University of Crete
Computer Science Department

RaJa
Cross-Site Scripting Detection and Prevention Using Source Randomization

Antonis Krithinakis

Master's Thesis

October 2010
Heraklion, Greece

University of Crete
Computer Science Department

RaJa
Cross-Site Scripting Detection and Prevention
Using Source Randomization

Thesis submitted by

Antonis Krithinakis

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

THESIS APPROVAL

Author: _____
Antonis Krithinakis

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

_____
Sotiris Ioannidis
Principal Researcher

_____
Maria G. Papadopouli
Assistant Professor

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

Heraklio, October 2010

Abstract

Cross-site Scripting (XSS) attacks have become a major threat to web security over the last few years. XSS extends the traditional code-injection attack in native applications to web applications. A promising technique for preventing code-injection attacks in general is Instruction Set Randomization (ISR).

In this thesis we design, implement and evaluate a complete randomization framework for JavaScript aiming at detecting and preventing Cross-Site Scripting (XSS) attacks. Our framework randomizes JavaScript source without changing the code structure. Thus, it can be practically deployed in modern web applications, which intermix server-side, client-side and markup technologies in order to produce dynamic content.

Our framework builds on well-known and highly used open source projects like Firefox web browser and Apache web server. Our solution can successfully detect and prevent the execution of real-world XSS attacks. We enable the framework for four popular web applications. It can successfully locate and randomize JavaScript code hidden in hundreds of thousands lines of mixed code. Our experience with these applications suggests that they need only a few manual changes in order to be enabled and more importantly these changes can be categorized. Finally, we show that our framework imposes a relatively small overhead on the server and negligible overhead on the client.

Supervisor: Professor Evangelos Markatos

Περίληψη

Οι επιθέσεις τύπου Cross-site Scripting (XSS) έχουν γίνει μια σημαντική απειλή για την ασφάλεια στο web τα τελευταία χρόνια. Το XSS επεκτείνει την παραδοσιακή επίθεση του code-injection από τις κανονικές εφαρμογές στις εφαρμογές του web. Μια πολλά υποσχόμενη τεχνική για την πρόληψη των code-injection επιθέσεων είναι η τεχνική του Instruction Set Randomization (ISR).

Σε αυτήν την εργασία σχεδιάζουμε, υλοποιούμε και αξιολογούμε ένα πλήρες σύστημα τυχαιοποίησης για τη JavaScript με στόχο την ανίχνευση και την πρόληψη Cross-site Scripting (XSS) επιθέσεων. Το σύστημα μας τυχαιοποιεί τον JavaScript κώδικα χωρίς να αλλάζει την δομή του. Αυτό πρακτικά σημαίνει ότι το σύστημα μας μπορεί να χρησιμοποιηθεί σε σημερινές εφαρμογές που συνδυάζουν server-side, client-side και markup τεχνολογίες με σκοπό να παράγουν δυναμικό περιεχόμενο.

Το σύστημα μας χτίζεται πάνω σε πολύ γνωστά και ευρέως χρησιμοποιούμενα προγράμματα ανοιχτού κώδικα όπως είναι ο περιηγητής Firefox και ο διακομιστής ιστοσελίδων Apache. Η λύση μας μπορεί να εντοπίσει και να αποτρέψει την εκτέλεση επιθέσεων XSS που υπάρχουν στο πραγματικό κόσμο. Εφαρμόζουμε το σύστημα σε τέσσερις δημοφιλείς web εφαρμογές. Το σύστημα μπορεί με επιτυχία να εντοπίζει και να τυχαιοποιεί κώδικα JavaScript που βρίσκεται κρυμμένος σε εκατοντάδες χιλιάδες γραμμές ανάμικτου κώδικα. Η εμπειρία μας με αυτές τις εφαρμογές δείχνει ότι προκειμένου να εφαρμοστεί το σύστημα μας χρειάζονται λίγες αλλαγές από τους προγραμματιστές των εφαρμογών και ότι αυτές οι αλλαγές μπορούν να κατηγοριοποιηθούν. Τέλος, δείχνουμε ότι η επιβάρυνση που επιβάλλεται από το σύστημα μας στο διακομιστή και στον περιηγητή είναι σχετικά μικρή.

Επόπτης: Καθηγητής Ευάγγελος Μαρκάτος

Acknowledgments

I am grateful to my supervisor, Professor Evangelos Markatos, for his guidance and continual encouragement through my studies. I also feel very grateful to Elias Athanasopoulos who has been abundantly helpful and has assisted me in numerous ways. I specially thank him for his infinite patience and support during my research.

My best regards to all past and current members of the Distributed Computing Systems Laboratory in ICS/FORTH for their support. Especially i would like to mention Assistant Professor George Kopidakis, Christos Papachristos, Andreas Makridakis, Antonis Papadogianakis, George Vasiliades, Dimitris Antoniadis, Spyros Antonatos, Iason Polakis, Petros Politopoulos, Eleni Gessiou, Spyros Ligouras, Zaxarias Tzermias, Thanasis Petsas, George Sykiotakis and Michalis Orfanoudakis.

Finally, I owe special gratitude to my family for continuous and unconditional support. It has been a constant source of love and strength all these years.

This work was carried out with the financial and technical support of the Institute of Computer Science (ICS) of the Foundation for Research and Technology - Hellas (FORTH).

This thesis is based on:

"An Architecture For Enforcing JavaScript Randomization in Web2.0 Applications", authored by Elias Athanasopoulos, Antonis Krithinakis and Evangelos P. Markatos. The paper was accepted for the proceedings of the 13th Information Security Conference (ISC) in October 2010 (Boca Raton, Florida, USA).

"Isolating JavaScript in Dynamic Code Environments", authored by Antonis Krithinakis, Elias Athanasopoulos, and Evangelos P. Markatos. The paper was accepted for the proceedings of the 1st Workshop on Analysis and Programming Languages for Web Applications and Cloud Applications (APLWACA), co-located with PLDI, in June 2010 (Toronto, Canada).

Στους γονείς μου, Μανώλη και Ασημίνα.

Στον αδερφό μου, Γιώργο.

Contents

List of Figures

List of Tables

1

Introduction

1.1 Cross-Site Scripting Attacks

Cross-Site Scripting (XSS) [2] is a type of computer security vulnerability typically found in web applications that enables malicious attackers to inject client-side script into web pages viewed by other users. It extends the traditional code-injection attack in native applications to web applications and it is considered as one of the most severe security threats over the last few years [35]. According to a September-2009 report published by the SANS Institute [35], *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. 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.

1.2 ISR and RaJa motivation

One promising approach for dealing with code-injection attacks in general is Instruction Set Randomization (ISR) [25]. The fundamental idea behind ISR is that the trusted code is transformed in randomized instances. Thus, the injected code when plugged to the trusted base *cannot speak the language of the environment* [26]. So far, the technique has been applied in native code [25] and SQL [16]. Architectures inspired by ISR for countering XSS attacks have been also proposed, like Noncespaces [20] and xJS [10]. However, to the best of our knowledge there has been no systematic effort for applying a randomization scheme directly to a client-side programming language, like JavaScript. This task is far from trivial. The problem is that the server part lacks the functionality to parse the client side code and successfully randomize it. In addition, even if the extra overhead of a client-side language parser running in the server side could be considered tolerable, all modern web applications intermix server-side, client-side and markup technologies in order to produce dynamic content. In such cases the parser should recognize and handle tokens of a third-party programming language.

In this thesis we design, implement and evaluate our own framework which we refer to as RaJa . In our solution randomization is directly applied to JavaScript source. We modify a popular JavaScript engine, Mozilla SpiderMonkey [7], to carefully modify all JavaScript identifiers and leave all JavaScript literals, expres-

```
1 <script>
2 var message = <?php echo "Hello World<br/>"; ?>;
3 document.write(message);
4 </script>
```

FIGURE 1.1: PHP mixes with JavaScript. In this example the PHP expression acts as a string literal.

sions, reserved words and JavaScript specific constructs intact. We further augment the engine to recognize tokens identifying the existence of a third-party programming language. This is driven by two observations:

- JavaScript usually mixes with one and only one server-side scripting language, like PHP, with well defined starting and ending delimiters.
- Server-side scripting elements when are mix-up with JavaScript source, they act as JavaScript identifiers or literals in the majority of the cases (see Figure 1.1).

To verify these speculations we deploy RaJa in four popular web applications. RaJa fails to randomize 9,5% of identified JavaScript in approximately half a million lines of code, which contain JavaScript, PHP and markup (HTML/XML). A carefully manual inspection of the failed cases suggests that failures are due to coding idioms that can be grouped in *five* specific practices. Moreover, these coding practices can be substituted with alternative ones. For example, a web application of approximately 150,000 lines of code, like WordPress, needs only a few code changes to be RaJa -enabled out of the box.

RaJa is not just a randomization scheme, but rather a complete architecture for randomizing all JavaScript source of a web application, without the web programmer being aware of it. More precisely, RaJa is based on the following assumption:

- All JavaScript stored in files on the server's disk is considered trusted.

- Only the JavaScript stored in files on the server's disk is considered trusted

The second assumption can be relaxed to account for JavaScript templates in databases, if read-only tables for this particular purpose introduced. Based on these assumptions, RaJa can be enabled in a web application by simply placing the web application's code in a web server that runs the RaJa tools and libraries. We discuss all components of the RaJa architecture in Chapter 2.

1.3 RaJa Benefits

RaJa has to offer the following benefits, compared to other existing XSS mitigation techniques.

- **DOM-based Solutions.** RaJa does not rely on the DOM [29] like several other approaches [18, 20, 31]. These approaches are promising and can capture effectively a large range of attacks, since an XSS code injection usually modifies the DOM structure. Nevertheless, they have limitations. They cannot detect attacks that can potentially take place while the DOM structure is absent [12]. In addition they fail to detect and prevent the unofficially termed *DOM-Based XSS* or *XSS of the Third Kind* attacks [28]. 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. 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 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.

- **Script Whitelisting.** RaJa does not rely on script whitelisting like BEEP [22]. Script whitelisting works as follows. For each client-side script that is con-

sidered trusted a cryptographic hash value is generated. The web application that supports script whitelisting includes a list of these hashes. Every time a JavaScript snippet is executed, the browser checks the whitelist. If an entry is found for the specific script then is considered trusted and its execution is not suspended. In different cases, there is a policy that defines whether the script will be executed or not. Script whitelisting suffers from code-injection attacks that are based on existing trusted (whitelisted) code [11]. For example, consider a script that is placed by the application programmer to run upon document rendering. For this specific script an entry exists in the whitelist. Script whitelisting technique does not check the location of the script inside the web document. An attacker could place the specific script to be executed upon a user's click.

1.4 Contributions

This thesis has the following contributions:

- We design, implement and evaluate RaJa, a complete framework for applying a randomization scheme directly to JavaScript, in order to counter Cross-Site Scripting attacks. Compared to similar efforts [10], RaJa can handle web applications where JavaScript and a server-side programming language (for example PHP) are intermixed (see Figure 1.1).

- We enable RaJa in a set composed of four real-world web applications, namely WordPress, phpBB, phpMyAdmin and Drupal, which experience code-mixing between JavaScript and PHP. RaJa succeeds in randomizing 90,5% of the JavaScript contained in all four web applications (of more than half a million LoCs in total).

1.5 Thesis Outline

The remainder of this thesis is organized as follows. In Chapter 2 we analyze in detail the complete RaJa architecture. We evaluate RaJa in Chapter 3. In Chapter 4 we analyze real-world coding practices, while deploying RaJa in four popular web applications. We discuss important aspects and limitations in regards to RaJa in Chapter 5. We present related work in Chapter 6 and we conclude in Chapter 7.

2

Architecture

In this chapter we present in detail the RaJa architecture. We highlight all key components that compose the architecture and code modifications we have performed in existing software. RaJa builds on well-known and highly used open source projects like Mozilla and Apache. Finally, we give a short overview in a collection of tools we have specifically build for system administrators who want to utilize the RaJa framework.

```
<!-- Original Document.  -->
<html>
<script>
var s = "Hello World!";
if (true)
  document.getElementByName("welcome").text = s;
</script>
<div id="welcome"></div>
</html>

<!-- Randomized Document.  -->
<html>
<script>
var s0x78 = "Hello World!";
if (true)
  document0x78.getElementByName0x78("welcome").
    text0x78 = s0x78;
</script>
<div id="welcome"></div>
</html>
```

FIGURE 2.1: A typical RaJa example.

2.1 Overview

RaJa is based on the idea of Instruction Set Randomization (ISR) to counter code injections in the web environment. XSS is the most popular code-injection attack in web applications and is usually carried out in JavaScript. Thus, RaJa aims on applying ISR to JavaScript. However, the basic corpus of the architecture can be used in a similar fashion for other client-side technologies.

In a nutshell, RaJa takes as input a web page and produces a new one with all JavaScript randomized. A simple example is shown in Figure 2.1. Notice that in the randomized web page all JavaScript variables (emphasized in the Figure) are concatenated with the random token `0x78`. All other HTML elements and JavaScript reserved tokens (like `var` and `if`) as well as JavaScript literals (like `"Hello World"`, `"welcome"` and `true`) have been kept intact. The randomized web

page can be rendered in a web browser that can de-randomize the JavaScript source using the random token. RaJa needs modifications both in the web server and the web client, as is the case of many XSS mitigation frameworks [10, 20, 22, 31].

In order to perform the randomization, RaJa needs to run as a pre-processor before any other server-side language (like PHP, ASP, etc.) takes place. RaJa assumes that only the JavaScript stored in files[1] in the server is trusted. Randomizing all trusted JavaScript ensures that any code injections will not be able to execute in a web browser that supports the framework.

A sample work-flow of a RaJa request-response communication is as follows. The RaJa-compliant web browser announces that it supports the framework using an `HTTP Accept`[2] header. The web server in turn opens all files needed for serving the requests and randomizes each one with a unique per-request key. Typically, a request involves several files that potentially host JavaScript, which are included in the final document through a server-side language. For example PHP uses `require` and similar functions to paste the source of a document in the final web response. RaJa makes sure that all JavaScript involved is randomized. Finally, the web server attaches an `HTTP X-RAJA-KEY` header field which contains the randomization key. The RaJa-compliant web browser can then de-randomize and execute all trusted JavaScript. Any JavaScript source code that is not randomized can be detected and prevented for executing. The potential code injection is logged in a file.

We now look into details on how we enable RaJa in the server and client side, respectively.

[1] This assumption can be augmented to support JavaScript stored in a database if we introduce read-only tables (see discussion in Chapter 5).

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

2.2 Server Side

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

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

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

In order to enable RaJa in Apache web server we use two basic components: (a) an Apache module (`mod_raja.so`), which operates as content generator and (b) a library interceptor which handles all `open()` calls issued by Apache. Although RaJa can be used with any server-side technology, for the purposes of this thesis we use PHP. Thus, we have configured the RaJa-enabled web server to use PHP as an output filter. For all experiments in this thesis we use PHP acting as an output filter, but if someone prefers to use PHP as a module and not as a filter, two

Apache web servers can be used with the RaJa-enabled Apache acting as a proxy to the PHP-enabled one.

The RaJa Apache module handles initially all incoming requests for files having an extension of `.html`, `.js` and `.php`. This can be configured to support many other file types (see below the configuration details). For each request it generates a random key and places it to a shared memory placeholder. It then opens the file in order to fulfill the request. The call to `open()` is intercepted using the `LD_PRELOAD` [4] functionality, available in most modern operating systems, by the RaJa randomizer. The latter acts as follows. It opens the file and tries to identify all possible JavaScript occurrences. That is, all code inside a `<script>` tag, as well as all code in HTML events such as `onclick`, `onload`, etc. For every JavaScript occurrence a parser, based on the Mozilla SpiderMonkey [7] JavaScript engine is invoked to produce the randomized source. All code is randomized using the token which is retrieved from the shared memory placeholder. We analyze in more detail the internals of the SpiderMonkey-based parser below.

The randomized code is placed in a temporary file and the actual `libc_open()` is called with the pathname of the randomized source. Execution is transferred back to the Apache RaJa module. The module takes care for two things. First, it attaches the correct `Content-Length` header field, since the size of the initial file has possibly changed (due to the extra tokens attached to JavaScript source). Second, it attaches the `X-RAJA-KEY` header field to the HTTP response, which contains the token for the de-randomization process. The key is refreshed per request. All randomized code is contained in an internal memory buffer. This buffer is pushed to the next operating element in the Apache module chain. If the original request is for a PHP file, then the buffer will be pushed to the PHP output filter. It is possible that PHP will subsequently open several files while processing `require()` or similar functions. Each `open()` issued by the PHP filter

is also intercepted by the RaJa randomizer and the procedure is repeated again until all PHP work has been completed. The size of the final response has possibly changed again, due to the PHP processing. PHP takes care for updating the `Content-Length` header field.

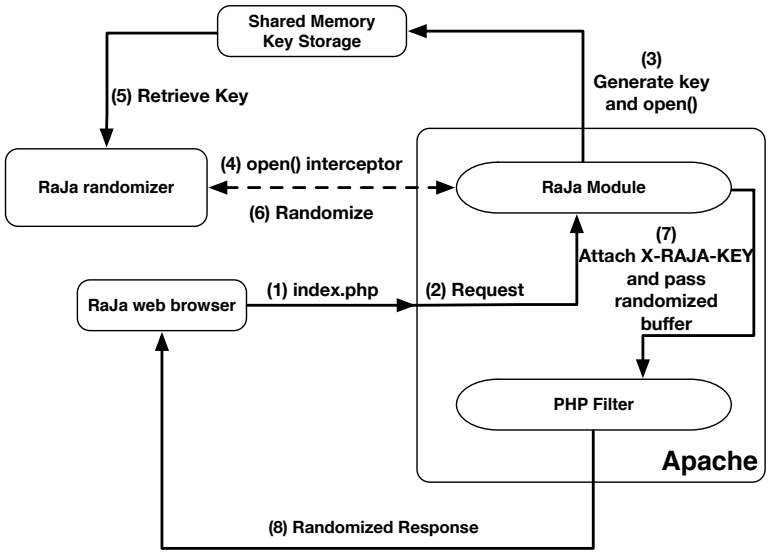

FIGURE 2.2: Schematic diagram of the RaJa architecture.

We present the control flow of the RaJa architecture in Figure 2.2 with all eight steps enumerated. We now proceed and present a step-by-step explanation of a RaJa-enabled request-response communication. In Step (1) the RaJa-enabled web client requests `index.php` from a RaJa-enabled web server. In Step (2) the request is forwarded to the RaJa module which in turn in Step (3) generates a key, stores the key in a shared memory fragment and opens the file `index.php`. In Step (4) the RaJa randomizer intercepts `open()` and in Step (5) it retrieves the key from the shared memory fragment. In Step (6) `index.php` is opened, randomized, saved to the disk in a temporary file and the actual `libc_open()` is called with the pathname of the just created file. In Step (7) control is transferred to the RaJa module which adds the correct `Content-Length` and `X-RAJA-KEY`

header fields. If the file is to be processed by PHP the buffer containing the randomized source is passed to the PHP filter. All `open()` calls issued from PHP will be further intercepted by the randomizer but we have omitted this in Figure 2.2 to make the graph more clear to the reader. Finally, in Step (8) the final document is served to the RaJa-enabled web browser.

2.3 Randomization

All JavaScript randomization is handled through a custom parser based on the SpiderMonkey [7] JavaScript engine. The RaJa parser takes as input JavaScript source code and it produces an output with all code randomized. For an example refer to Figure 2.1. The original SpiderMonkey interpreter parses and evaluates JavaScript code. In RaJa execution is disabled. Instead, all source is printed randomized with all JavaScript identifiers concatenated with a random token. This can be achieved by modifying the JavaScript lexical scanner of SpiderMonkey. Special care must be taken for various cases. We enumerate a few of them.

1. Literals. All literals, like strings and numbers, are parsed and directly pasted in the output in their original form.

2. Keywords. All keywords, like `if`, `while`, etc., are parsed and directly pasted in the output in their original form.

3. HTML comments. The original SpiderMonkey removes all comments before evaluation. The RaJa parser pastes all HTML comments in their original form.

4. Language mixing. Typically a web page has a mixture of languages such as HTML, PHP, XML and JavaScript. The RaJa parser can be configured to handle extra delimiters as it does with HTML comments and thus identify

```
1  <!-- Original Source.  -->
2  <?php if (user_exists($user)) { ?>
3    var message = <?php echo "Welcome" ?>;
4  <?php } else { ?>
5    var message = "Registration Needed.";
6  <?php } ?>
7
8  <!-- Randomized Source.  -->
9  <?php if (user_exists($user)) { }?>
10   var message0x78 = <?php echo "Welcome" ?>;
11 <?php } else { ?>
12   var message0x78 = "Registration Needed.";
13 <?php } ?>
```

FIGURE 2.3: Code mixing of JavaScript with alien languages. In this example PHP is used as an alien language example.

other languages, such as PHP, which heavily intermix with JavaScript. We further refer to these languages as *alien languages*.

We augment the RaJa parser to treat occurrences of alien languages inside JavaScript according to the following rules.

- *Rule 1.* An alien language occurrence is treated as a JavaScript identifier if it occurs inside a JavaScript expression.
- *Rule 2.* An alien language occurrence is treated as a JavaScript comment and is left intact if *Rule 1* is not applied.

We conclude to these basic two rules after investigating four popular and large, in terms of lines of code (LoCs), web applications. By manually checking how PHP is mixing with JavaScript, we observed that in the majority of the cases PHP serves as an identifier or literal inside a JavaScript expression (see line 3 in Figure 2.3). For a short example of how these two rules are applied refer to Figure 2.3. Consider that in line 3, Rule 1 is applied, while in lines 2, 4 and 6, Rule 2 is applied.

2.4 De-randomization

The de-randomization process takes place inside a RaJa-compliant browser. In our case this is Firefox with an altered SpiderMonkey engine. The modified JavaScript interpreter is initialized with the random token, taken from the `X-RAJA-KEY` header field. During the parse phase it checks every identifier it scans for the random token. If the token is found, the internal structure of the interpreter that holds the particular identifier is changed so as to hold the identifier de-randomized (i.e. the random token is removed). If the token is not found, the execution of the script is suspended and its source is logged as suspicious.

We take special care in order to assist in coding practices that involve dynamic code generation and explicit execution using the JavaScript's built-in function `eval()`. Each time a correctly randomized `eval()` is invoked in a script, the argument of `eval()` is not de-randomized. Notice that this is consistent with the security guarantees of the RaJa framework, since the `eval()` function is randomized in the first place and cannot be called explicitly by a malicious script unless the random token is somehow revealed. However, this approach is vulnerable to injections through malicious data that can be injected in careless use of `eval()`. For the latter case the RaJa framework can be augmented with tainting [32, 33, 37, 39].

2.5 Configuration and Administrator Tools

The RaJa framework can be configured with various dynamic options in order to be deployed in a web site. For example, it can be configured with the delimiters identifying an alien language.

```
# RaJa Configuration.
# Path to RaJa randomizer.
randomizer = /home/raja/framework/raja
# Files handled.
ext = php, html, js
# Meta-tokens.
extra-tokens = <?php|?>
```

FIGURE 2.4: A typical RaJa configuration file.

In Figure 2.4 we depict a sample configuration. It also includes a command-line tool-chain that can assist a web programmer or system administrator. These are the followings:

1. `raja`: Takes as input a token and a JavaScript source through standard input or a file and produces its randomized equivalent.

2. `raja-eval`: Takes as input a token and a randomized JavaScript source code through standard input or a file and evaluates it.

3. `raja-enforce`: Takes as input a token and a web page and identifies how many scripts can be randomized and how many cannot. The latter works as follows. The first processing unit of the tool takes as input the web page and attempts to identify all possible JavaScript source code occurrences. It first removes PHP and HTML comments and then searches for JavaScript. That is, all code inside a `<script>` tag, as well as all code in HTML events such as `onclick`, `onload`, etc. Notice the identified code may contain PHP language elements. The mixed JavaScript source code is stored for further processing. The second processing unit is the modified Mozilla SpiderMonkey JavaScript parser. For every stored source code from the previous phase, the parser is invoked to produce the randomized output. Apparently there are cases the tool fails to randomize the script and produces a syntax error.

2.6 Self-Correctness

In order to prove that the RaJa parser does not produce invalid JavaScript source we use the built-in test-suite of the SpiderMonkey engine. We first run the test-suite with the original SpiderMonkey interpreter and record all failures. These failures are produced by JavaScript features which are now considered obsolete. We subsequently randomize all tests, remove all E4X [17] tests because we do not support this dialect, re-run the test-suite with the `raja-eval` and record all failures. The failures are exactly the same. Thus, the modified SpiderMonkey behaves exactly as the original one in terms of JavaScript semantics.

3

Evaluation

In this chapter we evaluate RaJa. We measure the overhead RaJa imposes on the server and client side. We use the same evaluation techniques as those used in [10].

3.1 Server-Side Overhead

The most crucial part of RaJa performance is the server-side part. All web pages are examined for JavaScript source code and if an occurrence is found the RaJa randomizer performs a full-JavaScript parsing session. This process is repeated for every request (see Chapter 5 about caching issues). This is vital for the security guarantees of the framework, since the randomization key has to be refreshed in

every request. Otherwise, an XSS exploit can perform a request to reveal the key and then launch the actual attack.

In order to measure the overhead imposed on the server-side part, we request a set of web pages that embed a significant amount of JavaScript using the Apache Benchmark (`ab`) [1]. The sample of web pages is collected from the SunSpider [8] suite, which constitutes a collection of JavaScript benchmarks for measuring the performance of JavaScript engines. The suite 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.

We conduct two sets of experiments. For the first set we use `ab` over a Fast Ethernet (FE) network. We configure `ab` to issue 1000 requests for the heavy, normal and light web pages to stress a RaJa-enabled server. Then, we perform the same experiments using a vanilla Apache. Finally, we repeat all the above with the `ab` client running in a typical downstream DSL line.

Figure 3.1 summarizes the results for the case of the `ab` tool connecting to the web server through a FE connection. The RaJa server imposes an overhead that ranges from a few tens of milliseconds to about one hundred of milliseconds 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. In the normal case the delay is only a few milliseconds. However, in Figure 3.2 we present the same experiments over the DSL link. In the worst case (heavy) the server imposes a fixed delay of a few tens of milliseconds, like in the case of the Fast Ethernet setup. The overhead is negligible (less than a roundtrip in

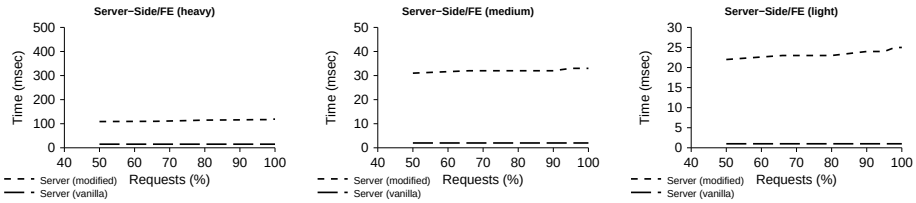

FIGURE 3.1: Server side evaluation using the Apache benchmark tool (`ab`) through a Fast Ethernet link.

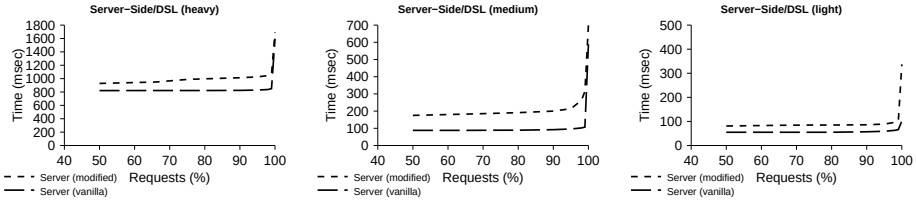

FIGURE 3.2: Server side evaluation when using the Apache benchmark tool (`ab`) through a DSL link.

today's Internet) since now the delivery overhead dominates. Same observations hold also for the request rate, presented in Table 3.1.

This drives us to conclude that RaJa imposes a fixed overhead of a few tens of milliseconds per page, which is not the dominating overhead.

Configuration	Heavy	Medium	Light
RaJa (r/sec)	1.08	5.55	12.07
Vanilla (r/sec)	1.22	7.25	18.14

TABLE 3.1: Requests served per second for the RaJa-enabled Apache and for the vanilla one, respectively.

3.2 Client-Side Overhead

The additional overhead in the client-side originates from the fact that a RaJa compliant web browser embeds a modified SpiderMonkey engine, which de-randomizes each JavaScript identifier while parsing. We expect this overhead to be relatively

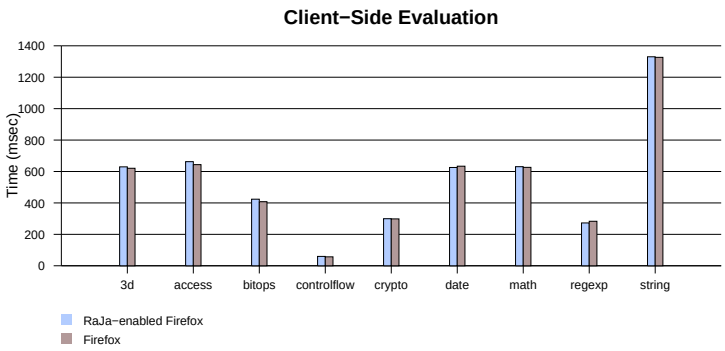

FIGURE 3.3: Client-side evaluation for a RaJa-enabled web browser using the SunSpider benchmarks.

small compared to the authentic SpiderMonkey engine, since it involves a string comparison, relatively to the size of the token, for every identifier parsed.

In order to evaluate the client-side overhead we use the SunSpider [8] suite, which includes a collection of JavaScript benchmarks. These tests are meant to stress the JavaScript engine of a web browser. The benchmarking is carried-out using JavaScript which in the RaJa case is also randomized and cannot be accounted. However, the code involved for accounting the benchmarks is significantly lesser than the code of the tests and thus the results are slightly deviate from the real overhead.

We plot the benchmark results for all nine different families of tests in Figure 3.3. As expected the RaJa -enabled web browser is slightly slower than the original Firefox.

4

Case Studies

In this chapter we test RaJa with popular web applications. We present our experiences from deploying the framework with existing source code, which is composed by multiple flavors of server-side and client-side code. We also highlight various coding idioms and practices we found.

4.1 Web Applications

Our analysis includes four popular web applications. These applications use PHP in order to produce JavaScript code dynamically. We now provide a short description for each one of these.

- **WordPress** [9]. WordPress is a popular blog engine based on PHP and MySQL. It is estimated that WordPress is used by over 202 million web sites. The source corpus includes PHP, HTML, XML, SQL and JavaScript code, mixed up in various ways. WordPress had many security vulnerabilities in the past and thus it is considered ideal for testing.

- **phpBB** [5]. A web application for developing interactive forums. It is written in PHP and it supports multiple database management systems and unlimited levels of sub-forums.

- **phpMyAdmin** [6]. This web application is intended to handle the administration of MySQL databases over a web front-end. Through the web interface a user can create, modify, or delete databases, tables, etc. phpMyAdmin contains a significant amount of JavaScript intermixed with PHP.

- **Drupal** [3]. Drupal is a content management system (CMS) written in PHP. It allows the system administrator to organize the content, automate administrative tasks and manage site visitors.

4.2 Methodology

We examine every file of all four applications source corpus. The `raja-enforce` tool, introduced in Chapter 2, is used to scan the source code and spot JavaScript snippets. That is, all code inside a `<script>` tag, as well as all code in HTML events such as `onclick`, `onload`, etc. All identifier tokens are randomized as they derive from the JavaScript syntax analysis. In other words this tool contains all the server-side functionality. Apparently there are cases the tool fails to randomize the script and produces a syntax error. This derives from the mixing between JavaScript and PHP. In other words the parser is confused when processing PHP language elements and tries to identify them as JavaScript language elements. This

means that the RaJa-enabled web server would have failed to serve the specific file. Apart from this, a syntax error can be produced in other cases where PHP is not involved. This occurs because the input the parser is given, may not be a complete JavaScript code but a fragment. For example the syntax analysis fails when a return statement exists out of a function block. This mostly occurs in code from HTML events. Every event has a default action. When an event handler is also defined (like `onclick`), the event handler can return a boolean to tell the browser whether or not to take the default action. Another example is when the parser has to deal with escaped string quotes which come from PHP string variable content extractions. In our implementation, such cases are successfully identified without producing any error messages. An example of the parser input is given in Figure 4.1. Successfully identified cases do not mean that the input is not mixed. In many cases PHP language elements contained in a script pass the syntax checking. This mostly occurs in JavaScript assignment expressions where the right value is a string constant containing PHP code. It is questionable if the rest of the PHP code has been modified wrongly due to the randomizer. We further proceed and check each file using PHP's lint mode to check the randomized files syntactically. All files succeed on passing the PHP syntax check.

```
1 <!-- Original Document.  -->
2 .... onsubmit="return checkPassword(this)"> ....
3 <!-- Parser Input  -->
4 //Start Of Input
5 return checkPassword(this)
6 //End Of Input
```

FIGURE 4.1: Parser input example from phpMyAdmin.

We manage to identify 163 scripts in all four web applications in which the tool fails to randomize them. These scripts would have made the RaJa-enable server to

fail. We further proceed and investigate each of the 163 mixed scripts manually. We manage to create a taxonomy with five different classes. We, finally, assign each script to the corresponding category. In the following section we describe each of the five categories. For the overall statistics of all web applications refer to Table 4.1 and 4.2. The third column of Table 4.2 specifies how many files include scripts. The fifth column indicates how many files include at least one mixed case.

Web Application	LoCs	Scripts	Non-mixed	Mixed
WordPress	143,791	187	136	51
phpBB	213,681	539	512	27
phpMyAdmin	178,447	263	183	80
Drupal	44,780	8	3	5
Total	580,699	997	834	163

TABLE 4.1: Summary of scripts that experience mixing in four real-world web applications.

Web Application	Files	With JavaScript	Pass	Fail
WordPress	279	48	28	20
phpBB	542	137	117	20
phpMyAdmin	304	65	28	37
Drupal	143	5	1	4

TABLE 4.2: Statistics for files contained in each web application.

4.3 Classification

The analysis of 163 different scripts, where JavaScript is mixed with PHP, produces the following five categories. For each category we provide an example from one of the four web applications that take part in the analysis. The example may be slightly simplified for presentation reasons.

- **Case 1.** *Partial injection of non-mixed JavaScript source using the PHP built-in function* `echo()`. This is the case where non-mixed JavaScript source code is injected in the web document via PHP, by using the built-in function `echo()`. RaJa cannot handle partial fragments of JavaScript, since the source must be first fully parsed and then randomized. It's hard for the parser to isolate portions of JavaScript injected using the PHP built-in `echo()` function and then re-assemble them in a snippet that can be successfully parsed. We present a example of a JavaScript snippet, which is injected using multiple calls to `echo()` in Figure 4.2. occurrences.

```
1  /* Case 1.  */
2  echo "<script>\n";
3  echo "document.write(...);"
4  echo "</script>\n";
5
6  /* Parser Input */
7
8  //Start Of Input
9  \n";
10 echo "document.write(...);"
11 echo "
12 //End Of Input
```

FIGURE 4.2: Example for Case 1: Partial injection of non-mixed JavaScript source using the PHP built-in function `echo()`.

- **Case 2.** *String concatenations.* In this case the final JavaScript code is generated by the concatenation of string literals and values of PHP variables. All dynamic generated strings are printed using the PHP built-in function `echo()`. The difficulty here occurs because both quotes (single and double) are used as part of a complex string concatenation operation. Randomizing the JavaScript part is hard in this case, since both quoting styles are signifi-

cant for both programming languages, JavaScript and PHP. In Figure 4.3 we present a string concatenation example.

```
/* Case 2.  */
$actions['quickedit'] =
        'onclick="commentReply.open(\''  .
        $post->ID                        .
        '\',\'edit\');"';

/* Parser Input */

//Start Of Input
commentReply.open(\'' .$post->ID. '\',\'edit\');
//End Of Input
```

FIGURE 4.3: Example for Case 2: String concatenation.

- **Case 3.** *Partial JavaScript code generation by PHP scripting blocks.* This is the most frequently occurring case of JavaScript and PHP intermixing. In this case PHP scripting blocks are put inside JavaScript. These scripts contain calls of `echo()`. After PHP is invoked, these blocks are evaluated and the final JavaScript source is generated. The parser fails the syntax analysis at the time it consumes the PHP scripting block. An example of this case is in Figure 4.4.

```
/* Case 3.  */
tinyMCEPreInit = {
 ...
 mceInit:{<?php echo $mce_options; ?>},
 ...
};
```

FIGURE 4.4: Example for Case 3: Partial JavaScript code generation by PHP scripting blocks.

- **Case 4.** *JavaScript code generation by using frameworks' meta languages.* In this case a framework uses a meta language to build JavaScript dynamically. This case exists only in phpBB. The framework generates dynamic content by transforming static HTML pages containing the meta languages elements. It loads the page, uses patterns to locate all meta language elements and then substitutes them with PHP code as the meta language semantics order. Secondly, the generated PHP code must be evaluated by using PHP function `eval()` to produce the final JavaScript code. In addition, the meta language is supposed to be more expressive than ordinary substitutions. It gives the programmer the opportunity to make easy conditional assignments. In Figure 4.5 we present an example which is used in order to assign a value in the `lang` and the `index` field. In such a case, isolating the JavaScript code is hard, since the complete source code is not known in advance. The parser fails at the time it consumes the meta language elements.

```
1  /* Case 4.  */
2  var RecaptchaOptions = {
3   lang  : {L_RECAPTCHA_LANG},
4   index :
5    <!-- IF $CAPTCHA -->
6          {$CAPTCHA}
7    <!-- ELSE -->
8          10
9    <!-- ENDIF -->
10 };
```

FIGURE 4.5: Example for Case 4: JavaScript code generation by using frameworks' meta languages.

- **Case 5.** *Markup injections.* Finally, the last case includes scripts where HTML special characters (like `&`) are injected in JavaScript expressions. These HTML characters are translated from a PHP filter before the script reaches the web browser. Before the translation the JavaScript expression is invalid. For example, the most frequently occurring case is when the sequence of HTML entities `&&` is translated to `&&`, which is the logical `AND` in a JavaScript expression. For an existing example of such a case, please refer to Figure 4.6. We are not aware of the goal of the web programmers that use this coding tactic.

```
1 /* Case 5.  */
2 onsubmit=
3   "return
4   (emptyFormElements(this, 'table')
5   &amp;&amp; checkFormElementInRange(
6   this, 'num_fields', ...);"
```

FIGURE 4.6: Example for Case 5: Markup injections.

Results. We now present the results from the classification of all 163 scripts of the four web applications. In Table 4.3 we list each application with all identified mixed scripts, categorized in each one of the five categories we present above. Notice that for all four application most of the mixed scripts fall in the third case. The meta-language case, Case 4, occurs only in phpBB. In addition, only three scripts fall in Case 5 and occurs only in phpMyAdmin. Case 1 cases are limited as well. Our observation suggests that there are only a few coding idioms, in order to mix PHP and JavaScript, that are used in real-world web applications. The dominant idioms is string concatenation, partial injection using PHP scripting blocks and custom meta-language technologies.

Web Application	Scripts	C1	C2	C3	C4	C5
WordPress	51	3	12	36	0	0
phpBB	27	1	0	0	26	0
phpMyAdmin	80	0	43	34	0	3
Drupal	5	0	2	3	0	0
Total	163	4	57	73	26	3

TABLE 4.3: Categorization of all mixed scripts in all four web applications.

4.4 Mixing Reduction

In this section we try to reduce the mixing between JavaScript and PHP code and successfully randomize JavaScript in failed cases of Section 4.3. This is possible by altering the mixed code or extending the server-side parser to support some individual cases. We propose code rewriting for Case 1 and Case 2 cases. We extend the parser for Case 3, Case 4 and Case 5 cases. We use both techniques for complicated Case 3 and Case 4 cases.

- **Case 1.** In this case it is easy to give an alternative coding idiom to avoid the parser failing. Instead of using PHP `echo()` calls to generate the code in the final document, the code can be injected as it is. The programmer can finalize the PHP scripting block , inject the JavaScript code and then start the PHP scripting block again. The alternative code is depicted in Figure 4.7.

```
1 /* Case 1 Alternative Idiom.  */
2 ... ?>
3
4 <script type='text/javascript'>
5         document.write(...);
6 </script>
7
8 <?php ...
```

FIGURE 4.7: Alternative approach for case 1.

- **Case 2.** This is the most difficult case to address. Consider that in our implementation the snippet should be modified to be syntactically correct by both the PHP interpreter and our JavaScript parser. Mix reduction in this case can be achieved by making less confusing use of quotes and by using as less as possible concatenation parts. The alternative code is depicted in Figure 4.8.

```
1 /* Case 2 Alternative Idiom.  */
2 $content = '\'' . $post . '\',\'edit\'';
3
4 $actions['quickedit'] =
5  'onclick=
6   "commentReply.open('.$content.');"';
```

FIGURE 4.8: Alternative approach for case 2.

- **Case 3.** In this case we try to address failed cases by both rewriting the code and extending the parser. The parser identifies the start and the end of a PHP scripting block in the syntax analysis, consumes it and then handles it as a valid JavaScript identifier. Recall the first rule from Chapter 2. However, there are cases where an identifier is not expected due to JavaScript semantics and the analysis fails again. Consider the case in line 4 of Figure 4.4. The curly brackets surrounding the PHP code denote an object inside. Thus, the JavaScript interpreter does not expect to parse a common identifier. The example of this case can be solved with a different coding practice suggested in Figure 4.9. Now the PHP scripting block is identified as JavaScript identifier.

```
1  /* Case 3 Alternative Idiom.  */
2  tinyMCEPreInit = {
3   ...
4   mceInit:
5     <?php
6        $mce_options_s = "{".$mce_options."}";
7        echo $mce_options_s;
8     ?> ,
9   ...
10 };
```

FIGURE 4.9: Alternative approach for case 3.

- **Case 4.** During the syntax analysis we use the same patterns phpBB uses to identify the meta language elements. After a successful identification the parser handles them as JavaScript identifiers, as above. In cases where the meta language elements are straight substituted this approach works. In cases where the meta language is quite complicated we propose an alternative writing to make it more simple. Consider the case of Figure 4.5 (lines 4-9). The JavaScript source cannot be parsed if the meta language code is removed, since the removal leaves two subsequent JavaScript identifiers. The latter produces a syntax error. The example of this case can be solved by giving a different coding practice suggested in Figure 4.10. This strategy makes code maintenance more difficult but helps the parser to successfully randomize the JavaScript. Now the meta language elements, {L_RECAPTCHA_LANG} and {CAPTCHA}, are treated as JavaScript identifiers. Notice that the second rule from Chapter 2 is applied in lines 2, 7 and 12.

```
/* Case 4 Alternative Idiom.  */
<!-- IF $CAPTCHA -->
var RecaptchaOptions = {
 lang: {L_RECAPTCHA_LANG},
 index: {CAPTCHA}
};
<!-- ELSE -->
var RecaptchaOptions = {
 lang: {L_RECAPTCHA_LANG},
 index: 10
};
<!-- ENDIF -->
```

FIGURE 4.10: Alternative approach for case 4.

- **Case 5.** In this last case the parser is extended to recognize HTML entities (like `&`) and simply consume and replace them in the syntax analysis.

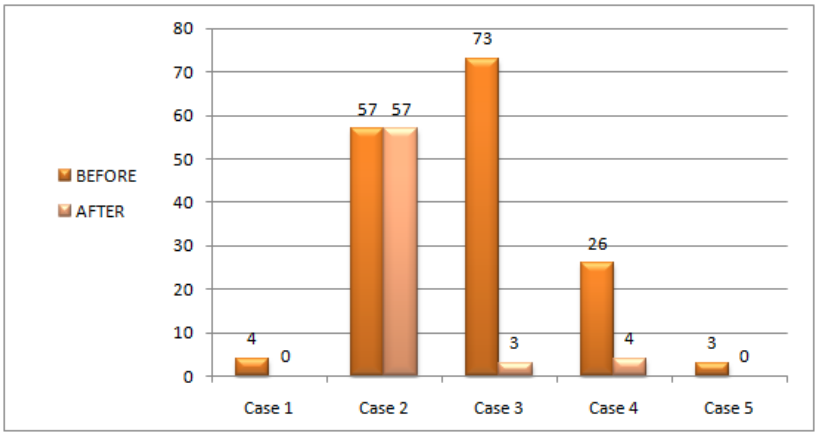

FIGURE 4.11: Script occurrences for each category for all four web applications before and after our alterations.

Results. In cases where the failed instances are few, like in Case 1, we rewrite the code as proposed. In addition we extend the parser semantics to successfully handle Case 3, Case 4 and Case 5. We then rerun the extended parser with the methodology described in Section 4.2. We further investigate the current failed

cases. In Figure 4.11 we plot the overall script occurrences we recorded for each of the five different cases before and after our interference. In Case 1 and Case 5 all failed cases are eliminated. In Case 3 and Case 4, the parser extensions managed to strongly reduce the failing rates. The few cases which remain need code rewriting in order to be addressed. In Case 2 the failed cases remain because neither code rewriting nor extensions are applied in the second run. In Table 4.4 we list each application with all identified mixed scripts in the second run, categorized in each one of the five categories. All remaining cases need code rewriting in order to be randomized. Notice that only phpMyAdmin needs a lot of code rewriting.

Web Application	Scripts	C1	C2	C3	C4	C5
WordPress	13	0	12	1	0	0
phpBB	4	0	0	0	4	0
phpMyAdmin	45	0	43	2	0	0
Drupal	2	0	2	0	0	0
Total	64	0	57	3	4	0

TABLE 4.4: Categorization of all mixed scripts in all four web applications in the second run.

5
Discussion

We now discuss some important aspects that we have not highlighted through out the thesis and are crucial for deploying RaJa in real systems.

5.1 JavaScript Templates in Databases

RaJa assumes that all legitimate JavaScript is stored in the web application's files. We consider that an attacker has not compromised the web server, since then an XSS attack is redundant. We also assume that incoming data from the network are not placed in files but in a database, since using the filesystem as a data storage for a web application is considered rather obsolete.

However, there is the possibility that a web application stores legitimate JavaScript code in a database. Since, RaJa randomizes only JavaScript stored in files, code fetched from a database is considered as a code injection. We are not aware of any real web application that stores legitimate JavaScript in databases. However, it is worth mentioning that RaJa can potentially co-operate with such a scheme provided that the tables storing the legitimate code have been marked as read-only and thus cannot contain both legitimate and malicious code.

5.2 Caching

RaJa refreshes the token which serves as the randomization key in every request. This is crucial for the security guarantees of the framework. If the key is not updated in every request then an attacker can potentially launch an XSS attack in two phases. In the first phase the attacker reveals the key and in the second phase performs the injection.

This strategy can break if a caching scheme is applied. For web pages that involve partial caching of page fragments, this strategy can even produce non-functional web pages. Cached fragments will have different randomization key from non-cached ones and the browser will not be able to de-randomize the complete JavaScript source correctly.

However, JavaScript is usually associated with dynamic pages, where caching is hard to be applied effectively. Making RaJa to cooperate with caching schemes is left as future work.

5.3 Cross-site request forgery

A Cross-site request forgery(CSRF) attack works as follows. The victim's web browser visits a malicious site. The site contains tasks which are usually linked

to specific urls. These urls request a specific task in a specific site. If the victim is logged into the site the task will be executed without her knowledge. The server that hosts the site thinks that because the request comes with the victim's cookies, the victim wanted to submit that form. Attackers often use HTML tags or JavaScript image object in order to execute CSRF attacks. These elements are embed into emails or websites.

Cross-site request forgery attacks differ from Cross-site Scripting attacks. CSRF occurs when the trust that a site has for the user is exploited. XSS occurs when the trust that a client has for the website or application is exploited. Cross-site request forgery attacks can be performed directly or by utilizing a Cross-site Scripting flaw.

In this thesis Cross-site request forgery attacks which occur in conjunction with an XSS attack are prevented. Some mitigation schemes for CSRF attacks can be found in [13, 23, 24].

6

Related Work

RaJa is based on Instruction Set Randomization (ISR) [25], which is a generic methodology to counter code-injection attacks. So far, the technique has been applied in native code [25] and SQL [16]. Keromytis discusses ISR as a generic methodology for countering code injections in [26] and he mentions that the technique can be potentially applied in XSS mitigation. However, to the best of our knowledge there has been no systematic effort towards this approach before. In addition there is no proposal for applying ISR directly to JavaScript.

Plenty of XSS mitigation scheme proposals exist. Dynamic tainting analysis is proposed in order to counter XSS attacks in [39]. In other similar approaches [31–33, 37] authors propose taint-tracking. RaJa does not rely on tainting at all. A

source-code based tainting technique [40] can certainly assist to cure cases where attacks happen due to careless use of `eval()`. Recall from Chapter 2 that in order to assist in dynamic code programming we take special treatment for explicit `eval()` calls. That is, we let `eval()` execute *unrandomized* code when it is called explicitly by the programmer. This is vulnerable to an XSS attack taking place with JavaScript code injected in the `eval()` call.

Noxes [27] is a client side approach which attempts to find and block unsafe URLs. XSS-GUARD [14] operates in the server side and performs input checking. We are not in favor of any techniques that perform content filtering, since we consider this process 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 [38] is a server-only approach which guarantees that untrusted content is not executed. The application server pre-renders the page and serves each web document in a form in which all dynamic content is correctly escaped to avoid possible code injections. However, Blueprint requires the web programmer to inject possible unsafe content (for example comments of a blog story) using a specific Blueprint API in PHP. Spotting all unsafe code fragments of a web application is not trivial. On the other hand, RaJa requires little assistance from the web programmer, since there are relatively few specific coding practices that affects it. Blueprint imposes a significant overhead compared to solutions based on native browser modifications, like RaJa.

Enforcing separation between structure and content is another prevention scheme for code injections [34]. The 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 RaJa, this technique cannot deal with attacks that are based on the content-sniffing algorithms of browsers [12] as well as attacks that modify the DOM structure using purely client-side code [28].

Last but not least, a series of modern and sophisticated attacks have been discovered by researchers lately, such as CSRF [13], XCS [15], CSV [36] and attacks that exploit the CSS cross-origin inclusion [30]. RaJa has not be designed specially to prevent any of the above attacks. However, plenty of them require the attacker to inject some JavaScript to carry out the attack in full. Thus, RaJa can prevent some particular instances of these attacks that require JavaScript injection, at least in part.

7
Conclusion

In this thesis we present RaJa. A complete framework for applying Instruction Set Randomization (ISR) [25] directly to JavaScript.

RaJa enforces source code randomization by taking care of code interference between JavaScript and other server-side programming languages. All source randomization is carried out using a popular JavaScript engine, Mozilla SpiderMonkey [7], augmented with two simple rules to account with code mixing between JavaScript and PHP. To test the effectiveness of our engine in randomizing complex sources, we deploy it in four popular large web applications. During our analysis we process more than half of a million of LoCs. We identify about 1,000 scripts, which contain 163 scripts, where JavaScript is intermixed with PHP and

markup. We manually investigate all 163 scripts and create a classification scheme of five distinct classes. Moreover, we try to address failed cases and reduce the mix by proposing alternative code idioms or extensions to the server-side parser. We further analyze the results. Only phpMyAdmin needs a lot of code rewriting in order to be RaJa-enabled.

Finally we evaluate RaJa in terms of performance. Our analysis suggests that (a) RaJa imposes a fixed overhead of a few tens of milliseconds per page, which is not the dominating overhead (less than a roundtrip in today's Internet), (b) imposes negligible overhead in the browser side.

Bibliography

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

[2] Cross-site scripting (XSS). http://en.wikipedia.org/wiki/Cross-site_scripting.

[3] Drupal: An open source content management platform. http://drupal.org/.

[4] LD_PRELOAD Feature. See man page of LD.SO(8).

[5] phpBB: One of the most widely used Open Source forum solution. http://www.phpbb.com/.

[6] phpMyAdmin: Application that handles the administration of MySQL over the World Wide Web. http://www.phpmyadmin.net/.

[7] SpiderMonkey (JavaScript-C) Engine. http://www.mozilla.org/js/spidermonkey/.

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

[9] WordPress Usage: 202 Million Worldwide 62.8 Million US. http://andrewapeterson.com/2009/09/wordpress-usage-202-million-worldwide-62-8-million-us/.

[10] E. Athanasopoulos, V. Pappas, A. Krithinakis, S. Ligouras, and E. P. Markatos. xJS: Practical XSS Prevention for Web Application Development. In *Proceedings of the 1st USENIX WebApps Conference*, Boston, US, June 2010.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] H. Bojinov, E. Bursztein, and D. Boneh. XCS: Cross Channel Scripting and Its Impact on Web Applications. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 420–431, New York, NY, USA, 2009. ACM.

[16] 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.

[17] E. ECMA. 357: ECMAScript for XML (E4X) Specification. *ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland*, 2004.

[18] 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.

[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] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] H. Lin-Shung, W. Zack, E. Chris, and J. Collin. Protecting Browsers from Cross-Origin CSS Attacks. In *CCS 10: Proceedings of the 17th ACM Conference on Computer and Communications Security*, New York, NY, USA, 2010. ACM.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

[36] P. Saxena, S. Hanna, P. Poosankam, and D. Song. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*.

[37] 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.

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

[39] 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.

[40] 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.