An Architecture For Enforcing JavaScript Randomization in Web2.0 Applications (Short Paper)

Elias Athanasopoulos, Antonis Krithinakis, and Evangelos P. Markatos

Institute of Computer Science,
Foundation for Research and Technology - Hellas
email: {elathan, krithin, markatos}@ics.forth.gr

Abstract. Instruction Set Randomization (ISR) is a promising technique for preventing code-injection attacks. In this paper we present a complete randomization framework for JavaScript aiming at detecting and preventing Cross-Site Scripting (XSS) attacks. RaJa randomizes JavaScript source without changing the code structure. Only JavaScript identifiers are carefully modified and the randomized code can be mixed with many other programming languages. Thus, RaJa can be practically deployed in existing web applications, which intermix server-side, client-side and markup languages.

1 Introduction

Cross-Site Scripting (XSS) extends the traditional code-injection attack in native applications to web applications. It is considered as one of the most severe security threats over the last few years [14]. One promising approach for dealing with code-injection attacks in general is Instruction Set Randomization (ISR) [8]. 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* [9]. So far, the technique has been applied in native code [8] and SQL [4]. Architectures inspired by ISR for countering XSS attacks has been also proposed, like Noncespaces [6] and xJS [3]. 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.

In this paper we present RaJa, which applies randomization directly to JavaScript. We modify a popular JavaScript engine, Mozilla SpiderMonkey [2], to carefully modify all JavaScript identifiers and leave all JavaScript literals, expressions, 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 server-side scripting language, like PHP, with well defined starting and ending delimiters.
- Server-side scripting elements when mixed up with JavaScript source act as JavaScript identifiers or literals in the majority of the cases.

```
1  <!-- Original Document.  -->
2  <html>
3  <script>
4  var s = "Hello World!";
5  if (true)
6    document.getElementByName("welcome").text = s;
7  </script>
8  <div id="welcome"></div>
9  </html>
10
11 <!-- Randomized Document.  -->
12 <html>
13 <script>
14 var s0x78 = "Hello World!";
15 if (true)
16   document0x78.getElementByName0x78("welcome").
17     text0x78 = s0x78;
18 </script>
19 <div id="welcome"></div>
20 </html>
```

Fig. 1. A typical RaJa example.

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, mixed up with JavaScript, PHP and markup. 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.

2 Architecture

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 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

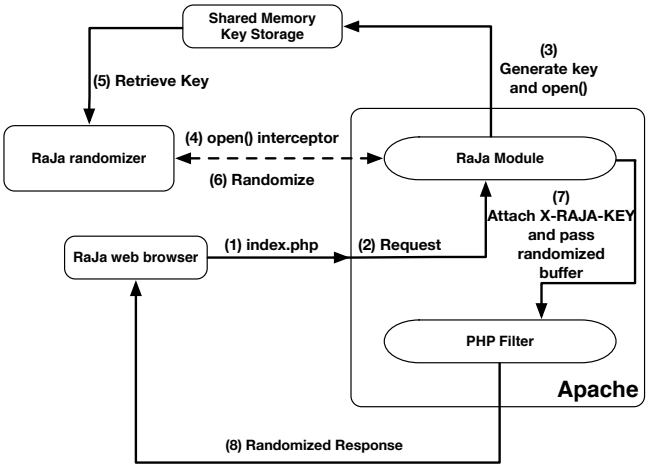

Fig. 2. Schematic diagram of the RaJa architecture.

and the web client, as is the case of many anti-XSS frameworks [7, 11, 6, 3]. In order to perform the randomization, RaJa needs to run as a pre-processor before any other server-side language (like PHP) 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 derandomize 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. In order to enable RaJa in a web server we use two basic components: (a) an Apache module (`mod_rjs.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 paper we use PHP. Thus, we have configured the RaJa-

[1] This assumption can be augmented to support JavaScript stored in a database if we introduce read-only tables.

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

```
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 } ?>
```

Fig. 3. Code mixing of JavaScript with alien languages. In this example PHP is used as an alien language example. In line 3, Rule 1 is applied, while in lines 2, 4 and 6, Rule 2 is applied.

enabled web server to use PHP as an output filter. For all experiments in this paper 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` [1] 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 [2] 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.

We present the control flow of the RaJa architecture in Figure 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 to make the graph more clear to the reader. Finally, in Step (8) the final document is served to the RaJa-enabled web browser.

Randomization. All JavaScript randomization is handled through a custom parser based on the SpiderMonkey [2] 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 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. 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 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 [10]. 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 3). For a short example of how these two rules are applied refer to Figure 3.

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 [15, 12, 13].

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 [5] tests because we do not support this dialect, re-run the test-suite with the `raja-eval` (a tool capable in executing randomized source) 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.

Acknowledgements. Elias Athanasopoulos, Antonis Krithinakis and Evangelos P. Markatos are also with the University of Crete. Elias Athanasopoulos is funded by the Microsoft Research PhD Scholarship project, which is provided by Microsoft Research Cambridge. The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 257007.

References

1. LD_PRELOAD Feature. See man page of LD.SO(8).
2. SpiderMonkey (JavaScript-C) Engine. `http://www.mozilla.org/js/spidermonkey/`.
3. Elias Athanasopoulos, Vasilis Pappas, Antonis Krithinakis, Spyros Ligouras, and Evangelos P. Markatos. xJS: Practical XSS Prevention for Web Application Development. In *Proceedings of the 1st USENIX WebApps Conference*, Boston, US, June 2010.
4. Stephen W. Boyd and Angelos D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, pages 292–302, 2004.
5. E. ECMA. 357: ECMAScript for XML (E4X) Specification. *ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland*, 2004.
6. Matthew Van Gundy and Hao 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, February 8-11, 2009.
7. Trevor Jim, Nikhil Swamy, and Michael 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.
8. G.S. Kc, A.D. 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.
9. Angelos 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.
10. Antonis Krithinakis, Elias Athanasopoulos, and Evangelos P. Markatos. Isolating JavaScript in Dynamic Code Environments. In *Proceedings of the 1st Workshop on Analysis and Programming Languages for Web Applications and Cloud Applications (APLWACA), co-located with PLDI*, Toronto, Canada, June 2010.
11. 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, February 8-11, 2009.
12. S. Nanda, L.C. 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.
13. Anh Nguyen-tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of the 20th IFIP International Information Security Conference*, pages 372–382, 2005.
14. SANS Insitute. The Top Cyber Security Risks. September 2009. `http://www.sans.org/top-cyber-security-risks/`.
15. 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, February 8-11, 2009.