Mozilla. In *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, June 2005.

[8] Hop home page. http://hop.inria.fr/.

[9] 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*, 2006.

[10] Trevor Jim, Nikhil Swamy, and Michael Hicks. BEEP: Browser-enforced embedded policies. http://www.research.att.com/~trevor/beep.html.

[11] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the Workshop on Programming Languages and Analysis for Security (PLAS)*, 2006.

[12] Paj's Home: Cryptography. http://www.pajhome.org.uk/crypt/index.html.

[13] 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*, 2006.

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

[15] Shriram Krishnamurthi. The CONTINUE server (or, how i administered PADL 2002 and 2003). In V. Dahl and P. Wadler, editors, *PADL*, volume 2562 of *Lecture Notes in Computer Science*. Springer, 2003.

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

[17] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(2):2–16, February 2005.

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

[19] Gervase Markham. Content restrictions. http://www.gerv.net/security/content-restrictions/, January 2006. Version 0.6.

[20] MITRE. Common vulnerabilities and exposures. http://cve.mitre.org.

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

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

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

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

[25] 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)*, 2006.

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

[27] Jesse Ruderman. Signed scripts in mozilla. http://www.mozilla.org/projects/security/components/signed-scripts.html.

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

[29] Optimizing page load time (and a little about the debug menu). http://webkit.org/blog/?p=75.

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

[31] Christian Schmidt. Comment on content restrictions proposal. http://weblogs.mozillazine.org/gerv/archives/007821.html, March 2005.

[32] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2006.

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

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

[35] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the USENIX Security Symposium*, 2006.

[36] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. Javascript instrumentation for browser security. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2007.