xJS: Practical XSS Prevention for Web Application Development

Elias Athanasopoulos, Vasilis Pappas, Antonis Krithinakis, Spyros Ligouras, Evangelos P. Markatos
Institute of Computer Science,
Foundation for Research and Technology - Hellas
email: `{elathan, vpappas, krithin, ligouras, markatos}@ics.forth.gr`
Thomas Karagiannis
Microsoft Research,
Cambridge, United Kingdom
email: `{thomas.karagiannis}@microsoft.com`

Abstract

We present `xJS`, a practical framework for preventing code-injections in the web environment and thus assisting for the development of XSS-free web applications. `xJS` aims on being fast, developer-friendly and providing backwards compatibility.

We implement and evaluate our solution in three leading web browsers and in the Apache web server. We show that our framework can successfully prevent all 1,380 real-world attacks that were collected from a well-known XSS attack repository. Furthermore, our framework imposes negligible computational overhead in both the server and the client side, and has no negative side-effects in the overall user's browsing experience.

1 Introduction

Code-injection attacks through Cross-Site Scripting (XSS) in the web browser have observed a significant increase over the previous years. According to a September-2009 report published by the SANS Institute [34], *attacks against web applications constitute more than 60% of the total attack attempts observed on the Internet. Web application vulnerabilities such as SQL injection and Cross-Site Scripting flaws in open-source as well as custom-built applications account for more than 80% of the vulnerabilities being discovered.* XSS threats are not only targeted towards relatively simple, small-business web sites, but also towards infrastructures that are managed and operated by leading IT vendors [2]. Moreover, recently widely adopted technologies, such as AJAX [15], exacerbate potential XSS vulnerabilities by promoting richer and more complex client-side interfaces. This added complexity in the web browser environment provides additional opportunities for further exploitation of XSS vulnerabilities.

Several studies have proposed mechanisms and architectures based on policies, communicated from the web server to the web browser, to mitigate XSS attacks. The current state of the art includes XSS mitigation schemes proposing whitelisting of legitimate scripts [17], utilizing randomized XML namespaces for applying trust classes in the DOM [16], or detecting code injections by examining modifications to a web document's original DOM structure [26]. While we believe that the aforementioned techniques are promising and in the right direction, they have weaknesses and they fail in a number of cases. As we show in this paper, whitelisting fails to protect from attacks that are based on already whitelisted scripts, while DOM-based solutions fail to protect from attacks where the DOM tree is absent [7].

To account for these weaknesses, in this paper, we propose `xJS`, which is a practical and simple framework that isolates legitimate client-side code from any possible code injection. Our contributions are thus twofold: i) we describe, implement and evaluate `xJS` and ii) we outline limitations of previous methodologies and a number of attacks that defeat existing approaches.

Our framework could be seen as a *fast randomization* technique. Instruction Set Randomization (ISR) [20] has been proposed for defending against code injections in native code or in other environments, such as code executed by databases [9]. However, we believe that adapting ISR to deal with XSS attacks is not trivial. This is because *web client-side code* is produced by the server and is executed in the client; the server lacks all needed functionality to manipulate the produced code. For example, randomizing the JavaScript instruction set in the web server requires at least one full JavaScript parser running at the server. Thus, instead of blindly implementing ISR for JavaScript, our design introduces *Isolation Operators*, which transpose all produced code in a new isolated domain. In our case, this is the domain defined by the `XOR` operator.

We design `xJS` with two main properties in mind:

- **Backwards Compatibility.** We aim for a practical, developer-friendly solution for constructing secure web applications and we ensure that the scheme

```
<div>                            <div>
<img onload=''render();''>       <img onload=''AlCtV...''>
<script>                         <script>
alert(''Hello World'');            vpSUlJTV2NHGwJyW/NHY...
<script>                         </script>
</div>                           </div>
```

Figure 1: Example of a web page that is generated by our framework.

provides backwards compatibility. `xJS` allows web servers to communicate to web browsers when the scheme is enabled or not. A web browser not supporting the framework may still render web applications, albeit without providing any of the security guarantees of `xJS`.

- **Low Computation Overhead**. Our design avoids the additional overhead of applying ISR in both web server and client, which would significantly increase the computational overheads. This is because the web code would be parsed twice (one in the server during serving and one in the client during execution). Instead, the isolation operator introduced in `xJS` applies the `XOR` function to the whole source corpus of all legitimate client-side code. Thus, the randomization process is fast, since XOR exists as a CPU instruction in all modern hardware platforms, and does not depend on any particular instruction set.

We implement and evaluate our solution in three leading web browsers namely FireFox, WebKit[1] and Chromium, and in the Apache web server.

Our evaluation shows that `xJS` can successfully prevent *all* 1,380 attacks of a well-known repository [14], imposes at the same time negligible computational overhead in the server and in the client side. Finally, our modifications appear to have no negative side-effects in the user web browsing experience. To examine user-perceived performance, we examine the behavior of `xJS`-enabled browsers through a leading JavaScript benchmark suite [3], which produces the same performance results in both the `xJS`-enabled and the original web browsers.

2 The xJS Framework

The fundamental concepts of our framework are *Isolation Operators* and *Action Based Policies* in the browser environment. We review each of these concepts in this section and, finally, we provide information about our implementation prototypes.

[1]WebKit is not a web browser itself, it is more like an application framework that provides a foundation upon which to build a web browser. We evaluate our modifications on WebKit using the Safari web browser.

`xJS` is a framework that can address XSS attacks carried out through JavaScript. However, our basic concept can be also applied to other client-side technologies, such as Adobe Flash. The basic properties of the proposed framework can be summarized in the following points.

- `xJS` prevents JavaScript code injections that are based on third party code or on code that is already used by the trusted web site.
- `xJS` prevents execution of trusted code during an event that is not scheduled for execution. Our framework guarantees that *only* the web site's code will be executed and *only* as the site's logic defines it.
- `xJS` allows for multiple trust-levels depending on desired policies. Thus, through `xJS`, parts of a web page may require elevated trust levels or further user authentication to be executed.
- `xJS` in principle prevents attacks that are based on injected data and misuse of the JavaScript `eval()` function. We discuss `eval()` semantics in detail in Sections 4 and 5.

Isolation Operators

`xJS` is based on Instruction Set Randomization (ISR), which has been applied to native code [20] and to SQL [9]. The basic concept behind ISR is to randomize the instruction set in such a way so that a code injection is not able to *speak the language of the environment* [21] and thus is not able to execute. In `xJS`, inspired by ISR, we introduce the concept of Isolation Operators (IO). An IO essentially transposes a source corpus to a new isolated domain. In order to de-isolate the source from the isolated domain a unique key is needed. This way, the whole source corpus, and not just the instruction set, is randomized.

Based on the above discussion, the basic operation of `xJS` is the following. We apply an IO such as the `XOR` function to effectively randomize and thus isolate all JavaScript source of a web page. The isolation is achieved since all code has been transposed to a new domain: the `XOR` domain. The IO is applied by the web server and all documents are served in their isolated form. To render the page, the web browser has to *de-isolate* the source by applying again the IO and then execute it.

Note that, in `xJS`, we follow the approach of randomizing the whole source corpus and not just the instruction set as in the basic ISR concept. We proceed with this choice since the web code is produced in the web server and it is executed in the web browser. In addition, the server lacks all needed functionality to manipulate the produced code. For example, randomizing the JavaScript instruction set needs at least one full

JavaScript parser running at the server. This can significantly increase the computational overhead and user-perceived latency, since the code would be parsed twice (one in the server during serving and one in the client during execution). However, the isolation can break web applications that explicitly evaluate dynamic JavaScript code using `eval()`. In that case, the web developer must use a new API, `xeval()`, since `xJS` alters the semantics of `eval()`. We further discuss this in Section 5. Finally, we select `XOR` as the IO because it is in general considered a fast process; all modern hardware platforms include a native implementation of the `XOR` function. However, our framework may be applied with any other IO.

Figure 1 depicts an `xJS` example. On the left, we show the source code as it exists in the web server and on the right, we provide the same source as it is fetched by the web browser. The JavaScript source has been `XOR`ed and a Base64 [18] encoding has been applied in order to transpose all non-printable characters to the printable ASCII range.

Action Based Policies

`xJS` allows for multiple trust-levels for the same web site depending on the desired operation. In general, our framework suggests that policies should be expressed as actions. Essentially, all trusted code should be treated using the policy "*de-isolate and execute*". For different trust levels, multiple IOs can be used or the same IO can be applied with a different key. For example, portions of client-side code can be marked with different trust levels. Each portion will be isolated using the `XOR` function, but with a different key. The keys are transmitted in HTTP headers (see the use of `X-IO-Key`, later in this section) every time the server sends the page to the browser.

Expressing the policies in terms of actions has the following benefit. The injected code cannot bypass the policy, unless it manages to produce the needed result after the action is applied to it. The latter is considered practically very hard, even for trivial actions such as the `XOR` operation. One possible direction for escaping the policy is using a brute force attack. However, if the key is large enough the probability to succeed is low.

Defining the desired policy set is out of the scope of this paper. For the purpose of our evaluation (see Section 4) we use one policy, which is expressed as "*de-isolate (apply XOR) and execute*". Other example policies can be expressed as "de-isolate and execute under user confirmation", "de-isolate with the X key and execute", etc.

2.1 Implementation

Browser Modifications. All three modified web browsers operate in the following way. A custom HTTP header field, `X-IO-Key`, is identified in each HTTP response. If the key is present, this is an indication that the web server supports the framework, and the field's value denotes the key for the de-isolation process. This is also a practical way for incremental deployment of the framework in a backwards compatible fashion. At the moment, we do not support multiple keys, but extending the browser with such a feature is considered trivial. On the other hand, the web browser communicates to the web server that it supports the framework using an `Accept`[2] header field for every HTTP request.

As far as WebKit and Chromium are concerned, we had to modify two separate functions. First, the function that handles all events (such as `onload`, `onclick`, etc.), and second, the function that evaluates a JavaScript code block. We modified these functions to (i) decode all source using Base64 and (ii) apply the `XOR` operation with the de-isolation key (the one transmitted in `X-IO-Key`) to each byte. FireFox has a different design. It also uses two functions, one for compiling a JavaScript function and one for compiling a script. However, these functions operate recursively. We further discuss this issue in Section 4.

Server Modifications. For the server part of `xJS` we are taking advantage of the modular architecture of the Apache web server. During Apache's start-up phase all configuration files are parsed and modules that are concerned with processing an HTTP request are loaded. The main processing unit of the apache web server is the content generator module. A module can register content generators by defining a handler that is configurable by using the `SetHandler` or `AddHandler` directives. These can be found in Apache's configuration file (httpd.conf).

Various request phases that precede the content generator exist. They are used to examine and possibly manipulate some request headers, or to determine how the request will be handled. For example the request URL will be matched against the configuration, because a certain content generator must be used. In addition the request URL may be mapped to a static file, a CGI script or a dynamic document according to the content generator's operation. Finally after the content generator has sent a reply to the browser, Apache logs the request.

Apache (from version 2 and above) also supports filters. Consider the filter chain as a data axis, orthogonal to the request processing axis. The request data may be processed by input filters before reaching the content generator. After the generator has finished generating the response various output filters may process it before being sent to the browser.We have created an

[2]For the definition of the `Accept` field in HTTP requests, see: `http://www.w3.org/Protocols/HTTP/HTRQ_Headers.html#z3`

Apache module which operates as a content generator. For every request, that corresponds to an HTML file in the disk, the file is fetched and processed by our module. The file is loaded in memory and stored in a buffer. The buffer is transfered to an HTML parser (based on the `HTMLParser` module from libxml2 [39]). This is an HTML 4.0 non-verifying parser with API compatible with the XML parser ones. When the parsing is done our module traverses the parser's XML nodes in memory and searches for all nodes that contain JavaScript (`<script>` nodes and events). If there is a match the XOR operation is applied using the isolation key to each byte of the JavaScript source. Finally all source is encoded in Base64.

After encoding all possible JavaScript source in the web page, the buffer is sent to the next operating module in the chain; this might be an output filter or the web browser. Implementing `xJS` as a content generator module has the benefit of isolating by encryption all JavaScript source before any dynamic content, which might include XSS attacks, is inserted. Our framework can cooperate with other server-side technologies, such as PHP, in two ways: (a) by using two Apache's servers (one running `xJS` and the other one the PHP module) and (b) by configuring PHP to run as a filter. All evaluation results presented in Section 4 are collected using the second setup.

Secret Key. The secret key that is used for the `XOR` operation is a string of random alphanumeric characters. The length of the string can be arbitrary. For all experiments presented in this paper a two-character string is used. Assuming that S_l is the JavaScript source of length l and K_L is the secret key of length L, the encoding works as follows: $Enc(S_i) = S_i \oplus K_{(i \,\%\, L)}, 0 < i < l$. It is implied that the ASCII values of the characters are used. The secret key is refreshed per request. We do not consider Man-in-the-Middle (MiM) attacks, since during a MiM an attacker can alter the whole JavaScript source without the need of an injection through XSS.

3 Attacks Covered

In this section we present a new form of XSS attack, which we refer to as *return-to-JavaScript* attack, in analogy with the *return-to-libc* attack in native code. This kind of XSS attack can escape script whitelisting, used by existing XSS mitigation schemes. We further highlight some important issues for DOM-based XSS mitigation schemes. All the attacks listed in this section can be successfully prevented by `xJS`.

3.1 *return-to-JavaScript* Attacks

A practical mitigation scheme for XSS attacks is script whitelisting, proposed in BEEP[17]. BEEP works as follows. The web application includes a list of cryptographic hashes of valid (trusted) client-side scripts. The browser, using a *hook*, checks upon execution of a script if there is a cryptographic hash in the whitelist. If the hash is found, the script is considered trusted and executed by the browser. If not, the script is considered non-trusted and the policy defines whether the script may be rendered or not. *Script whitelisting is not sufficient.* Despite its novelty, we argue here that simple whitelisting may not prove to be a sufficient countermeasure against XSS attacks. To this end, consider the following.

Location of trusted scripts. As a first example, note that BEEP does not examine the script's location inside the web document. Consider the simple case where an attacker injects a trusted script, initially configured to run upon a user's click (using the `onclick` action), to be rendered upon document loading (using the `onload`[3] action). In this case the script will be executed, since it is already whitelisted, but not as intended by the original design of the site; the script will be executed upon site loading and not following a user's click. If, for example, the script deletes data, then the data will be erased when the user's browser loads the web document and not when the user clicks on the associated hyperlink.

Exploiting legitimate whitelisted code. Attacks may be further carried out through legitimate white-listed code. XSS attacks are typically associated with injecting arbitrary client-side code in a web document, which is assumed to be foreign, i.e., not generated by the web server. However, it is possible to perform an XSS attack by placing code that *is* generated by the web server in different regions of the web page. This attack resembles the classic *return-to-libc* attack [11] in native code applications and thus we refer to as *return-to-JavaScript*. Return oriented programming suggests that an exploit may simply transfer execution to a place in `libc`[4], which may cause again execution of arbitrary code on behalf of the attacker. The difference with the traditional buffer overflow attack [29] is that the attacker has not injected any *foreign* code in the program. Instead, she transfers execution to a point that already hosts code that can assist her goal. A similar approach can be used by an attacker to escape whitelisting in the web environment. Instead of injecting her own code, she can take advantage of existing *whitelisted* code available in the web site. Note that, typically, a large fraction of client-side code is not

[3]One can argue that the `onload` action is limited and usually associated with the `<body>` tag. The latter is considered hard to be altered through a code-injection attack. However, note, that the `onload` event is also available for other elements (e.g. images, using the `<img>` tag) included in the web document.

[4]This can also happen with other libraries as well, but `libc` seems ideal since (a) it is linked to every program and (b) it supports operations like `system(), exec(), adduser()`, etc., which can be (ab)used accordingly. More interestingly, the attack can happen with no function calls but using available combinations of existing code [36].

executed upon document loading, but is triggered during user events, such as mouse clicks. Below we enumerate some possible scenarios for XSS attacks based on whitelisted code, which can produce (i) annoyance, (ii) data loss and (iii) complete takeover of a web site.

Annoyance. Assume the blog site shown in Figure 2. The blog contains a JavaScript function `logout()`, which is executed when the user clicks the corresponding hyperlink, *Logout* (line 4 in Fig. 2). An attacker could perform an XSS attack by placing a script that calls `logout()` when a blog entry is rendered (see line 7 in Fig. 2). Hence, a user reading the blog story will be forced to logout. In a similar fashion, a web site that uses JavaScript code to perform redirection (for example using `window.location.href = new-site`) can be also attacked by placing this whitelisted code in an `onload` event (see line 8 in Fig. 2).

Data Loss. A web site hosting user content that can be deleted using client-side code can be attacked by injecting the whitelisted deletion code in an `onload` event (see line 9 in Fig. 2). AJAX [15] interfaces are popular in social networks such as Facebook.com and MySpace.com. This attack can be considered similar to a SQL injection attack [5], since the attacker is implicitly granted access to the web site's database.

Complete Takeover. Theoretically, a web site that has a full featured AJAX interface can be completely taken over, since the attacker has all the functionality she needs a-priori whitelisted by the web server. For example, an e-banking site that uses a JavaScript `transact()` function for all the user transactions is vulnerable to XSS attacks that perform arbitrary transactions.

A workaround to mitigate the attacks presented above is to include the event type during the whitelisting process. Upon execution of script `S1`, which is triggered by an `onclick` event, the browser should check the whitelist for finding a hash key for `S1` *associated with an* `onclick` *event*. However, this can mitigate attacks which are based on using existing code with a different event type than the one initially intended to by the web programmer. Attacks may still happen. Consider the *Data Loss* scenario described above, where an attacker places the deletion code in `onclick` events associated with new web document's regions. The attacker achieves to execute legitimate code upon an event which is not initially scheduled. Although the attacker has not injected her own code, she manages to escape the web site's logic and associate legitimate code with other user actions. Attacks against whitelisting, based on injecting malicious data in whitelisted scripts, have been described in [26].

3.2 DOM-based Attacks

There is a number of proposals [16, 26, 13] against XSS attacks, which are based on information and features provided by DOM [24]. Every web document is rendered according to DOM, which represents essentially its esoteric structure. This structure can be utilized in order to detect or prevent XSS attacks. One of the most prominent and early published DOM-based techniques is DOM sandboxing, introduced originally in BEEP.

DOM sandboxing works as follows. The web server places all scripts inside `div` or `span` HTML elements that are attributed as *trusted*. The web browser, upon rendering, parses the DOM tree and executes client-side scripts only when they are contained in *trusted* DOM elements. All other scripts are marked as non-trusted and they are treated according to the policies defined by the web server. We discuss here in detail three major weaknesses of DOM sanbdoxing as an XSS mitigation scheme: (i) element annotation and (ii) DOM presence.

Element annotation. Enforcing selective execution in certain areas of a web page requires identification of those DOM elements that may host untrusted code or parts of the web application's code that inject unsafe content. This identification process is far from trivial, since the complexity of modern web pages is high, and web applications are nowadays composed of thousands lines of code. To support this, in Table 1 we highlight the number of `script`, `div` and `span` elements of a few representative web page samples. Such elements can be in the order of thousands in modern web pages. While there is active research to automate the process of marking untrusted data [35, 23] or to discover taint-style vulnerabilities [19, 25], we believe that, currently, the overhead of element annotation is prohibitive, and requires, at least partially, human intervention. On the contrary, `xJS` does not require taint-tracking or program analysis to identify trusted or untrusted parts of a web document or a web application.

	Facebook.com	MySpace.com	Digg.com
`script`	23	93	82
`div`	2708	264	302
`span`	982	91	156

Table 1: Element counts of popular home pages indicating their complexity.

DOM presence. All DOM-based solutions require the presence of a DOM tree. However, XSS attacks do not always require a DOM tree to take place. For example, consider an XSS attack which bypasses the content-sniffing algorithm of a browser and is *carried within* a PostScript file [7]. The attack will be launched when the file is previewed, and there is high probability that upon previewing there will be no DOM tree to surround the injected code. As browsers have been transformed to a generic preview tool, we believe that variants of this attack will manifest in the near future.

```
<html>
 <head> <title> Blog! </title> <head>
 <body>
  <a onclick="logout();">Logout</a>
  <div class="blog_entry" id="123"> {...} <input type="button" onclick="delete(123)"></div>
  <div class="blog_comments"> <ul>
   <li> <img onload="logout();" src="logo.gif">
   <li> <img onload="window.location.href='http://www.google.com';" src="logo.gif">
   <li> <img onload="delete(123);">
  </div>
  <a onclick="window.location.href='http://www.google.com';">Google</a>
 </body>
</html>
```

Figure 2: A minimal Blog site demonstrating the whitelisting attacks.

Another example is the unofficially termed *DOM-Based XSS* or *XSS of the Third Kind* attacks [22]. This XSS type alters the DOM tree of an already rendered page. The malicious XSS code does not interact with the sever in any way. In such an attack, the malicious code is embedded inside a URI after the fragment identifier. [5] This means that the malicious code (a) is not part of the initial DOM tree and (b) is never transmitted to the server. Unavoidably, DOM-based solutions [16, 26] that define trust classes in the DOM tree at server side will fail. The exploit will never reach the server and, thus, never be associated with or contained in a trust class.

3.3 Attacks Not Addressed

xJS aims on protecting against XSS attacks that are based on JavaScript injections. The framework is not designed for providing defenses against iframe injections and drive-by downloads [30], injections that are non-JavaScript based (for example, through arguments passed in Flash objects) and Phishing [12]. However, some fundamental concepts of xJS can be possibly applied to non-JavaScript injections.

4 Evaluation

In this section we evaluate the xJS prototype. Our evaluation seeks to answer four questions: (a) how many real XSS attacks can be prevented, (b) what the overhead on the server is, (c) what the overhead on the web browser is and, finally, (d) if the framework imposes any side-effects in the user's browsing experience.

4.1 Attack Coverage

We first evaluate the effectiveness of the xJS framework to prevent real-world XSS attacks. xJS aims on preventing traditional XSS attacks, as well as the XSS attacks described in Section 3.

Real-world exploits. To verify that xJS can cope with real-world XSS exploits, we use the repository hosted by XXSed.com [14] which includes a few thousands of XSS vulnerable web pages. This repository has been also used for evaluation in other papers [26]. The evaluation of the attack coverage through the repository is not a straightforward process. First, XSSed.com mirrors all vulnerable web pages with the XSS code embedded in their body. Some of them have been fixed after the publication of the vulnerability. These updated pages cannot be of use, since xJS prevents the code injection before it takes place and there is no way for us to have a copy of the original vulnerable web page (without the XSS code in its body). Second, we have no access to the vulnerable web server and, thus, we cannot use our server-side filter for the evaluation.

To address the aforementioned limitations, we conduct the evaluation as follows. First, we resolve all web sites that are still vulnerable. To this end, we download all 10,154 web pages listed in XSSed.com, along with their attack vectors. As the attack vector we define the URL along with the parameters that trigger the vulnerability.[6] Since XSS attacks that are based on a redirection without using any JavaScript cannot be addressed by xJS, we remove all such cases. Thus, we exclude 384 URLs that have an iframe as attack vector, 416 URLs that have a redirection to XSSed.com as attack vector and 60 URLs that have both an iframe and a redirection to XSSed.com as attack vector.

After this first pre-processing stage, the URL set contains all web pages that were vulnerable at some period in time and their vulnerability can be triggered using JavaScript; for example, the attack vector contains a call to the alert() function. We then exclude from the set all web-pages for which their vulnerability has been fixed after it became public in XSSed.com. To achieve this, we request each potentially vulnerable page through a custom proxy server we built using BeautifulSoup [31]. The task of the proxy is to attach some JavaScript code that overrides the alert() function with a URL request

[5]For more details about the fragment identifier, we refer the reader to http://www.w3.org/DesignIssues/Fragment.html.

[6]For example, consider the attack vector: http://www.needforspeed.com/undercover/home.action?lang=\"⟩⟨script⟩alert(document.cookie);⟨/script⟩&region=us

to a web server located in our local network. Since all attack vectors are based on the `alert()` function the web server recorded all successful attacks in its access logs. Using this methodology we manage to identify 1,381 web pages which are still vulnerable as of early September 2009. Our methodology suggests that about 1 in 9 web pages have not been fixed even after the vulnerability was published.

We use the remaining 1,381 pages as our final testing set. Since we cannot install our modified Apache in each of the vulnerable web sites, we use our proxy for simulating the server-side portion of `xJS`. More precisely, for each vulnerable page, we request the vulnerable document through our proxy with a slightly altered vector. For example, for the following attack vector,

```
http://site.com/page?
id=<script>alert("XSS");</script>
```

the proxy instead requests the URL,

```
http://site.com/page?
id=<xscript>alert("XSS");</xscript>.
```

Notice that the `script` tag has been modified to `xscript`. Using this methodology, we manage to build all vulnerable web pages with *the attack vector embedded but not in effect*. However, the JavaScript code contained in the web document is not isolated. Thus, the next step is to force the proxy to parse all web documents and apply the XOR function to the JavaScript code. At this point, all vulnerable web pages have the JavaScript code isolated and the attack vector defunct. Hence, the last step is to re-enable the attack vector by replacing the `xscript` with `script` and return the web page to the browser. All web pages also include some JavaScript code responsible for the `alert()` overloading. This code modifies all `alert()` calls to perform a web request to a web server hosted in our local network. If our web server records requests, the `alert()` function is called or, in other words, the XSS exploit run.

To summarize the above process, our experiment to evaluate the efficacy of the `xJS` framework is the following. We request each web page from the collected set which includes 1,381 still vulnerable web pages through a custom proxy that performs all actions described above. All web pages are requested using a modified Firefox. We select the modified Firefox in Linux, because it is easier to instrument through a script. We manually tested a random sample of attacks with modified versions of WebKit and Chromium and recorded identical behavior.

After Firefox has requested all 1,381 vulnerable pages through our custom proxy, we inspect our web server's logs to see if any of the XSS attacks succeeded. Our web server recorded just one attack. We carefully examined manually this particular attack and found out that it is a web page that has the XSS exploit stored inside its body and not in its attack vector [4]. The particular attack succeeded just as a side-effect of our evaluation methodology. If `xJS` were deployed in the vulnerable web server, this particular attack would also have been prevented. Hence, *all* 1,380 real-world XSS attacks were prevented successfully by our framework.

Attacks presented in Section 3. For the attacks presented in Section 3, since to our knowledge they have not been observed in the wild yet, we performed various custom attack scenarios using a popular web framework, Ruby on Rails [37]. We created a vulnerable blog and then installed the vulnerable blog service to a modified Apache server and browsed the blog using all three modified web browsers. As expected, in all cases, `xJS` succeeded in preventing the attacks.

We now look at specific attacks such as the ones based on a code injection in data and the careless use of `eval()`. The injected code is in plain text (non-isolated), but unfortunately it is attached to the isolated code after the de-isolation process. The injected code will be executed as if it is trusted. However, there is a way to prevent this. In fact, the internal design of Firefox gives us this feature with no extra cost. Firefox uses a `js_CompileScript()` function in order to compile JavaScript code. The design of this function is recursive and it is essentially the implementation of the actual `eval()` function of JavaScript. When Firefox identifies the script `eval($_GET('id'));`, de-isolates it, calls the `eval()` function, which in principle calls itself in order to execute the `$_GET('id')` part. At the second call, the `eval()` again de-isolates the `$_GET('id')` code, which is in plain text. The second de-isolation process fails and thus the code does not execute.

Our Firefox implementation can address this type of attack. WebKit and Chromium must be further modified to support this functionality. We have successfully implemented this process in Chromium after a small amount of code changes. However, this modification affects the semantics of `eval()`. For a more detailed discussion, please see Section 5.

4.2 Server Overhead

We now measure the overhead imposed on the server by `xJS`. To this end, we request a set of web pages that embed a significant amount of JavaScript. We choose to use the SunSpider suite [3] for this purpose. The SunSpider suite is a collection of JavaScript benchmarks that ship with WebKit and measure the performance of JavaScript engines. It is composed of nine different groups of programs that perform various complex operations. We manually select three JavaScript tests from the SunSpider suite. The *heavy* test involves string operations with many lines of JavaScript. This is probably the most

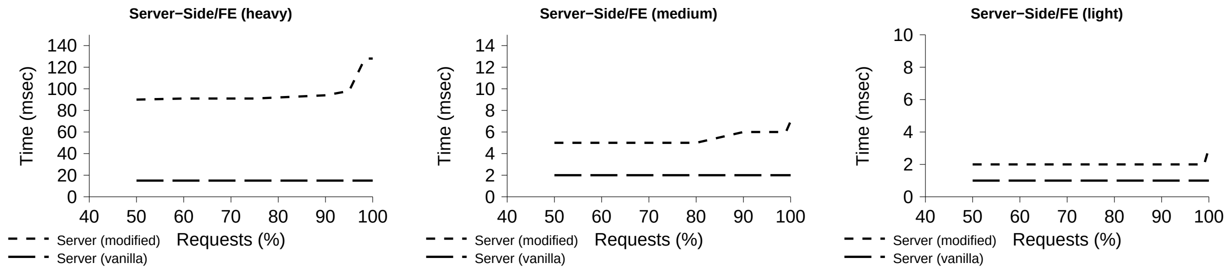

Figure 3: Server side evaluation when the Apache benchmark tool (`ab`) is requesting each web page through a Fast Ethernet link. In the worst case (heavy) the server imposes delay of a factor of five greater, while in the normal case the delay is only a few milliseconds.

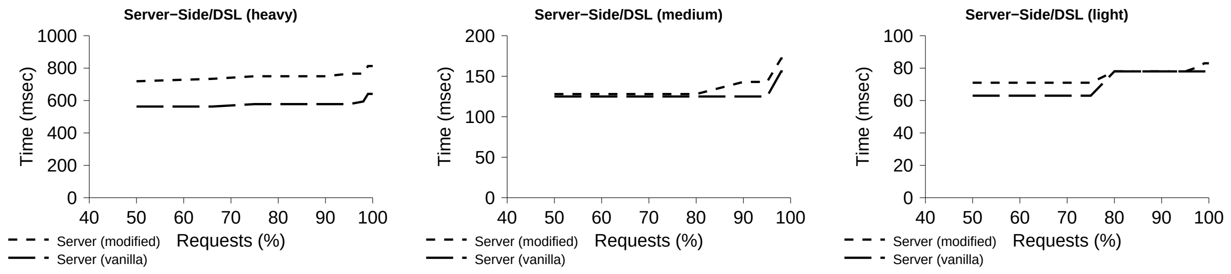

Figure 4: Server side evaluation when the Apache benchmark tool (`ab`) is requesting each web page through a DSL link. In the worst case (heavy) the server imposes a fixed delay of a few tens of milliseconds, like in the case of the Fast Ethernet setup (see Figure 3). However, this delay does not dominate the overall delivery time.

processing-intensive test in the whole suite, composed of many lines of code. The *normal* test includes a typical amount of source code like most other tests that are part of the suite. Finally, the *light* test includes only a few lines of JavaScript involving bit operations.

We conduct two sets of experiments. For the first set we use `ab` [1], which is considered the de-facto tool for benchmarking an Apache web server, over a Fast Ethernet (FE) network. We configure `ab` to issue 100 requests for the heavy, normal and light web pages, while the `xJS`module is enabled. Then, we perform the same experiments using the tests and with the `xJS` Apache module removed. We repeat all the above with the `ab` client running in a typical downstream DSL line (8 Mbps).

Figure 3 summarizes the results for the case of the `ab` tool connecting to the web server through a FE connection. The modified Apache imposes an overhead that ranges from a few (less than 6 ms and less than 2 ms for the normal and light test, respectively) to tens of milliseconds (about 60 ms) in the worst case (the heavy web page). While the results are quite promising for the majority of the tests, the processing time for the heavy page, which is over a factor of five greater, could be considered significant. In Figure 4 we present the same experiments over the DSL link. The overhead is still the same and it is negligible (less than a roundtrip in today's Internet) since now the delivery overhead dominates. This drives us to conclude that the Apache module imposes a fixed overhead of a few milliseconds per page, which is not the dominating overhead.

4.3 Client Overhead

Having examined the server-side overhead, we now measure the overhead imposed on the browser by `xJS`. We use the SunSpider test suite with 100 iterations, with every test executed 100 times. We use the `gettimeofday()` function to measure the execution time of the modified functions in each browser. Each implementation has two functions altered. The one that is responsible for handling code associated with events, such as `onclick`, `onload`, etc., and the one that is responsible for evaluating JavaScript code blocks. The modifications of WebKit and Chromium are quite similar (Chromium is based partially on WebKit). The modifications of Firefox are substantially different. In Firefox we have modified internally the JavaScript `eval()` function which is recursive. These differences affect the experimental results in the following way. In WebKit and Chromium we record fewer long calls in contrast with Firefox, in which we record many short calls.

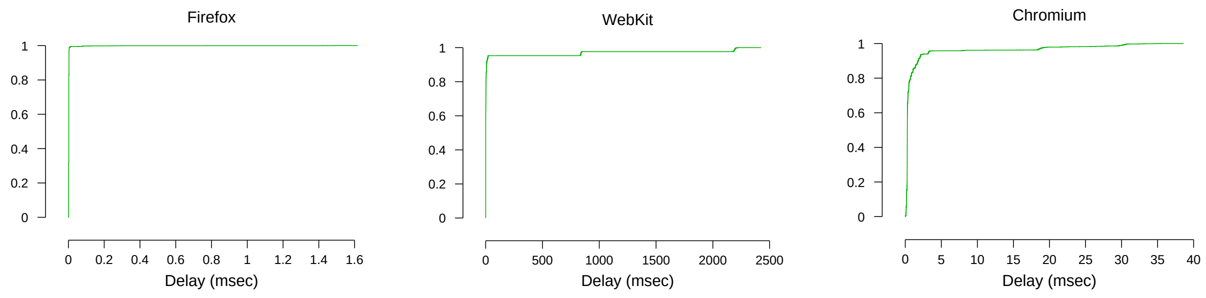

Figure 5: Cumulative distribution for the delay imposed by all modified function calls in the Firefox, WebKit and Chromium implementation, respectively. As delay we assume the time needed for the modified function to complete minus the time needed for the unmodified one to complete. Notice that the majority of function calls imposes a delay of a few milliseconds.

In Figure 5 we present the cumulative distribution of the delays imposed by all modified recorded function calls for Firefox, WebKit and Chromium, during a run of the SunSpider suite for 100 iterations. As delay we define the time needed for the modified function to complete minus the time needed for the unmodified one to complete. Observe that the Firefox implementation seems to be the faster one. All delays are less than 1 millisecond. However, recall that Firefox is using a lot of short calls, compared to the other two browsers. Firefox needs about 500,000 calls for the 100 iterations of the complete test suite. In Figure 5 we plot the first 5,000 calls for Firefox (these calls correspond to one iteration only) of the complete set of about 500,000 calls, for visualization purposes and to facilitate comparison, and all 4,800 calls needed for WebKit and Chromium to complete the test suite, respectively. Chromium imposes an overhead of a few milliseconds per call, while WebKit seems to impose larger overheads. Still, the majority of WebKit's calls impose an overhead of a few tens of milliseconds.

4.4 User Browsing Experience

We now identify whether user's browsing experience changes due to xJS. As user browsing experience we define the performance of the browser's JavaScript engine (i.e., running time), which would reflect the user-perceived rendering time (as far as the JavaScript content is concerned) for the page. We run the SunSpider suite *as-is* for 100 iterations with all three modified web browsers and with the equivalent unmodified ones and record the output of the benchmark. In Figure 6 we plot the results for all different categories of tests. Each category includes a few individual benchmark tests. As expected there is no difference between a modified and a non modified web browser for all three platforms, Firefox, WebKit and Chromium. This result is reasonable, since after the de-isolation process the whole JavaScript source executes normally as it is in the case with a non compatible with the xJS framework web browser. Moreover, this experiment shows that xJS is not restrictive with legitimate web sites, since all the SunSpider suite (some thousands of JavaScript LoCs) run without any problem or side-effect.

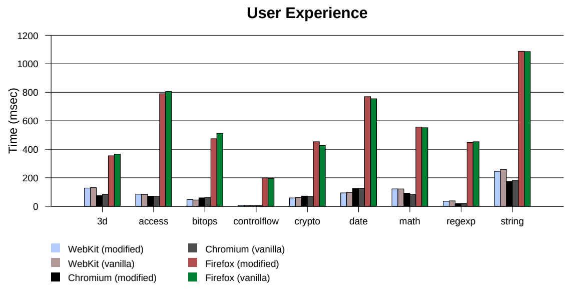

Figure 6: Results from the SunSpider test suite. Notice that for each modified browser the results are comparable with the results of its unmodified equivalent. That is, all de-isolated JavaScript executes as expected in both modified and unmodified browser.

5 Discussion

We now discuss potential limitations of our approach and offer possible workarounds.

JavaScript Obfuscation. Web pages served by xJS have all JavaScript encoded in Base64. Depending on the context this may be considered as a feature or not. For example, there are plenty of custom tools that obfuscate JavaScript on purpose. Such tools are used by certain web sites for protecting their JavaScript code and prevent visitors from copying the code. We should make clear that emitting all JavaScript encoded does not harden the development process, since all JavaScript manipulation takes place during serving time. While debugging, web developers may safely switch off `xJS`. Blueprint [38] also emits parts of a web page in Base64.

eval() Semantics and Dynamic Code. As previously discussed (see Section 4), in order for `xJS` to cope with XSS attacks that are based on malicious injected data, the semantics of `eval()` must change. More precisely, our Firefox modifications alter the `eval()` function in the following way. Instead of simply evaluating a JavaScript content, the modified `eval()` function performs de-isolation before evaluation. This behavior can break web applications that are based on the generation of dynamic JavaScript code, which is executed using `eval()` at serving time. While this type of programming might be considered inefficient and error-prone, we suggest the following workaround. The JavaScript engine can be enhanced with an `xeval()` variant which does not perform any de-isolation before evaluation. The web programmer must explicitly call `xeval()` if this is the desired behavior. Still, there is no possibility for the attacker to evaluate her code (using `xeval()`), since the original call to `xeval()` must be already isolated.

Code Templates and Persistent XSS. Web developers frequently use templates in order to produce the final web pages. These templates are stored usually in a database and sometimes they include JavaScript. The database may also contain data produced by user inputs. In such cases, the code injection may take place *within* the database (Persistent XSS). This may occur if trusted code and a user input containing malicious code are merged together before included in the final web page. This case is especially hard to track, since it involves the programmer's logic to a great extent. The challenge lies in that client-side code is hosted in another environment (the database) which is also vulnerable to code injections. `xJS` assumes that all trusted JavaScript is stored in files and not in a database. If the web developer wishes to store legitimate JavaScript in a database then she can place it in read-only tables. With these assumptions, `xJS` can cope with persistent XSS. Recall from Section 2 that `xJS` module is the first to run in the Apache module chain and, thus, all JavaScript isolation will take place before any content is fetched from databases or other external sources.

6 Related Work

The closest studies to `xJS` are BEEP [17], Noncespaces [16] and DSI [26]. Throughout the paper, we have highlighted certain cases where the aforementioned methodologies fail (e.g., see Section 3). We have presented attacks that escape whitelisting (proposed in [17]) and cases where DOM-based solutions [16, 26] are not efficient. Our framework, `xJS`, can cope with XSS attacks that escape whitelisting [6], and does not require any information related to DOM; `xJS` can also prevent attacks that leverage the content-sniffing algorithms of web browsers [7].

Our technique is based on Isolation Operators and it is inspired by Instruction Set Randomization (ISR) [20]. Solutions based on ISR have been applied to native code and to SQL injections [9]. Keromytis discusses ISR as a generic methodology for countering code injections in [21] and he mentions that the technique can be potentially applied in XSS mitigation. However, to the best of our knowledge there has been no systematic effort towards this approach before.

In [40] the authors propose to use dynamic tainting analysis to prevent XSS attacks. Taint-tracking has been partially or fully used in other similar approaches [26, 35, 28, 27]. Although `xJS` does not rely at all on tainting, a source-code based tainting technique [43] can certainly assist in separating all server-produced JavaScript. The server side of `xJS` will be able to efficiently mark all legitimate client-side code and also identify malicious data. However, the performance might degrade.

Blueprint [38] is a server-only approach which guarantees that untrusted content is not executed. The application server pre-renders the page and serves each web document in a form in which all dynamic content is correctly escaped to avoid possible code injections. However, Blueprint requires the web programmer to inject possible unsafe content (for example comments of a blog story) using a specific Blueprint API in PHP. Spotting all unsafe code fragments of a web application is not trivial. Blueprint imposes further a significant overhead compared to solutions based on natively browser modifications, like `xJS`.

Enforcing separation between structure and content is another prevention scheme for code injections [32]. This proposed framework can deal with XSS attacks as well as SQL injections. As far as XSS is concerned, the basic idea is that each web document has a well defined structure in contrast to a stream of bytes, as it is served nowadays by web servers. This allows the authors to enforce a separation between the authentic document's structure and the untrusted dynamic content from user input, which is attached to it. However, in contrast to `xJS`, this technique cannot deal with attacks that are based on the content-sniffing algorithms of browsers [7] as well as attacks that modify the DOM structure using purely client-side code [22].

Script accenting [10] is based also on `XOR` for isolating scripts in the web browser. Script accenting aims on providing an efficient mechanism for implementing the *same origin policy* in a web browser, but it is not explicitly related with XSS. Leaks that can take place due to the DOM separation from the JavaScript engine, inside a web browser, and can lead to browser compromising have been studied in [8]. These attacks can be considered more severe than XSS and cannot be captured by `xJS`. MashupOS [41] and subsequent work Gazelle [42] view browsers as a multi-principal OS where a principal is labeled by a web site's origin following the *same-origin policy* [33]. MashupOS analyzed and proposed protection and communications abstractions that a browser should expose for web application developers. In particular, `<sandbox>` is proposed to embed untrusted content and can be used by developers to prevent XSS attacks as long as they can correctly differentiate trusted content from untrusted ones. In comparison, our work does not require explicit inclusion of untrusted content from developers.

7 Conclusion

In this paper we present `xJS`, a practical and developer-friendly framework against the increasing threat of XSS attacks. The motivation for developing `xJS` is twofold. First, we want an XSS prevention scheme that can cope with the new *return-to-JavaScript* attacks presented in this paper and second, we want the solution to be easily adopted by web developers.

We implement and evaluate our solution in three leading web browsers and in the Apache web server. Our evaluation shows that (a) every examined real-world XSS attack can be successfully prevented, (b) negligible computational overhead is imposed on the server and browser side, and (c) the user's browsing experience and perceived performance is not affected by our modifications.

Acknowledgements

We would like to thank the anonymous reviewers for their feedback, Professor Angelos D. Keromytis for early discussions and our shepherd Helen Wang. Her assistance for improving the paper was invaluable. Elias Athanasopoulos, Antonis Krithinakis, Spyros Lygouras and Evangelos P. Markatos are also with the University of Crete. Elias Athanasopoulos is funded by the Microsoft Research PhD Scholarship project, which is provided by Microsoft Research Cambridge.

References

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

[2] McAfee: Enabling Malware Distribution and Fraud. `http://www.readwriteweb.com/archives/mcafee_enabling_malware_distribution_and_fraud.php`.

[3] SunSpider JavaScript benchmark. `http://www2.webkit.org/perf/sunspider-0.9/sunspider.html`.

[4] XXSed.com vulnerability 35059. `http://www.xssed.com/mirror/35059/`.

[5] C. Anley. Advanced SQL injection in SQL server applications. *White paper, Next Generation Security Software Ltd*, 2002.

[6] E. Athanasopoulos, V. Pappas, and E. Markatos. Code-Injection Attacks in Browsers Supporting Policies. In *Proceedings of the 2nd Workshop on Web 2.0 Security & Privacy (W2SP)*, Oakland, CA, May 2009.

[7] A. Barth, J. Caballero, and D. Song. Secure Content Sniffing for Web Browsers or How to Stop Papers from Reviewing Themselves. In *Proceedings of the 30th IEEE Symposium on Security & Privacy*, Oakland, CA, May 2009.

[8] A. Barth, J. Weinberger, and D. Song. Cross-Origin JavaScript Capability Leaks: Detection, Exploitation, and Defense. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, Quebec, August 2009.

[9] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, pages 292–302, 2004.

[10] S. Chen, D. Ross, and Y.-M. Wang. An Analysis of Browser Domain-Isolation Bugs and a Light-Weight Transparent Defense Mechanism. In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS)*, pages 2–11, New York, NY, USA, 2007. ACM.

[11] S. Designer. Return-to-libc attack. *Bugtraq, Aug*, 1997.

[12] R. Dhamija, J. Tygar, and M. Hearst. Why Phishing Works. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 581–590. ACM New York, NY, USA, 2006.

[13] A. Felt, P. Hooimeijer, D. Evans, and W. Weimer. Talking to Strangers Without Taking their Candy: Isolating Proxied Content. In *SocialNets '08: Proceedings of the 1st Workshop on Social Network Systems*, pages 25–30, New York, NY, USA, 2008. ACM.

[14] K. Fernandez and D. Pagkalos. XSSed.com. XSS (Cross-Site Scripting) information and vulnerable websites archive. `http://www.xssed.com`.

[15] J. Garrett et al. Ajax: A New Approach to Web Applications. *Adaptive path*, 18, 2005.

[16] M. V. Gundy and H. Chen. Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 8-11, 2009.

[17] T. Jim, N. Swamy, and M. Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 601–610, New York, NY, USA, 2007. ACM.

[18] S. Josefsson. RFC 4648: The Base16, Base32, and Base64 Data Encodings, 2006. http://tools.ietf.org/html/rfc4648.

[19] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *Proceedings of the 27th IEEE Symposium on Security and Privacy*, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

[20] G. Kc, A. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks with Instruction-Set Randomization. In *Proceedings of the 10th ACM conference on Computer and Communications Security*, pages 272–280. ACM New York, NY, USA, 2003.

[21] A. D. Keromytis. Randomized Instruction Sets and Runtime Environments Past Research and Future Directions. Number 1, pages 18–25, Piscataway, NJ, USA, 2009. IEEE Educational Activities Department.

[22] A. Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. Web Application Security Consortium, Articles, 4.7. 2005.

[23] L. C. Lam and T.-c. Chiueh. A General Dynamic Information Flow Tracking Framework for Security Applications. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 463–472, Washington, DC, USA, 2006. IEEE Computer Society.

[24] A. Le Hors, P. Le Hegaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document Object Model (DOM) Level 3 Core Specification. *World Wide Web Consortium, Recommendation REC-DOM-Level-3-Core-20040407*, 2004.

[25] M. Martin and M. S. Lam. Automatic Generation of XSS and SQL Injection Attacks with Goal-directed Model Checking. In *Proceedings of the 17th USENIX Security symposium*, pages 31–43, Berkeley, CA, USA, 2008. USENIX Association.

[26] Y. Nadji, P. Saxena, and D. Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 8-11, 2009.

[27] S. Nanda, L. Lam, and T. Chiueh. Dynamic Multi-Process Information Flow Tracking for Web Application Security. In *Proceedings of the 8th ACM/IFIP/USENIX international conference on Middleware*. ACM New York, NY, USA, 2007.

[28] A. Nguyen-tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of the 20th IFIP International Information Security Conference*, pages 372–382, 2005.

[29] A. One. Smashing the stack for fun and profit. *Phrack magazine*, 49(7), 1996.

[30] N. Provos, P. Mavrommatis, M. Rajab, and F. Monrose. All your iFRAMES point to us. In *Proceedings of the 17th conference on Security symposium*, pages 1–15. USENIX Association, 2008.

[31] L. Richardson. Beautiful Soup-HTML/XML parser for Python, 2008.

[32] W. Robertson and G. Vigna. Static Enforcement of Web Application Integrity Through Strong Typing. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, Quebec, August 2009.

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

[34] SANS Insitute. The Top Cyber Security Risks. September 2009. http://www.sans.org/top-cyber-security-risks/.

[35] R. Sekar. An Efficient Black-box Technique for Defeating Web Application Attacks. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 8-11, 2009.

[36] H. Shacham. The Geometry of Innocent Flesh on the Bone: return-into-libc without Function Calls (on the x86). In *CCS '07: Proceedings of the 14th ACM conference on Computer and Communications Security*, pages 552–561, New York, NY, USA, 2007. ACM.

[37] B. Tate and C. Hibbs. *Ruby on Rails: Up and Running*. O'Reilly Media, Inc., 2006.

[38] M. Ter Louw and V. Venkatakrishnan. Blueprint: Precise Browser-neutral Prevention of Cross-site Scripting Attacks. In *Proceedings of the 30th IEEE Symposium on Security & Privacy*, Oakland, CA, May 2009.

[39] D. Veillard. Libxml2 project web page. *http://xmlsoft. org*, 2004.

[40] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceeding of the 14th Annual Network and Distributed System Security Symposium (NDSS)*, 2007.

[41] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and Communication Abstractions for Web Browsers in MashupOS. In T. C. Bressoud and M. F. Kaashoek, editors, *SOSP*, pages 1–16. ACM, 2007.

[42] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The Multi-Principal OS Construction of the Gazelle Web Browser. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, Canada, August 2009.

[43] W. Xu, E. Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the 15th USENIX Security Symposium*, pages 121–136, 2006.