⊕JS: Lightweight Cross-Site Scripting Prevention Using Isolation Operators

Elias Athanasopoulos, Vasilis Pappas, Spyros Ligouras, Evangelos P. Markatos
Institute of Computer Science,
Foundation for Research and Technology - Hellas
email: {elathan, vpappas, ligouras, markatos}@ics.forth.gr

Thomas Karagiannis
Microsoft Research,
Cambridge, United Kingdom
email: {thomas.karagiannis}@microsoft.com

TR-400, October 2009
FORTH-ICS

Abstract

Cross-site scripting (XSS) attacks constitute one of the major threats for today's web sites. Recently reported numbers on XSS vulnerabilities, coupled with the increasing complexity of modern web browsers, clearly highlight the need for effective mitigation mechanisms. However, despite the significance of these attacks, a definitive approach against any type of XSS vulnerability sill remains elusive. To further highlight this absence of effective countermeasures, we present a new family of code-injection attacks that defeat existing approaches for XSS prevention. The identified attacks resemble the classic *return-to-libc* attack in native code.

To account for the detected vulnerabilities, we proceed and present a fast and practical mechanism, namely xJS, that isolates all legitimate client-side code from possible code injections. xJS is a lightweight mechanism that is based on the concept of Instruction Set Randomization (ISR). We implement and evaluate our solution in three leading web browsers, namely FireFox, WebKit and Chromium, 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 [44], *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 [19], 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.

XSS attacks typically refer to injecting client-side code, usually JavaScript, in a web document. The injection can take a number of forms and could among other things be performed by content submission; for example, code injection can take place by a user posting a comment with embedded JavaScript in a blog. This results in all browsers rendering the submitted content to execute the attacker's JavaScript. The malicious code may have various effects that range from stealing user cookies to arbitrary operations resulting in compromising users' identity or data. We would like to stress that while XSS is typically associated with stealing cookies, there are certain cases where the attacker's goal is different. For example, consider the case where the attacker injects some client-side code which deletes or modifies data from a user's profile.

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 [22], utilizing randomized XML namespaces for applying trust classes in the DOM [20], or detecting code injections by examining modifications to a web document's original DOM structure [36]. 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 or when the attack code never reaches the server [30].

To account for these weaknesses, in this paper, we propose $\oplus$JS (or simpler, 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 outline limitations of previous methodologies and a number of attacks that defeat existing approaches, and ii) we describe, implement and evaluate xJS.

We first examine potential weaknesses in previous methodologies, and we illustrate a number of attacks that defeat policy based mechanisms like the one proposed by BEEP [22]. This family of attacks resembles the classic *return-to-libc* attack in native code [15]. Further, we highlight all major weaknesses in preventing XSS attacks using information encapsulated in the DOM structure [20, 36].

To address these limitations, we design the xJS framework which is inspired mainly by the concept of Instruction Set Randomization (ISR) [27]. The foundations of xJS lie in the use of *Isolation Operators* in order to randomize the whole source corpus of client-side code and in policies expressed as *Browser Actions*. Our framework targets XSS attacks carried through JavaScript, and guarantees that all trusted client-side code can be successfully isolated from possibly untrusted code.

Our framework could be seen as a *fast randomization* technique. ISR has been proposed for defending against code injections in native code or in other environments, such as code executed by databases [14]. 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: i) ease of deployment, and ii) low computation overhead. In particular, xJS is characterized by:

- **Ease of deployment.** We wish to have a practical and simple to implement and deploy solution. This is crucial if the proposed framework is to be adopted. As is typical in many approaches for XSS mitigation, our solution needs co-operation from both sides, web server and client. However, the xJS implementation is extremely lightweight and simple, with no more than 100 lines of code for each of three major web browsers (Firefox, WebKit and Chromium). Similarly, the server-side implementation is a fairly straightforward process. We discuss the architecture and implementation

of xJS in detail in Section 3.

- **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 covers three different perspectives: i) ability of the proposed framework to successfully prevent attacks, ii) computational overheads at the client and the server, and iii) impact on user web browsing experience. To evaluate attack mitigation, we use real-world attacks collected from an XSS attack repository [18]. Our evaluation shows that xJS can successfully prevent *all* 1,380 attacks of the repository, 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 [6], which produces the same performance results in both the xJS-enabled and the original web browsers.

Our contributions can be summarized in the following points:

- We outline a series of limitations and vulnerabilities of previously described methodologies to mitigate XSS attacks (Section 2).
- We describe the design, architecture and implementation of xJS, a practical and simple to implement framework to prevent XSS attacks (Section 3).
- We evaluate xJS 's performance and computational overheads in real-world documented attacks (Section 4).

2 Threat Model

In this section, we present the threat model we address in this paper. First, we provide a short introduction to XSS attacks. We then describe XSS mitigation practices suggested in previous works to address such attacks, and in particular BEEP [22] that proposed *script whitelisting*. Despite the novelty of BEEP, we highlight a series of new XSS attacks that may escape from script whitelisting. A preliminary report on these kind of attacks can be found in [10]. We further proceed and discuss approaches for XSS prevention that rely on the DOM structure of a web document [20, 36], such as *DOM sandboxing*, and highlight how these approaches can sometimes fail.

2.1 XSS Overview

An XSS attack is a typical code injection performed in web applications. The attacker aims on injecting her client-side code into a web document, which will eventually render in the victim's web browser. Upon rendering, the malicious code may steal information from the user's browser environment or force the user's browser to perform specific actions. Most XSS attacks are carried out using JavaScript, although other client-side technologies can be also used. A typical example of an XSS attack is the following. Consider an attacker submitting a comment, which embeds malicious JavaScript code, to a web blog. All other users visiting the blog and viewing the comments of the story will host the attacker's JavaScript code in their browsers. The malicious code may issue a request to the attacker's web server with a URI that embeds the user's cookie. This URI may have the form:

[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 evaluated our modifications on WebKit using the Safari web browser.

```
http://www.attacker.com/page?{document.cookie}
```

The attacker may thus collect all cookies from users viewing the specific web blog. The adversary can then hijack users' sessions, since frequently a web cookie contains information for user authentication.

Although traditionally XSS attacks have been associated with stealing cookies, the attack itself has a broader target. As we argue in this paper, plenty of modern web sites have employed rich, user-driven AJAX interfaces and most of their provided operations are carried out through client-side code. In addition, there are efforts for client-side toolkits [5]. Attackers targeting this kind of web sites can benefit from the richness of client-side code and thus create XSS exploits which perform arbitrary operations through a victim's web browser. These operations may lead to a number of consequences, from annoyance and data loss to even a complete takeover of a user's profile, depending on the nature of the vulnerable web site. We discuss such proof-of-concept attacks later in this section.

2.2 Script Whitelisting and its Vulnerabilities

A practical mitigation scheme for XSS attacks is script whitelisting, proposed in BEEP[22]. 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 places 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`[2] 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 whitelisted 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 [15] in native code applications. Return oriented programming suggests that an exploit may simply transfer execution to a place in `libc`[3], which may cause again execution of arbitrary code on behalf of the attacker. The difference with the traditional buffer overflow attack [40] 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 executed upon document loading, but is triggered during user events, such as mouse clicks or mouse movements. 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.

[2] 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.

[3] 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 [46].

```
1: <html>
2:  <head> <title> Blog! </title> <head>
3:  <body>
4:   <a onclick="logout();">Logout</a>
5:   <div class="blog_entry" id="123"> {...} <input type="button" onclick="delete(123)"></div>
6:   <div class="blog_comments"> <ul>
7:    <li> <img onload="logout();" src="logo.gif">
8:    <li> <img onload="window.location.href='http://www.google.com';" src="logo.gif">
9:    <li> <img onload="delete(123);">
10:  </div>
11:  <a onclick="window.location.href='http://www.google.com';">Google</a>
12: </body>
13:</html>
```

Figure 1: A minimal Blog site demonstrating the whitelisting attacks.

- *Annoyance.* Assume the blog site shown in Figure 1. The blog contains a JavaScript function `logout()`, which is executed when the user clicks the corresponding hyperlink, *Logout* (line 4 in Fig. 1). 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. 1). 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. 1).
- *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. 1). AJAX [19] interfaces providing such functionality are popular in social networks such as Facebook.com and MySpace.com. This attack can be considered similar to a SQL injection attack [9], 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 quick workaround to mitigate the attacks presented above is to include the event type during the whitelisting process. For example, 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 only 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. Another form of attacks against whitelisting, based on injecting malicious data in whitelisted scripts, has been described in [36].

2.3 DOM-based Techniques

There is a number of proposals [20, 36, 17] against XSS attacks, which are based on information and features provided by DOM [32]. 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*. For example, consider the construct:

```
<div class='trusted'>
  <script> ... </script>
</div>
```

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) node splitting, (ii) element annotation and (iii) DOM presence.

Node splitting. An attacker can inject a malicious script surrounded, on purpose, by misplaced HTML tags in order to escape from a particular DOM node. Consider for example the construct:

```
<i>{ message }<\i>
```

which, denotes that a message should be rendered in *italic* style. If the message variable is filled in with:

```
</i><b> bold message </b><i>
```

then the carefully placed `<i>` and `<b>` tags result the message to be displayed in **bold** style, rather than *italic*.

The authors of BEEP suggest a workaround for dealing with node-splitting. All the data inside an untrusted `div` must be placed using a special coding idiom in JavaScript. A more elegant approach is to randomize the DOM elements [20, 36]. The attacker then needs to know a key in order to escape from a node. Consider the example:

```
<nonce42:div> ... </nonce42:div>
```

In this case the key is 'nonce42'. As long as the key is complex enough to be predicted through a brute force attack, this technique is considered sufficient. However, the only published implementation so far, Noncespaces, is based on XHTML [41] which is a strict HTML dialect. XHTML inherits many properties from XML. An XHTML document must comply to various criterions of validness in order to be rendered by a browser. Noncespaces is based on this strictness to identify potential attacks. Any code-injection attempt produces a non-valid document, which is rejected by the browser (i.e., the page is not rendered). An attacker may exploit this strictness by injecting malformed HTML code on purpose, invalidating this way the target web pages. A simple example of such an attack could be the case of a malicious user posting malformed HTML messages in users' profiles of a social network.

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 by 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 [45, 31] or to discover taint-style vulnerabilities [26, 34], 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.

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 [11]. 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.

	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.

```
1: <?php
2:  $s = "<div id='malicious'>" . $_GET["id"] . "</div>";
3:  echo $s;
4: ?>
5: <script>
6:  eval(document.getElementById('malicious').innerHTML);
7: </script>
```

Figure 2: XSS example based on the injection of malicious data.

Another example is the unofficially termed *DOM-Based XSS* or *XSS of the Third Kind* attacks [30]. This XSS flavor alters the DOM tree of an already rendered page, without the malicious code come in any contact with the server. In such an attack, the malicious code is embedded inside a URI after the fragment identifier. [4] 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 [20, 36] that define trust classes in the DOM tree at server side will fail. The malicious code will never reach the server and, thus, never be associated with or contained in a trust class.

3 The xJS Framework

Having presented the various weaknesses of previous methodologies in the previous section, we now describe in detail the `xJS` framework for preventing XSS attacks. The fundamental concept of our framework is the *Isolation Operators*. To achieve practical deployment, we further propose *Code Separation* for client-side code and *Action Based Policies* in the browser environment. We review each of these three concepts in this section and, finally, we provide information about our implementation prototypes which include three leading web browsers (Firefox, WebKit and Chromium) and the Apache web server.

`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. Consider the example in Figure 2. If an attacker inserts JavaScript code in the `id` field of the `GET` request, then the code will be executed. The above document is vulnerable, because the `eval()` function is used carelessly. One way to prevent this kind of code injection is by using tainting [38, 36].

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

Our framework can be augmented to prevent such attacks using tainting or by modifying the `eval()` function, as it happens in our `xJS` implementation for Firefox. 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 [27] and to SQL [14]. 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* [28] 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).

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.

Code Separation

Traditionally, one thinks of web code in terms of server-side and client-side code. The server-side part is usually written in a scripting language (PHP, Ruby, Python, Perl, etc.), or even in native code, that is pre-processed by the server. The rest of the web code is considered as client-side code and it is evaluated in the web browser. The web server can pre-process the server-side code by looking for specific delimiters. For example, a PHP code fraction is enclosed in `<?php` and `?>`.

Our framework suggests that the web code separation should span across three domains: *server-side*, *client-side* and *markup*. More precisely, in our prototype implementation we use a pre-processor to filter all trusted JavaScript code, which is enclosed in the `< < < <` and `> > > >` delimiters. In addition, we use the "+=" operator for all asynchronous events such as `onclick` and `onload`, which are associated with JavaScript code.

Figure 3 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 [24] encoding has been applied in order to transpose all non-printable characters to the printable ASCII range.

Note that `xJS` may be applied without the introduction of any new special delimiters. In fact, we have also implemented a version of `xJS` which does not depend on any special delimiters. Instead, an HTML parser pre-processes every web page, scans for all `script` tags and asynchronous events associated with JavaScript, and applies the `XOR` operator.

However, we believe that introducing special delimiters is preferable. Using delimiters for client-side code assists in a *clean* separation schema distinguishing the three fundamental classes of a web document,

```
1:  <div>                            1:  <div>
2:    <img onload+=''render();''>    2:    <img onload=''AlCtV2NH...''>
3:    <<<<                           3:    <script>
4:    alert(''Hello World'');        4:      vpSUlJTV2NHGwJyW/NHY...
5:    >>>>                           5:    </script>
6:  </div>                           6:  </div>
```

Figure 3: Example of a web page that is generated by our framework. On the left, the figure shows the source code as exists in the web server and on the right the same source as it is fetched by the web browser. The JavaScript source has been XORed and a Base64 encoding has been applied in order to transpose all non-printable characters to the printable ASCII range.

namely (i) server-side code, (ii) client-side code and (iii) markup. This kind of code separation also assists significantly in possible manipulations of the whole client-side code corpus, since applying an IO to all produced JavaScript is trivial. On the contrary, separating all client-side code using special delimiters requires the change of current programming disciplines. We further discuss this issue in Section 5.

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 header, later in this section) every time the server sends the web page to the web 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 complex enough the probability of a brute force attack 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*".

3.1 Implementation

We implemented xJS by modifying three leading web browsers, namely Firefox, WebKit and Chromium. We also created a filter for the Apache web server.

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. 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[5] 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

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

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.

Implementing the xJS prototype took roughly a few days for the three browsers and no more than 100 lines of code per implementation. Since we lack comprehensive knowledge about all the internals of these browsers, we consider our modifications to be non optimized.

As far as the web server is concerned, we implemented an Apache filter using the Ruby [35] programming language. The filter acts essentially as a pre-processor similar to the one used by PHP [4]. This means that the filter processes the page before any dynamic content, which might include XSS attacks, is inserted in the web document. The filter parses all document code and isolates all JavaScript (wrapped inside < < < < and > > > > or with the '+=' notation for events) using the XOR function and a random key. It finally encodes the result in Base64, places the <script> tag if needed, and attaches the random key to the X-IO-Key. The key is refreshed in every response.

We chose to implement the filter only for prototyping. However, our intention is to develop a regular Apache module in the near future to address any performance concerns.

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 is the overhead on the server, (c) what is the overhead on the web browser and, finally, (d) if the framework imposes any side-effects in the user's browsing experience. As discussed in the previous section, our implementation covers three web browsers (Firefox, WebKit and Chromium) and the Apache web server.

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 2.

Real-world exploits. To verify that xJS can cope with real-world XSS exploits, we use the repository hosted by XXSed.com [18] which includes a few thousands of XSS vulnerable web pages. This repository has been also used for evaluation in other papers [36]. 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 [42]. The task of the proxy is to attach

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

some JavaScript code that overrides the `alert()` function with a URL request 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. However, 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 [7]. 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 2. For the attacks presented in Section 2, 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 [47]. 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()`. Recall the example from Section 3 (see Figure 2). 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

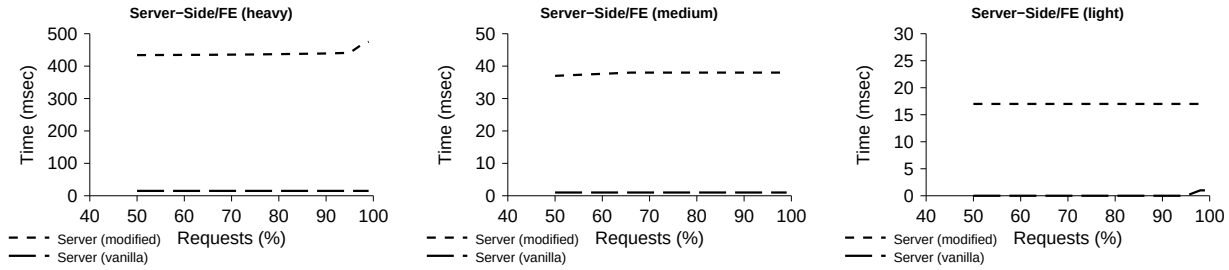

Figure 4: Server side evaluation when the Apache benchmark tool (`ab`) is requesting each web page through a Fast Ethernet link. In the worst case (heavy) the server imposes delay of a few hundreds of milliseconds, while in the normal case the delay is only a few milliseconds.

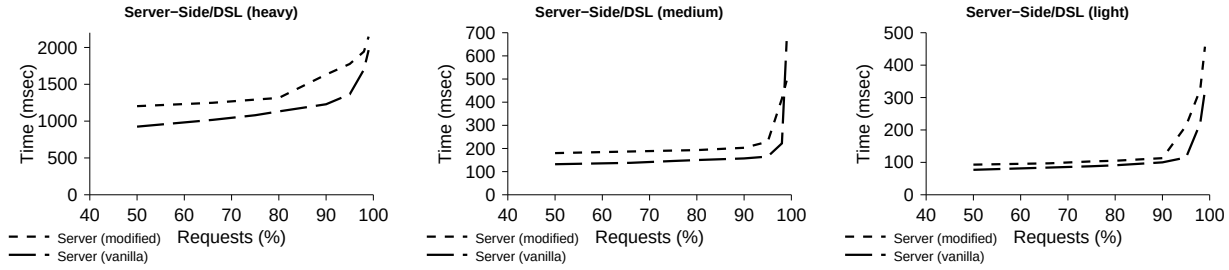

Figure 5: Server side evaluation when the Apache benchmark tool (`ab`) is requesting each web page through a DSL link. In the worst case (heavy) the server imposes a fixed delay of a few hundreds of milliseconds, like in the case of the Fast Ethernet setup (see Figure 4). However, this delay does not dominate the overall delivery time.

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 [6] 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 by 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 processing-intensive test in the whole suite, composed by 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.

Since these web pages do not use the special delimiters to separate the JavaScript source, we wrote a

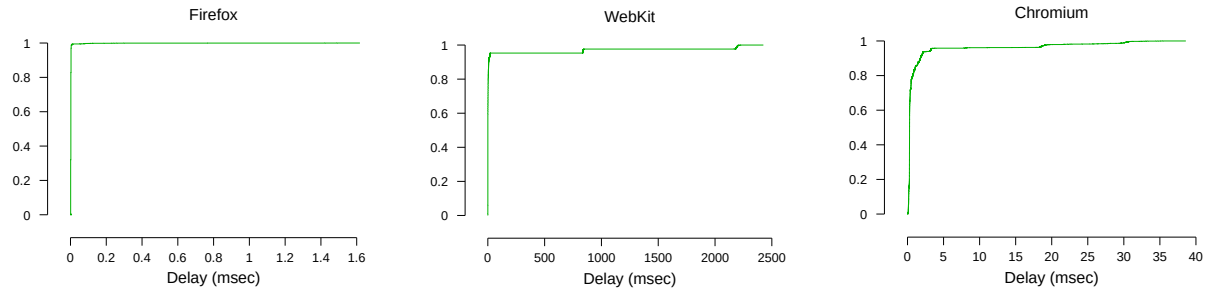

Figure 6: Cumulative distribution for the delay imposed by all modified function calls in the Firefox, WebKit and Chromium implementation, respectively. As delay we assume the time needed for the modified function to complete minus the time needed for the unmodified one to complete. Notice that the majority of function calls imposes a delay of a few milliseconds.

script to compile an HTML page to a document that separates all JavaScript using `< < < <` and `> > > >`. The script parses the target document for identifying all `<script>` tags and events (such as `onclick`, `onload`, etc.), and replaces them with our special delimiters.

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. Then, we perform the same experiments using the official tests (without the special delimiters) and with the `xJS` Apache filter removed. Finally, we repeat all the above with the `ab` client running in a typical downstream DSL line (6 Mbps).

Figure 4 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 tens (less than 20 ms and less than 40 ms for the light and normal test, respectively) to hundreds of milliseconds (about 450 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 could be considered significant. However, in Figure 5 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 filter imposes a fixed overhead of a few milliseconds per page, which is not the dominating overhead. Nevertheless, there is space for improvement. A similar implementation in native code will significantly outperform the Ruby filter, which uses complex and time consuming regular expressions. Yet, the Ruby filter does not cause a dramatic overhead in the server, since the isolation process is based on a fast operation, the `XOR` function.

4.3 Client Overhead

Having examined the server-side overhead, we now measure the overhead imposed on the browser by `xJS`. As in the previous section, 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.

In Figure 6 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 6 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.

To conclude, our modifications in all three browsers impose a negligible overhead in the order of a few milliseconds per call.

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 7 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.

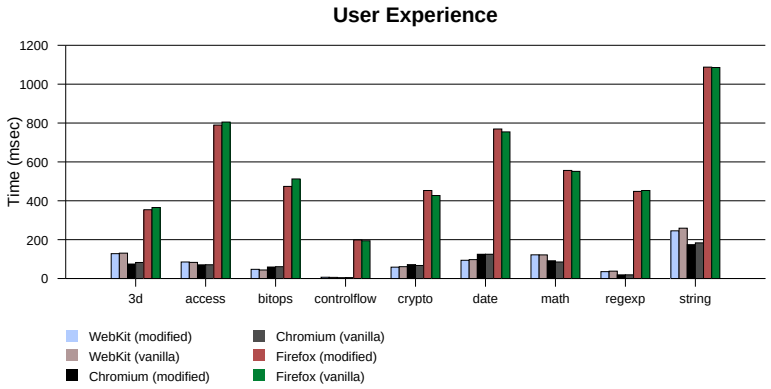

Figure 7: Results from the SunSpider test suite. Notice that for each modified browser the results are comparable with the results of its unmodified equivalent. That is, all de-isolated JavaScript executes as expected in both modified and unmodified browser.

4.5 Summary

Our evaluation findings may be summarized in the following points.

- `xJS` can successfully prevent all 1,380 real-world attacks collected from a well-known XSS attack repository, XSSed.com.

- Our framework imposes negligible computational overhead in both the server and the client side. As far as the server is concerned, xJS imposes a fixed overhead of a few milliseconds per page, which is not the dominating overhead. As far as the client is concerned, xJS does not impose a more than a few milliseconds overhead per call.
- xJS does not have any impact in the user's browsing experience. Browsers that support the framework have similar results with their unmodified equivalents in the SunSpider benchmark.

5 Discussion

Over the previous sections, we have presented xJS, which is a practical, lightweight framework against XSS attacks. We now discuss potential limitations of our approach and offer possible workarounds if possible. We further examine potential implementation issues and concerns that were left open.

Coding Style

The proposed separation scheme for client-side code using special delimiters requires the change of current programming disciplines, which might be undesirable. However, we have noted in Section 3 that introducing special delimiters for client-side code is not critical for xJS, yet, delimiters help the programmer to maintain a clean source corpus composed of code associated with three different domains (server-side, client-side and markup). We now discuss alternative options for an xJS framework without any special delimiters.

First, a web developer may use a server-side function, implemented in PHP or in a similar technology, in order to inject all JavaScript. For example, she may use an xjs(code, key) function, which isolates all input code using the XOR operator and then injects it to the document. This function has to also emit the correct X-IO-Key header field. Note that this is a similar approach to Blueprint [48], where the web programmer has to identify all unsafe parts of the code and then explicitly emit them using a specific Blueprint API in PHP. Second, a server module can be used to pre-process all web pages and isolate all JavaScript code without looking for special delimiters. This server module can act exactly as our filter using as special delimiter the <script> tag. In fact, we have implemented an xJS variant using the Nokogiri [3] library, which requires no special delimiters.

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. Finally, note that Blueprint [48] also emits parts of a web page encoded in Base64/Base32.

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 (see Section 2), 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

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. 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. Without explicit assistance from the database, we believe that it is hard to isolate the trusted code in this case. We plan to further investigate such cases in our future work.

Mashups

A mashup is a web site that collects information from third parties and presents it to the user. One could argue that a mashup is essentially a code injection process [33]. The main site fetches code from third party sites and injects it to the web documents it generates. For such cases, we may have several combinations of sites implementing or not `xJS`.

A fraction of the third party web sites implements the framework. This case produces a mix of client-side code in the web browser where some parts are isolated and other parts, trusted or not, are not. The final web document in this case will probably be non-functional. This issue can be addressed if each web server reports whether the framework is supported or not. This may be achieved using the `X-IO-Key` header, since when emitted it implies that the framework is supported. However, the security of the mashup, as far as XSS attacks are concerned, is not fully guaranteed.

None of the third party web sites implements the framework, but the main site. This case produces a negative result. The mashup isolates all collected, third party, client-side code in the final web document and thus advertises *all* generated client-side code as trusted. The code possibly includes code injections performed in any of the third party web sites. Thus, the framework must *not* be, in any case, applied in proxy environments on behalf of third party sites.

All third party web sites implement the framework, but not the main site. This case is considered healthy, since all authentic sources perform the isolation. The isolated code is trusted, even if the main site does not implement the framework. However, the main site must also transfer the keys (the `X-IO-Key`) in order for the browser to be able to perform the de-isolation process. As long as the main site is considered trusted, then the framework guarantees no code injection incidents in the final web document.

File Creation/Overwrite

An attacker may create or overwrite a file that is served by the web server with her own. The attacker's code is served as if it was part of the original web site's code base. In that case there is no way for our framework to identify that code as malicious. Although an attack of this form is possible, we believe that if a web server is vulnerable to file creation/overwrite the attacker can then inject *server-side* code and manage to takeover the complete web site. This threat model is much more superior than a typical XSS attack.

Attacks not Addressed by xJS

There are a few web threats that are not explicitly related to XSS attacks. However, they occasionally occur in the context of an XSS attack. Below, we shortly discuss threats that are not directly prevented by our framework descirbed in the following section, and we consider them out of the scope of this work.

Phishing [16] aims on luring a user to submit her credentials in a non authentic web site, which looks like an authentic one. For defending against Phishing, we refer the reader to [8]. Sometimes, Phishing can be achieved by injecting a malicious `iframe` in a vulnerable web page. This can be considered an XSS attack, however the impact is quite lower than the one imposed by injection of malicious JavaScript code. Our framework does not protect against `iframe` injection. Some ways to mitigate this particular XSS

attack variant can be found in [36].

Cross-Site Request Forgery (CSRF) and login CSRF attacks have been extensively studied in [12]. CSRF attacks are launched by malicious sites that generate web requests towards other popular web sites. The web requests are executed by the victim's web browser and, thus, the web browser fills in all the state (e.g. cookies) required, in order to send the web request with the victim's credentials. If the victim has already logged in, for example, in her e-banking or e-mail web site, then the requests launched by the malicious web site will succeed. Additionally, another type of CSRF attack, the *login CSRF* attack, does not assume that the user is already logged in a target web site. Instead, the attack aims at forcing the user to login in a web site with the attacker's credentials. In this paper, we do not address either the CSRF or the login CSRF attack. Proposals for mitigation of these attacks can be found in [12, 25, 23].

6 Related Work

The closest studies to xJS are BEEP [22], Noncespaces [20] and DSI [36]. Throughout the paper, we have highlighted certain cases where the aforementioned methodologies fail (e.g., see Section 2). We have presented attacks that escape whitelisting (proposed in [22]) and cases where DOM-based solutions [20, 36] are not efficient. Our framework, xJS, can cope with XSS attacks that escape whitelisting [10], and does not require any information related to DOM; xJS can also prevent attacks that leverage the content-sniffing algorithms of web browsers [11].

Our technique is based on Isolation Operators and it is inspired by the Instruction Set Randomization (ISR) [27]. Solutions based on ISR have been applied to native code and to SQL injections [14]. Some discussion about using ISR for XSS attacks can be found in [28], but to the best of our knowledge there has not been any systematic effort towards this approach before.

As far as XSS attack prevention is concerned, the literature is quite rich. In [49] the authors propose to use dynamic tainting analysis to prevent XSS attacks. Taint-tracking has been partially or fully used in other similar approaches [36, 45, 39, 37]. Although xJS does not rely at all on tainting, a source-code based tainting technique [50] can certainly assist in separating all server-produced JavaScript. In such a case 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.

Noxes [29] aims on finding and blocking unsafe URLs purely at the client side, while XSS-GUARD [13] aims on performing all input checking at the server side. We are not in favor of any techniques that perform content filtering, since we regard these processes as highly complicated. To support this argument we refer the reader to the XSS Cheat Sheet [21], which lists various obscure ways to perform a code injection.

Blueprint [48] 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 [43]. 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 [11] as well as attacks that modify the DOM structure using purely client-side code [30].

7 Conclusion

In this paper, we have presented xJS, a practical framework to prevent against the increasing threat of XSS attacks. xJS was motivated by a number of new code-injection attacks that defeat existing approaches. In particular, we have highlighted a new family of attacks that resembles the classic *return-to-libc* attack in native code. The attacks are based on injecting existing trusted code, which is already whitelisted, in the vulnerable web site. Mitigation of these attacks is critical for today's web, as modern web sites are rich in client-side code.

xJS is a fast and practical way to isolate all legitimate client-side code from possible code injections. The xJS framework, inspired by the Instruction Set Randomization technique, suggests the use of Isolation Operators (IO). An IO, such as one that is based on the XOR function, aims on *randomizing* all client-side source corpus for protecting it from code injections. All client-side code is transposed to a new domain, in our case the domain defined by the XOR function, and thus it is completely isolated from all code injections. Finally, our framework suggests policies expressed as Browser Actions. The web browser executes all trusted client-code after it has been de-isolated through an action, in our case the XOR function.

We implemented and evaluated our solution in three leading web browsers, namely FireFox, WebKit and Chromium, 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.

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] Nokogiri, HTML parsing library. http://nokogiri.rubyforge.org/.

[4] PHP: Hypertext preprocessor. http://www.php.net/.

[5] Popular web toolkits. http://code.google.com/webtoolkit/, http://struts.apache.org/, http://www.djangoproject.com/.

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

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

[8] B. Adida. BeamAuth: Two-Factor Web Authentication with a Bookmark. In *Proceedings of the 14th ACM conference on Computer and Communications Security*, pages 48–57. ACM New York, NY, USA, 2007.

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

[10] 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.

[11] 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.

[12] A. Barth, C. Jackson, and J. C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, 2008.

[13] P. Bisht and V. N. Venkatakrishnan. XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks. In *Proceedings of the 5th International Conference for Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 23–43, 2008.

[14] 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.

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

[16] 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.

[17] 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.

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

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

[20] 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.

[21] R. Hansen. XSS (Cross-Site Scripting) Cheat Sheet. Esp: for filter evasion. `http://ha.ckers.org/xss.html`.

[22] 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.

[23] M. Johns and J. Winter. RequestRodeo: Client Side Protection Against Session Riding. In *Proceedings of the OWASP Europe 2006 Conference*, pages 5–17.

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

[25] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing Cross-Site Request Forgery Attacks. In *Proceedings of the Second IEEE Conference on Security and Privacy in Communications Networks (SecureComm)*, 2006.

[26] 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.

[27] 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.

[28] 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.

[29] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks. In *Proceedings of the 21st ACM Symposium on Applied Computing (SAC)*, pages 330–337. ACM New York, NY, USA, 2006.

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

[31] 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.

[32] 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.

[33] B. Livshits and U. Erlingsson. Using Web Application Construction Frameworks to Protect Against Code Injection Attacks. In *PLAS '07: Proceedings of the 2007 workshop on Programming Languages and Analysis for Security*, pages 95–104, New York, NY, USA, 2007. ACM.

[34] 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.

[35] Y. Matsumoto. *Ruby programming language*. Addison Wesley Publishing Company, 2002.

[36] 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.

[37] 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.

[38] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceeding of the 13th Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

[39] 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.

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

[41] S. Pemberton, M. Altheim, D. Austin, F. Boumphrey, J. Burger, A. Donoho, S. Dooley, K. Hofrichter, P. Hoschka, M. Ishikawa, et al. XHTML 1.0: The extensible hypertext markup language, 2000.

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

[43] 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.

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

[45] 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.

[46] 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.

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

[48] 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.

[49] 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.

[50] 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.