University of Crete
Computer Science Department

⊕JS: Fighting Cross-Site Scripting Attacks Using Isolation Operators

Vasileios Pappas

Master's Thesis

June 2009
Heraklion, Greece

University of Crete
Computer Science Department

⊕JS: Fighting Cross-Site Scripting Attacks Using Isolation Operators

Thesis submitted by

Vasileios Pappas

in partial fulfilment of the requirements for the
Master of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Vasileios Pappas

Committee approvals: _____
Evangelos P. Markatos
Professor, Thesis Supervisor

_____
Sotiris Ioannidis
Assistant Researcher

_____
Maria G. Papadopouli
Assistant Professor

Departmental approval: _____
Panos Trahanias
Professor, Chairman of Graduate Studies

Heraklio, June 2009

Abstract

Web is a very dynamic ecosystem that is constantly evolving. It started as a collection of static HTML web pages and advanced to rich "Web 2.0" applications. As a side-effect, all this new functionality and features gave birth to new types of attacks.

In this thesis we focus on *Cross-Site Scripting* (XSS) attacks. We present new code injection attacks that defeat existing approaches for (XSS) prevention. This family of attacks resembles the classic *return-to-libc* attack in native code. Based on our findings, we proceed and present a fast and practical way to isolate all legitimate client-side code from possible code injections. We implement and evaluate our solution in one of the leading web browsers namely Firefox and in the Apache web server. Our framework can successfully prevent all 1,152 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. Finally, our modifications have no negative side-effects in the user's experience.

Supervisor: Prof. Evangelos P. Markatos

Περίληψη

Η Web πλατφόρμα είναι ένα εξαιρετικά δυναμικό οικοσύστημα το οποίο εξελίσσεται συνεχώς. Όταν πρωτοεμφανίστηκε, ήταν μια συλλογή από στατικές HTML σελίδες η οποία μετεξελίχθηκε σε πλούσιες "Web 2.0" εφαρμογές. Σαν αρνητική επίπτωση βέβαια, όλη αυτή η νέα λειτουργικότητα και χαρακτηριστικά, γέννησαν νέα είδη επιθέσεων.

Σε αυτήν την εργασία, επικεντρωνόμαστε στις επιθέσεις τύπου ***Cross-Site Scripting***. Παρουσιάζουμε νέες επιθέσεις εισαγωγής κώδικα οι οποίες ξεπερνούν τις υπάρχουσες τεχνικές για την αντιμετώπιση XSS επιθέσεων. Αυτή η οικογένεια επιθέσεων μοιάζει με της κλασικές return-to-libc επιθέσεις σε native κώδικα. Στηριζόμενοι στα αποτελέσματα μας, προχωράμε και παρουσιάζουμε ένα γρήγορο και πρακτικό τρόπο για την απομόνωση όλου του γνήσιου client-side κώδικα από κώδικα ο οποίος έχει εισαχθεί με κακόβουλο σκοπό. Υλοποιήσαμε και αξιολογήσαμε την λύση μας σε έναν από τους γνωστότερους Web περιηγητές, τον Firefox, και στον Web εξυπηρετητή, Apache. Η τεχνική απέτρεψε και τις 1.152 πραγματικές επιθέσεις που συλλέξαμε από μια γνωστή πηγή XSS επιθέσεων. Ως επί των πλείστων, η τεχνική μας δεν επιβάλει σημαντικό επιπλέον υπολογιστικό κόστος ούτε στον εξυπηρετητή, ούτε και στον περιηγητή. Τέλος, αυτές οι αλλαγές δεν παρουσιάζουν αρνητικές επιπτώσεις στην εμπειρία χρήσης του περιηγητή από τον τελικό χρήστη.

Επόπτης: Καθ. Ευάγγελος Μαρκάτος

Acknowledgments

First of all, I am grateful to Elias Athanasopoulos for his inspiration and ideas that actually led to this thesis. I would also like to thank everyone in our lab for their support and especially my supervisor, Evangelos P. Markatos, for his valuable help and advises during my Master studies.

Contents

List of Figures

List of Tables

1

Introduction

One of the most profound techniques for compromising a system is by performing a code injection. The process of injecting malicious code in an existing trusted code base can be applied to numerous different environments. For example, the attack can take place in native code, usually referred to as a buffer overflow [28], in a database environment through a `SQL` injection [6] or in the web browser's environment using the Cross-Site Scripting (XSS) technique.

Lately, we have observed a significant increase in XSS attacks. It is worth mentioning that *during the second half of 2007, 11,253 site-specific cross-site vulnerabilities were documented by XSSed, compared to 2,134 traditional vulnerabilities documented by Symantec* [31]. This observation translates to the fact that XSS attacks carried out on web sites were roughly 80% of all documented secu-

rity vulnerabilities. On the other hand, new technologies, such as AJAX [14], give more opportunities for code injection in the web environment. These technologies encourage the creation of rich and more complex interfaces with more vulnerabilities.

An XSS attack is typically carried out as follows. An attacker injects some client-side code, usually JavaScript, in a web document. The injection may be performed, but is not limited to, in a content submission. For example, a user posts a comment in a blog story in which she embeds some JavaScript. The result is that every web browser that renders the comment of the blog story will also execute the attacker's JavaScript. The malicious code can steal the users' cookies or perform arbitrary operations that can lead from simple annoyance to financial data loss.

There are numerous proposals for XSS attack mitigation. In this thesis we are particularly interested in policy based mechanisms like the one proposed by BEEP [16]. We spot limitations in the approach and develop new XSS attacks that succeed to bypass the policy framework. Finally, we propose our own framework, which we refer to as $\oplus$JS or in the simple form xJS . Our framework is inspired mainly by the concept of Instruction Set Randomization (ISR) [20]. xJS uses *Isolation Operators* in order to randomize the whole source corpus of client-side code and policies expressed as *Browser Actions*. Our framework guarantees that all trusted client-side code can be successfully isolated from possibly untrusted, and thus, it prevents all possible code injections in the browser environment.

Our contributions. We present new code injection attacks that defeat existing approaches, such as *Whitelisting* [16]. This family of attacks resembles the classic *return-to-libc* attack in native code [11]. We highlight all major weaknesses in preventing XSS attacks using a *DOM Sandboxing* technique. Based on our findings, we proceed and present a fast and practical way to isolate all legitimate client-side code from possible code injections. We implement and evaluate our solution in one of the leading web browsers namely FireFox and in the Apache web server. Our framework can successfully prevent all 1,152 real-world attacks that were collected

form an XSS attack repository [13]. Moreover, our framework imposes negligible computational overhead in the server and in the client side. Finally, our modifications have no negative side-effects in the user experience.

This thesis is organized as follows. In Chapter 2 we present in detail the reasons which drove us to this work. We enlist all the properties of the threat model we are trying to fight in Chapter 3. Our proposed solution is presented in Chapter 4 and is evaluated in Chapter 5. Its limitations are discussed in Chapter 6. We review related work in Chapter 7 and finally we conclude in Chapter 8.

2

Motivation

This chapter enumerates all our motivations. Lately, we have seen plenty of published papers for the detection and mitigation of XSS attacks. We would like to highlight the reasons for putting more effort on that direction.

Our framework focuses on three major factors: (a) practical implementation, (b) low computational overhead, and (c) attack coverage.

Practical Implementation. Our first objective is to have an easy and straight forward to implement solution. This can significantly assist to the hard task of deployment. As it is the case of other recent proposals [16, 24] our solution needs co-operation from both sides, web server and client, and thus deployment is considered a very demanding procedure. The implementation of this prototype took roughly a few days for the Firefox web browser and no more than 100 lines of code

(for the code isolation only – no policy handling). As far as the web server modifications are concerned, again the process was fairly easy. The modular architecture of the Apache web server allowed us to extend the web server's features with a reasonable amount of work.

Low Computation Overhead. Our framework could be seen as a *fast randomization* technique. Indeed, we are inspired in great extent from the Instruction Set Randomization (ISR) [20] concept. ISR has been proposed for defending against code injections in native code or in other environments, such as code executed by databases [10]. However, we believe that applying an ISR technique for dealing with XSS attacks is not trivial. The basic problem is that *web code* is produced in the server and it is executed in the client. In addition, the server lacks all needed functionality to manipulate the produced code. For example, randomizing all JavaScript code in the web server needs at least one full JavaScript parser running in the server. This can significantly increase the computational overhead and as a negative result the code will be parsed twice (one in the server during serving and one in the client during execution). Instead of trying to implement ISR for JavaScript we decided to implement an *isolation operator*, which transposes all produced code in a new isolated domain. The isolation operator is based on the `XOR` function which is considered fast; it can be found implemented as a CPU instruction in all modern hardware platforms.

Attack Coverage. Finally, our third concern was to come up with a solution that covers a large fraction of XSS exploits. Unfortunately, as it is discussed in detail in [24], XSS attacks have become significantly sophisticated. Sometimes they can be carried out even through a file upload [7]. As we show in the evaluation chapter, our framework successfully prevents all real-world attacks hosted by an XSS attack repository [13]. Moreover, as discussed in detail in Chapter 5, our framework can prevent more sophisticated attacks that are based on the careless use of the JavaScript `eval()` function.

To summarize our objectives, is the design and evaluation of a practical, computational inexpensive framework to defend against the majority of XSS attacks.

3

Threat Model

In this chapter we present in detail the threat model we are trying to address in this thesis. First, we present a short introduction of XSS attacks. Second, we describe two common practices, suggested in BEEP [16], that try to mitigate the problem, namely *whitelisting* and *DOM sandboxing*. We shortly present new attacks that can escape from these techniques and thus we define a family of XSS attacks that can actually take place in a real-world web sites. Our framework aims on shielding web sites from such code injections. Finally, we close this chapter with the limitations of our framework presenting attacks that can not be addressed.

3.1 XSS Overview

A cross-site scripting attack is a typical code injection performed in web applications. The attacker aims on injecting her code into a web document, which will eventually get rendered in a 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 activities. Most XSS attacks are carried out using JavaScript, although other client-side technologies can be also used. For example, consider an attacker submitting a comment, which embeds a JavaScript code snippet, in a web blog. All other users visiting the web blog and viewing the comments of the story will host the attacker's JavaScript code snippet in their browsers. The malicious JavaScript 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:

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

Thus, the attacker can collect all cookies from users viewing the web blog. The cookies can lead the adversary to hijack the sessions of the victim users, since quite frequently a web cookie contains information for user authentication.

Although traditionally XSS attacks have been associated with *cookies stealing*, the attack itself has a broader target set. As we argue in this thesis, plenty of modern web sites have employed rich user-driven AJAX interfaces. That is, most of the operations are carried out through client-side code. In addition, there are efforts for client-side toolkits [3]. Adversaries targeting this kind of web sites can benefit from the richness of the client-side code and thus create XSS exploits which perform operations on behalf of the victim user. These operations may cause annoyance, data loss or complete takeover of a user's profile, as we discuss in detail later in this chapter.

3.2 Whitelisting and DOM sandboxing

One way to prevent execution of untrusted client-side code in a web browser is by using a policy framework. This methodology was originally presented in BEEP [16] and it is enforced using two techniques, namely: *Whitelisting* and *DOM sandboxing*.

We shortly review *Whitelisting* and *DOM sandboxing* in this chapter.

Script whitelisting 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 JavaScript code snippet, if there is a cryptographic hash for that script in the white-list. If the hash is found the script is considered trusted and the browser executes it. If not, the script is considered non-trusted and the policy defines if the script will be rendered or not.

Notice that there is no checking for the location of the script inside the web document. For example, 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`[1] action).

DOM sandboxing works as follows. The web server places trusted scripts inside `div` or `span` HTML elements that are attributed as *trusted*. For example, consider the construct:

```
<div class='trusted'> ... 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. This method is vulnerable to the *node-splitting* attack, in which a malicious script is surrounded, on purpose, by misplaced HTML tags in order to escape from a DOM node. Consider for example the construct:

[1] 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 the `onload` event is also available for other elements (e.g., images using the `<img>` tag) included in the web document.

```
<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 should result the message to be displayed in **bold** style, rather than *italic*.

The authors of BEEP suggest a workaround for dealing with node-splitting, but we consider it rather inefficient, since all the data inside an untrusted `div` must be placed using a special coding idiom in JavaScript. We rather agree with a more elegant approach suggested in [15] (some very similar ideas have also been proposed in [24]).

Notice that DOM sandboxing requires the code injection to take place in an existing DOM tree. However, as it was recently shown, this is not always the case; a simple file upload and rendering is enough [7].

3.3 Defeating Whitelisting

Most XSS attacks are considered to happen by injecting arbitrary client-side code in a web document. This code 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. Return oriented programming suggests that an exploit may simply transfer execution to a place in `libc`[2], which may cause again execution of arbitrary code on behalf of the attacker. The difference with the traditional buffer overflow attack [28] is that

[2]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 [30].

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

FIGURE 3.1: A minimal Blog site demonstrating the whitelisting attacks.

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 in order to escape whitelisting. Instead of injecting her code, she can take advantage of existing *white-listed* code available in the web document. Note that, typically, a large fraction of client-side code is not executed upon document loading, but it is triggered during user events, such as mouse clicks or mouse movements. Below we enumerate some possible scenarios.

Annoyance. Assume the blog site shown in Figure 3.1 that has a JavaScript function `logout()`, which is executed when the user clicks *Logout* (see line 4 in Fig. 3.1). An attacker could perform an XSS attack by placing a script that calls `logout()` when a blog entry is rendered (see line 9 in Fig. 3.1). A user reading a blog story will be forced to logout. In a similar fashion, a web site that uses JavaScript code to perform redirection (for example using one of the standard ways, like `window.location.href=X`) can be also attacked by placing this white-listed code in an `onload` event (see line 10 in Fig. 3.1).

Data Loss. In a similar fashion, a portal which places user content that can be deleted using client-side code (AJAX [14] interfaces are popular in social networks like `Facebook` and `MySpace`) can be attacked by injecting the white-listed deletion code in an `onload` event (see line 11 in Fig. 3.1). This can be considered similar to a `SQL` injection attack [6], since the attacker implicitly grants 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 she needs to use already white-listed by the web server. For example, a bank web 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 trying to execute

script `S1`, which was triggered by an `onclick` event, the browser should check the white-list 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 used initially by the web programmer. Attacks may still happen. Consider the *Data Erasing* attack described above and an attacker that places the deletion code in `onclick` events associated with new web document's regions, not initially designed by the web programmer.

Finally, attacks that are based on injecting malicious data in white-listed scripts have been described in [24].

3.4 Defeating DOM Sandboxing

DOM sanbdoxing marks regions defined by `div` and `span` tags as trusted or non-trusted. JavaScript code is executed only if it is contained in a trusted region. We assume that a technique like Noncespaces [15] is used to prevent node-splitting.

We have two arguments against DOM sandboxing. First, we believe that marking a region as trusted or non-trusted may not always be that trivial. Especially, given the complexity of modern web sites, which are typically composed by hundreds of different `div` elements and thousands of JavaScript code. But, even if the marking is carried out correctly, there is no guarantee that a trusted `div` element will never host code from an XSS attack. The site designer should take care of this issue, programmatically. More precisely, the site designer should provide guarantees that a trusted `<div>` element will never host user input. Second, XSS attacks do not always need a DOM tree in order to take place. For example, consider an XSS attack which is *carried in* a PostScript file [7]. The attack will be launched when the file is previewed. There is high probability that upon previewing there will be no DOM tree to surround the injected code.

3.5 Addressable Attacks

In this thesis, we propose a framework that can address XSS attacks that are carried out using JavaScript. Our basic concept can also be applied to other client-side technologies (for example Adobe Flash), but our primary goal is JavaScript based XSS attacks.

Moreover, our framework aims on preventing JavaScript code injections that are based on third party code or on code that is already used by the trusted web site. Thus, our framework can prevent execution of trusted code, which has been injected by an attacker in specific portion of a web document.

Finally, our framework can in principle prevent attacks that are based on injected data and misuse of the JavaScript `eval()` function. Consider, the following example:

```
<?php
  $s = "<div id='malicious'>" . $_GET["id"] . "</div>";
  echo $s;
?>
<script>
eval(document.getElementById('malicious').innerHTML);
</script>
```

If an attacker insert 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 [26]. The main idea is to *mark* data that are foreign (i.e. they were part of a user input over the network or a database) as unsafe. Tainting analysis has also been used in [24] for dealing with attacks like the above one. Our framework can be augmented to prevent such attacks using tainting or by modifying the `eval()` function. In fact, our Firefox implementation prevents this kind of code injections. We discuss this issue in detail in Chapter 5.

3.6 Non-Addressable Attacks

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.

Phishing [12] aims on luring a user to submit her credentials in a non authentic web site, which looks like an authentic one. For defending Phishing we refer the reader to [5]. Sometimes, Phishing can be achieved by injecting a malicious `iframe` in a vulnerable web page. This can be considered as an XSS attack, however we believe that its 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 attack can be found in [24].

Cross-Site Request Forgery (CSRF) and login CSRF attacks have been extensively studied in [8]. 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. In the case that the victim has already logged in her e-banking or e-mail web site, then the requests enforced by the malicious web site will succeed having numerous dramatic consequences. On the other hand, there is a type of CSRF attacks, the *login CSRF* attack, that does not assume that the user is already logged in a target web site. Instead, the attack aims to force the user to login in a web site with the attacker's credentials.

In this thesis we do not address either CSRF neither login CSRF attacks. Some proposals for mitigation of these attacks can be found in [8, 17, 19].

4

Architecture

In this chapter we present in detail the xJS framework for preventing XSS attacks. The fundamental concept of the framework is based on *Isolation Operators*. However, in order to be applied practically we propose *Code Separation* for client-side code and *Action Based Policies* in the browser environment. We review each of these three concepts. At the end of this chapter we provide information about our implementation prototype in the Firefox web browser and the Apache web server.

4.1 Isolation Operators

One methodology that can be used to prevent code injections is Instruction Set Randomization (ISR), which has been applied to native code [20] and to SQL [10]. The basic concept behind ISR is to randomize the instruction set in such a way that

a code injection is not able to *speak the language of the environment* [21] and thus is not able to execute. In this thesis, inspired by ISR, we introduce the concept of Isolation Operators (IO), which essentially randomize the whole source corpus and not just the instruction set.

We follow this approach for the following reason. 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 in the server. This can significantly increase the computational overhead and as a negative result the code will be parsed twice (one in the server during serving and one in the client during execution).

Applying an IO such as the `XOR` function can effectively randomize all JavaScript source, not just the instruction set. The *isolation* is achieved since all trusted code produced by the web server has been transposed to a new domain: the `XOR` domain. The web browser has to *de-isolate* the source by applying again the IO and then execute it.

4.2 Code Separation

Traditionally we think 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 in 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 `< < < <` and `> > > >`. In Figure 4.1 we depict an `xJS` example.

On the left is the source code as it exists in the web server and on the right is the same source as it is fetched by the web browser. The JavaScript source has been XORed and a Base64 [18] encoding has been applied in order to transpose all non-printable characters to the printable ASCII range.

```
<div>                          <div>
  <<<<                           <script>
  alert("Hello World");          vpSUlJTV2NHGwJyW/NHY...
  >>>>                           </script>
</div>                         </div>
```

FIGURE 4.1: Example of a web page that is generated by our framework.

This kind of code separation assists significantly to a possible manipulation of the whole client-side code corpus. If this scheme is enforced then applying an IO to all produced JavaScript is trivial.

4.3 Action Based Policies

Finally, 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 different key. The keys are transmitted in HTTP headers (see the use of X-IO-Key header, later in this chapter) 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 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 of a brute force attack to succeed is low.

Defining the complete policy set is out of the scope of this thesis. For the purpose of our evaluation (see Chapter 5) we use one policy, which is expressed as "*de-isolate (apply XOR) and execute*".

4.4 Implementation

In order to test our framework we modified the Firefox web browser. We also created a filter for the Apache web server.

Implementing the `xJS` prototype took roughly a few days for the web browser side and no more than 100 lines of code. Since we have lack of knowledge about all the internals of such a large project, we consider our modifications to be non optimized. Our major concern was to have working prototypes in order to test the framework and not official patches.

The modified web browser operates in the following way. For each HTTP response it searches for the `X-IO-Key` header field. If found, it uses its value, which is 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.

For our prototype implementation, we altered two functions in the Firefox's JavaScript implementation. The function that handles all events (such as `onload`, `onclick`, etc.) and 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. It is worthy noting here that these functions operate recursively. We further discuss this issue in Chapter 5.

As far as the web server is concerned, we implemented an Apache filter using the Ruby [23] programming language. The filter acts essentially as a pre-processor similar to the one used by PHP [2]. It parses all document code and isolates all

JavaScript (injected inside < < < < and > > > > or with the '+=' notation for events) using the XOR function and a random key. It finally encodes the result in Base64, it places the <script> tag if needed and it attaches the random key to the X-IO-Key. The key is refreshed in every response.

We chose to implement the filter, in order to have a prototype rapidly. However, our intention is to develop an Apache module in the near future, mainly for performance issues.

5 Evaluation

In this chapter we evaluate our xJS prototype. We have modified the Firefox web browser and we have extended the functionality of the Apache web server using a filter. Our evaluation seeks to answer four questions: (a) how many real XSS attacks can be prevented, (b) what is the overhead in the server, (c) what is the overhead in the web browser and, finally, (d) does the framework impose any side-effects in the user's experience. Below, we address each of these four questions separately. We summarize the results of our evaluation at the end of this chapter.

5.1 Attack Coverage

In this part we try to identify how effective is the xJS framework in preventing real-world XSS attacks. We used the repository hosted by xssed.com [13] which

includes a few thousands of XSS vulnerable web pages. This repository has been also used for evaluation in other papers [24].

Apparently, there are a couple of issues which make the attack coverage evaluation of `xJS` quite challenging. First, some attacks listed in `xssed.com` have been already fixed. The web site hosts the vulnerable web pages after the code injection has happened. However, this cannot be of use, since `xJS` aims on preventing the code injection before it takes place. Second, we have no access to the vulnerable web server and, thus, we cannot use our server-side filter. In order to overcome all these issues we choose to conduct the evaluation in the following way.

URLs	**Number**	**Percentage**
`iframe` attack vector	384	4,1%
redirection to `xssed.com`	416	4,5%
redirection & `iframe`	60	0,6%
Failed to parse	2,518	27,5%
Still vulnerable	1,152	12,5%
Total	9,156	100,0%

TABLE 5.1: Evaluation data categorization.

First, we needed to resolve all still vulnerable web sites. In order to do so, we downloaded 9,156 URLs from `xssed.com`. From this sample we excluded 384 URLs that had an `iframe` as attack vector, 416 URLs that had a redirection to `xssed.com` as attack vector and 60 URLs that had both an `iframe` and a redirection `xssed.com` as attack vector. We remained with all URLs that were vulnerable at some period of time and the vulnerability could be triggered using the `alert()` function. We proceeded and requested each vulnerable page through a custom proxy server we built. The task of the proxy was to attach a small JavaScript code snippet that overrides the `alert()` function with a URL request to a web server located in our laboratory. Since, all attack vectors were based on using the `alert()` function, in order to demonstrate the vulnerability, the web server

recorded all successful attacks in its access logs. Using this methodology we managed to identify 1,152 web pages, which were still vulnerable. Our proxy failed to open 2,518 web pages due to parsing errors. These are web pages that have misplaced HTML code that was unable to be fully parsed. Our methodology suggests that about 1 in 7 web pages had not been fixed after the vulnerability was published. Table 5.1 gives a synoptic view of the data that where used for our evaluation.

Second, we needed a way to simulate the web server filter we use in the `xJS` framework in order to perform the client-side code isolation. Again, we used our proxy in the following way. For each vulnerable page, the proxy requested again the web document but with a different attack vector. For example, for the attack vector below:

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

the proxy requested the URL:

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

Using this methodology, the proxy managed to build all vulnerable web pages with the attack vector embedded but not in effect. The next step was the proxy to parse all vulnerable pages, identify all JavaScript code, which we consider trusted and isolate it using the `XOR` IO. At this stage, we had all vulnerable web pages with all JavaScript isolated and with the attack vector defunct. The last step was to re-enable the attack vector by replacing the *xscript* with *script* again and return the web page to the browser. All web pages were also including the JavaScript snippet for the `alert()` overloading. After the end of the experiment, we recorded *zero* requests to our web server. This means that *all* XSS attacks were prevented successfully by our framework. Thus, we concluded that, as far as the sample collected through `xssed.com` is concerned, the `xJS` framework has 100% success in XSS attack prevention.

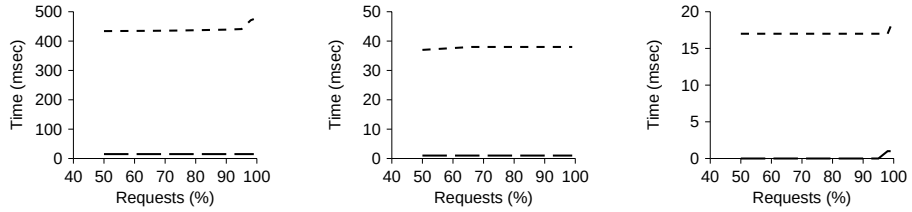
Finally, we tested attacks we have presented in Chapter 3 that are based on a code injection in data and the careless use of `eval()`. Recall the example we used in Chapter 3:

```
<?php
  $s = "<div id='malicious'>" . $_GET["id"] . "</div>";
  echo $s;
?>
<script>
eval(document.getElementById('malicious').innerHTML);
</script>
```

Normally, someone would expect xJS to be unable to prevent this kind of attack. The injected code will be in plain text (non-isolated), but unfortunately it will be attached to isolated code after the de-isolation process. Thus, the injected code will be executed as it was trusted. However, there is a way to prevent this behavior. In fact, the internal design of Firefox gave as this feature with no extra cost. Firefox is using 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($_GET('id'));`, it will de-isolate it, call the `eval()` function, which in principle will call itself in order to execute the `$_GET('id')` part. At the 2nd call, the `eval()` will try again to de-isolate the `$_GET('id')` code, which if it is in plain text will fail and thus will never get executed.

5.2 Server Overhead

In this part we try to identify the overhead in the server imposed by xJS. In order to measure the server overhead, we need to have a set of web pages that embed a significant amount of JavaScript. We chose to use some web pages with JavaScript code that ship with the SunSpider suite [4]. We manually selected three JavaScript tests. One which is considered *heavy* because it is a hard test involving string operations with many lines of JavaScript (it is probably the hardest test in the whole suite), one which is considered *normal* because it has a typical amount of source code like most of the other tests, and one which is considered *light* because it is a



(a) Heavy (b) Medium (c) Light

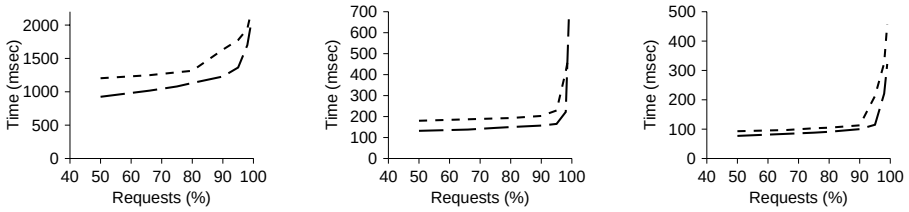
FIGURE 5.1: Server side evaluation using the Apache benchmark tool (`ab`) (Fast Ethernet link).

few lines of JavaScript, involving bit operations. We will further refer to this tests as heavy, medium and light.

Since, these web pages do not use the special delimiters to separate the JavaScript source, we wrote a 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 it replaces them with our special delimiters.

We conducted two sets of experiments. During the first experiment we used `ab` [1], which is considered as the de-facto tool for benchmarking an Apache web server, over a Fast Ethernet (FE) network. We configured `ab` to issue 100 requests for the heavy, normal and light web page, respectively. After the end of the experiments, we removed the Apache filter and we run `ab` again to benchmark the web server with our modifications removed. In this run, we used the official tests (without the special delimiters). Finally, we repeated all the above experiment with an `ab` client running in a typical DSL line (6 Mbps).

In Figure 5.1 we depict the results for the benchmarking when the `ab` tool is connected to the web server using a FE connection. Notice, that the modified Apache imposes from a few tens to hundreds of milliseconds in the worst case (the heavy web page – Figure 5.1(a)). From a first look this is quite unpromising. However, in Figure 5.2 we depict the same experiments over the DSL link. The



(a) Heavy (b) Medium (c) Light

FIGURE 5.2: Server side evaluation using the Apache benchmark tool (`ab`) (DSL link).

overhead is still the same and it is negligible, since 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 some heavy and time consuming regular expressions. Apparently, even the Ruby filter does not cause a dramatic overhead in the server.

5.3 Client Overhead

In this part we try to identify the overhead in the browser imposed by `xJS`. We use the Sunspider test suite with 100 iterations. That means that every test of about 15 JavaScript programs is executed 100 times. We used the `gettimeofday()` function to measure the execution time of the modified functions in the browser. For our prototype implementation we have altered two functions. The one that is responsible for handling code associated with events, such as `onclick`, `onload`, etc., and the one that is responsible for evaluating whole JavaScript code blocks. In Firefox we modified internally the JavaScript `eval()` function which is recursive.

In Figure 5.3 we depict the cumulative distribution of the delays imposed by all modified recorded function calls for Firefox during a run of the Sunspider suite

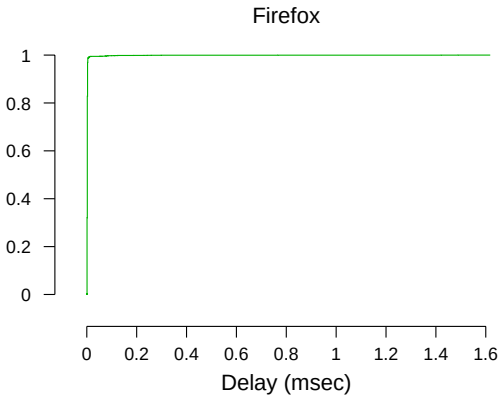


FIGURE 5.3: Cumulative distribution for the delay imposed by all modified function calls in the Firefox.

for 100 iterations. As delay we assume the time needed for the modified function to complete minus the time needed for the unmodified one to complete.

All delays are less than 1 millisecond. Firefox needed about 500,000 calls for the 100 iterations of the test suit. In Figure 5.3 we plot the first 5,000 calls (these calls correspond to one test iteration only) of the complete set of about 500,000 calls, in order to be more assimilable.

5.4 User Experience

Finally, in this part we try to identify if the user's experience changes due to `xJS`. We run the Sunspider suite for 100 iterations with the modified web browser and with the equivalent unmodified one and record the output of the benchmark. In Figure 5.4 we plot the results for 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. 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.

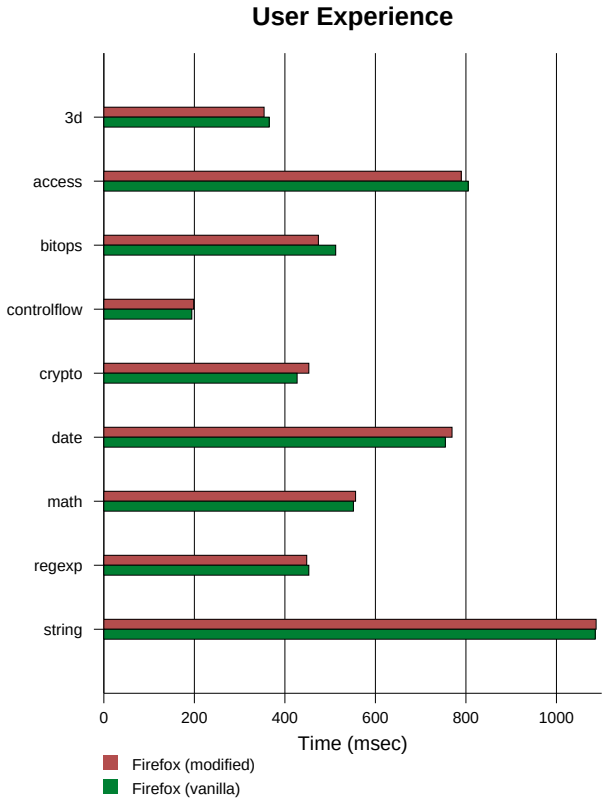


FIGURE 5.4: Results from the Sunspider test suite.

5.5 Summary

We summarize our evaluation results. Our evaluation proves that xJS can successfully prevent all 1,152 real-world attacks that were collected from an XSS attack repository [13]. Moreover, our framework imposes negligible computational overhead in the server and in the client side. Finally, our modifications do not yield any negative side-effects in the user's experience. All client-side code executes as expected.

6

Limitations

In this chapter we highlight some aspects that we consider as limitations of our approach. We shortly review each one of them and provide possible workarounds. This chapter establishes the basic roadmap for our future work.

Coding Style. The proposed separation scheme for all client-side code using special delimiters requires the change of current programming disciplines.

However, we understand that this scheme may produce negative reactions. Thus, we discus some alternative ways to achieve the same result.

First, the web developer can 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 will isolate using the `XOR` operator all input code and then inject it to the document. Apparently, this function has to also

emit the correct `X-IO-Key` header field. 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, but using as special delimiters the `<script>` and `</script>` tags.

Dynamic Code. 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 that they are produced by user inputs. A possible scenario is that the code injection can take place *in* the database. This can happen if trusted code and a user input that may contain untrusted code are merged together before included in the final web page. This case is quite hard to track, since it involves the programmer's logic in great extent. The critical part is that client-side code is hosted in another environment (the database) which is also vulnerable to code injections. We consider it hard to isolate the trusted code without explicit assistance from the database. Our plans is 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. The main site will fetch code from third party sites and inject it to the web documents it generates. There is a number of possible things that may produce confusion or even have negative result.

A fraction of the third party web sites have implemented the framework. This case will produce a mixed up of client-side code in the web browser. Parts will be isolated and parts, trusted or not, will be in plain text. There will be confusion and the final web document will be probably non functional. This issue can be addressed if each web server reports if the framework is supported or not. This can be achieved using the `X-IO-Key` header, since when it is 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 have implemented the framework, but the main site. This case will produce a negative result. The mashup will isolate all collected, third party, client-side code in the final web document and thus will advertise *all* generated client-side code as trusted. The code will possibly include 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 have implemented 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 the browser to be able to perform de-isolation. As long as the main site is considered trusty, then the framework guarantees no code injection incidents in the final web document.

File Creation/Overwrite. An attacker could create or overwrite a file that is served by the web server and insert JavaScript code using the IO. 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 even takeover the whole web site.

7

Related Work

The most relevant work to this thesis is the BEEP [16] framework. In this thesis, we essentially tried to highlight weaknesses in the methodology proposed by BEEP and to develop a framework that can address all possible issues. We have reviewed in detail most aspects of BEEP in Chapter 3.

Our technique is based on Isolation Operators and it is inspired by the Instruction Set Randomization (ISR) [20]. Solutions based on ISR have been applied to native code and to `SQL` injections [10]. Some discussion about using ISR for XSS attacks can be found in [21], but to the best of our knowledge there has not been any systematic effort towards this approach.

As far as XSS attack prevention is concerned, the literature is quite rich. In [32] the authors propose to use dynamic tainting analysis to prevent XSS attacks. Taint-

tracking has been partially or fully used in similar approaches [24, 25, 27, 29]. Noxes [22] aims on finding and blocking unsafe URLs purely at the client side, while XSS-GUARD [9] aims on performing all input checking at the server side.

Web attacks that are not in principle related to XXS, but they somehow assist in a XSS attack have also been presented lately. More precisely, Noncespaces [15] address the *node splitting* attack, where malicious code escapes a trusted region in a web document. Their solution is also based on ISR. Our framework is not vulnerable to the node splitting attack, since it doesn't need a DOM tree to have effect. More interestingly in [7] the authors have presented ways to perform an XSS attack through file uploads. Our framework can successfully prevent such XSS attacks.

8

Conclusion

In this thesis, we tried to explore new techniques for performing XSS attacks and how they can be prevented. More precisely, we presented new code injection attacks that defeat existing approaches for XSS attack prevention. This family of attacks 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. Taking into account that modern web sites are rich in client-side code, we consider these attacks critical. Based on our findings, we proceeded and presented `xJS` a fast and practical way to isolate all legitimate client-side code from possible code injections.

Our 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 of the source corpus that it is to be protected from code injections. The result is that all client-side code is transposed to a new domain, in our case the domain defined by the XOR operator, and thus it is completely isolated from all code injections. Finally, our framework suggests policies expressed as Browser Actions. More precisely the web browser executes all trusted client-code after it has been de-isolated through an action, in our case the XOR operation.

We implemented and evaluated our solution in the Firefox web browser and in the Apache web server. Our evaluation aimed on identifying if (a) the framework can successfully prevent real-world XSS attacks, (b) if the framework imposes unrealistic computational overhead in the server side, (c) if the framework imposes unrealistic computational overhead in the web browser and (d) if the user's experience is altered due to our modifications in the browser. Our extensive evaluation proved that xJS can successfully prevent all 1,152 real-world attacks that were collected from an XSS attack repository [13]. Moreover, our framework imposes negligible computational overhead in the server and in the client side. Finally, our modifications did not yield any negative side-effects in the user's experience. All client-side code was executed as expected.

Last but not least, we reviewed possible issues we consider as limitations of our approach. We presented each one of those and offered workarounds. These issues compose the basic roadmap for our future work.

Bibliography

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

[2] PHP: Hypertext Preprocessor. `http://www.php.net/`.

[3] Popular Web Toolkits. `http://code.google.com/webtoolkit/`, `http://struts.apache.org/`, `http://www.djangoproject.com/`.

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

[5] 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.

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

[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, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *To appear at the 15th ACM Conference on Computer and Communications Security (CCS 2008)*, 2008.

[9] P. Bisht and V. Venkatakrishnan. XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks. *Lecture Notes in Computer Science*, 5137:23–43, 2008.

[10] S. Boyd and A. Keromytis. SQLrand: Preventing SQL injection attacks. *Lecture notes in computer science*, 3089:292–302, 2004.

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

[14] J. Garrett et al. Ajax: A new approach to web applications. *Adaptive path*, 18, 2005.

[15] 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.

[16] 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.

[17] M. Johns and J. Winter. RequestRodeo: client side protection against session riding. In *Proceedings of the OWASP Europe 2006 Conference, refereed papers track, Report CW448*, pages 5–17.

[18] S. Josefsson. The Base16, Base32, and Base64 Data Encodings, 2003.

[19] 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.

[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. *IEEE Security and Privacy*, 7(1):18–25, 2009.

[22] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 330–337. ACM New York, NY, USA, 2006.

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

[24] 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)*, 2009.

[25] 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.

[26] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed System Security Symposium (NDSS)*, 2005.

[27] A. Nguyen-tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *In 20th IFIP International Information Security Conference*, pages 372–382, 2005.

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

[29] 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)*, 2009.

[30] 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.

[31] Symantec Corp. April 2008. 1-3. Retrieved on 2008-05-11. Symantec Internet Security Threat Report: Trends for July-December 2007 (Executive Summary).

[32] 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 Network and Distributed System Security Symposium (NDSS'07)*, 2007.