An Empirical Study of the Framework Impact on the Security of JavaScript Web Applications

Ksenia Peguero
George Washington University
Washington, D.C., USA
kseniad@gwu.edu

Nan Zhang
Pennsylvania State University
University Park, PA, USA
nan@ist.psu.edu

Xiuzhen Cheng
George Washington University
Washington, D.C., USA
cheng@gwu.edu

ABSTRACT

Background: JavaScript frameworks are widely used to create client-side and server-side parts of contemporary web applications. Vulnerabilities like cross-site scripting introduce significant risks in web applications.

Aim: The goal of our study is to understand how the security features of a framework impact the security of the applications written using that framework.

Method: In this paper, we present four locations in an application, relative to the framework being used, where a mitigation can be applied. We perform an empirical study of JavaScript applications that use the three most common template engines: Jade/Pug, EJS, and Angular. Using automated and manual analysis of each group of applications, we identify the number of projects vulnerable to cross-site scripting, and the number of vulnerabilities in each project, based on the framework used.

Results: We analyze the results to compare the number of vulnerable projects to the mitigation locations used in each framework and perform statistical analysis of confounding variables.

Conclusions: The location of the mitigation impacts the application's security posture, with mitigations placed within the framework resulting in more secure applications.

CCS CONCEPTS

• **Security and privacy → Web application security**; • **Software and its engineering →** *Development frameworks and environments*; Software defect analysis;

KEYWORDS

JavaScript security; web security; web frameworks; framework analysis; template engines; cross-site scripting

1 INTRODUCTION

Many security vulnerabilities in modern web applications see common occurrences on numerous real-world websites, as demonstrated by studies like the Open Web Application Security Project (OWASP), which measures the most common vulnerabilities and releases their Top Ten list every few years [23]. There has been substantial research on the problems of discovering vulnerabilities [14, 18, 20, 21, 30, 33] and preventing them through user-facing solutions, e.g., web application firewalls (WAFs), browser plugins, and other instrumentation like NoScript for Firefox [19], XSS Filter for Internet Explorer [8], DexterJS [24], etc.

Somewhat surprisingly, there has been relatively little research on the other side of vulnerability prevention - i.e., how to properly design tools and instruments for developers, so as to reduce the likelihood for them to make mistakes and incur vulnerabilities in their web applications. In the current state of practice, when developers are security-aware and know of a potential vulnerability they may be introducing in their code, they can address it in several ways. Consider a generic injection vulnerability (which could be an SQL injection, a code injection, an operating system injection, a cross-site scripting (XSS), etc.) as an example. The different options available for a developer to mitigate the risk of introducing injection vulnerabilities are:

- **L0 - No mitigation** in place. This provides a base case of lack of any protection and the presence of a vulnerability;
- **L1 - Custom function**, such as a sanitization or filtering routine, that is written by developers and is included in their own code;
- **L2 - An external library** that provides a sanitization function and is called from developers' code;
- **L3 - A framework plugin**, similar to an external library, that is a third-party code used by developers, but it integrates tightly with the framework;
- **L4 - Built-in mitigation control**, that is implemented in the framework as a function or feature providing protection out of the box.

The question we would like to study is *which of these different mitigation levels produce more secure code*? To study this problem, we selected one of the most common vulnerabilities that has been present in the OWASP Top 10 since 2003 [12] and is still in the top 3 issues in OWASP Top 10 2017 [22] – cross-site scripting (XSS). We decided to focus on JavaScript frameworks, both client-side and server-side, as the issue of XSS assumes vulnerable JavaScript running in the browser.

Typically, an XSS vulnerability occurs when untrusted user input is sent to the output (a web page) without prior validation or encoding. Most contemporary frameworks have some built-in protection for this scenario. A common protection is using a template engine to process HTML pages that employs an encoding function replacing raw HTML tags with HTML entities that are not interpreted as tags by the browser. However, there are scenarios where developers do need to keep certain HTML tags in user input. For example, a blog application often requires rich text editing of posts and comments, so users can highlight text in bold, italics, or use emoticons (basically, images). Content management systems

WWW '18 Companion, April 23–27, 2018, Lyon, France

ACM ISBN 978-1-4503-5640-4/18/04.
https://doi.org/10.1145/3184558.3188736

(CMSs) also have similar requirements for using rich text elements in the pages. When developers face this requirement, there are several approaches they can take which correspond to the different levels of mitigation introduced above:

- Turn off the built-in output encoding and allow all HTML tags in the user input for certain fields, such as a post body in a blog application. From our experience, this is the most common approach and it leads to the greatest number of XSS vulnerabilities, since turning encoding completely off allows any dangerous HTML tags, including scripts. No mitigation technique is used, which corresponds to level L0.
- Turn off the built-in output encoding function and write their own sanitization routine that will either allow only "safe" HTML tags (whitelisting) or disallow any "dangerous" HTML tags (blacklisting). Writing custom sanitization functions yields subpar solutions and eventually leads to vulnerabilities in the code, as blacklists can often be bypassed and developers may create insecure whitelists. This approach corresponds to level L1.
- Turn off the built-in output encoding function and instead use an existing third-party sanitization function. Such an approach may result in secure applications, if the third-party function itself is secure and easily integrates with the framework. Some third-party solutions require additional configuration to run smoothly with the framework. Examples of such solutions in server-side JavaScript are the *markdown* library [17], presenting text in a wiki-format instead of HTML, and the *sanitizer* library [28], based on the Google Caja project. These solutions are rather complex and we did not see them commonly used in the projects we analyzed. This approach corresponds to level L2.
- Similar to using third-party sanitization functions, some frameworks may include plugins that enhance the functionality of the framework. However, for the use case studied in this paper and the JavaScript frameworks examined, we did not find any plugins that would correspond to level L3.
- Instead of completely turning off the built-in output encoding function, use a sanitization function provided by the framework that only returns a safe subset of HTML, preventing any XSS. This approach corresponds to level L4.

Therefore, these mitigation levels demonstrate the distance of the solution from the framework, where L1 is farthest away and not connected to the framework at all, and L4 is the closest, since the solution is built into the framework itself. We would like to study the impact of each of these solutions on the number of mistakes a developer makes and, correspondingly, the number of vulnerabilities the developer introduces in the code. Our **hypothesis** is that *the closer the mitigation is located to the framework itself, the fewer vulnerabilities the code will have*, since the framework will provide an out-of-the-box solution and a developer would not need to make additional decisions on how to implement a mitigation using a third-party or a homebred function each time user input needs to contain HTML tags.

2 APPROACH

2.1 Template Engines

Using *templates* or *template engines* is a popular mechanism for JavaScript frameworks to separate server-side data from client-side code. Naturally, it can be used to prevent XSS as well. For our study, we selected the three most popular template engines used by JavaScript applications – Jade/Pug, EJS, and AngularJS (version 1.2 or older, not Angular 2.X) templates. All these templates have some protection from XSS built into them:

- In Jade/Pug, any variable put into curly braces is automatically HTML-escaped, as well as a variable assigned to a tag using the equals sign (e.g., h1=title).
- EJS uses HTML format with additional syntax. Any variable included using the <%=variable%> tag will be automatically HTML-escaped.
- AngularJS performs escaping automatically for any variable included in curly braces and for the right context (HTML, URL, CSS, etc.) without requiring a developer to use special syntax. Therefore, it performs automatic contextually-aware escaping [1].

While all three frameworks have built-in protection from XSS, we examined the specific use case described in the Introduction section, where a developer needs to output user inputs but keep some HTML tags in it. That is, the application must output "safe" HTML by sanitizing the incoming user input.

Jade/Pug has the functionality to maintain HTML in user input. This is accomplished by using an exclamation point syntax (!value or tag!=value). However, Jade/Pug does not automatically perform sanitization of the output and the user input would be reflected as is. To output only the safe subset of the user input, developers need to either call an external function to perform the needed sanitization (L2) or write a custom function (L1). Jade/Pug does not have any framework plugins that would do the safe-HTML sanitization (L3).

EJS provides a functionality to keep raw HTML in user input with the <%- value %> syntax. However, similar to Jade/Pug, it does not include any sanitization of the output to keep the safe HTML subset in it. Since it does not have any plugins to do the safe-HTML sanitization (L3), developers also have to either write a custom function (L1) or use a third-party library (L2).

AngulaJSr, on the other hand, has a built-in feature that automatically outputs a safe subset of HTML when HTML content needs to be maintained in the user input. That is done by using a different tag in the AngularJS template (`ng-bind-html` vs. `ng-bind` or the curly-brace markup). When a developer needs to include some HTML tags in the user input, the developer uses a built-in framework feature (L4); and the output then contains "safe" HTML tags and the application is still protected from XSS. The most common way to output unsafe HTML in AngularJS is to explicitly turn off the sanitization by calling the $sce.trustAsHtml() function on the variable content and use `ngBindHtml` directive in the template. Another way is to override the $sceDelegate() service by configuring the $sceDelegateProvider(). However, this approach is used extremely rarely, therefore we concentrate on the first approach (the use of $sce.trustAsHtml()).

Consequently, our goal is to compare applications written in Jade/Pug, EJS, and AngularJS and the number of vulnerabilities they have in the case when an application needs to output user-provided HTML in a safe manner. To do this, we need to get a good sample of applications that require outputting HTML tags safely in the user input. We decided to concentrate on blogs and content management systems (CMS) since they both satisfy this requirement and represent popular needs in practice. We used GitHub to obtain a set of applications written using each framework.

2.2 Selection Criteria

As mentioned above, we concentrated on full-stack JavaScript applications available on GitHub. Our search conditions included the following: (1) application type is blog or CMS (in terms of functionality), (2) server-side technology is Express.js and Node.js, and (3) client-side technology is Jade/Pug, EJS, or AngularJS.

Further, we refined our search to include contemporary and more popular applications by introducing several specifically designed filtering conditions, including (4) last commit date is no later than 2013; (5) a repository has at least 1 star ("stars" in GitHub mark how popular the application is); (6) the language of the repository is JavaScript, HTML, CSS, TypeScript (since GitHub only allows to specify one language, and developers do not always select JavaScript, even when it is a JavaScript application)

Based on these six selection criteria we identified 65 Jade/Pug projects, 54 EJS projects, and 51 AngularJS project.

2.3 Analysis Pipeline

For each type of template engine we created the following analysis pipeline: (1) built a parser or extended an existing open source parser for the template engine; (2) built an analyzer or extended an existing open source analyzer for the template engine; (3) extended the analyzer ruleset to automatically identify the XSS vulnerability. Our source code and modifications to the open source projects can be found at [15].

After identifying the subset of applications for each framework we executed the following steps for each group of applications:

(1) automatically downloaded from GitHub information about project owners (developers) and the template files needed for analysis (.jade, .ejs, .html, depending on the template engine);
(2) ran the parser-analyzer pair for that template engine on the downloaded files and recorded the reported potential vulnerabilities;
(3) manually analyzed the automatically reported vulnerabilities in each project and filtered out false positives;
(4) performed statistical analysis of results.

2.4 Parsers and Analyzers

Jade/Pug. For Jade/Pug templates we used two open source packages utilized by a popular JavaScript linting utility ESLint [6]: pug-lexer [25] and pug-parser [26]. We extended the pug-lexer package with a rule that analyzes unescaped handlebars syntax, for example, !{variable}. This rule outputs lexemes with the type "interpolated-code" and with an attribute "mustEscape: false". Then we created a script that combines our improved pug-lexer with pug-parser, similar to how eslint-plugin-pug operates [7]. With our modifications, the pug-parser adds the "Code" token with the attribute "mustEscape: false" into the produced AST for all elements that were not automatically escaped by the template engine. These elements constitute potential XSS vulnerabilities[1], which are then reported using our jadeFilesAnalyzer script.

EJS. For EJS templates we looked at a few open source projects, such as fis-parser-ejs [9], ejs-lint [5], and then decided to extend the EJS core project [4], because it offers most versatile access to the generated document structure. We added a new model to EJS that builds a list of unescaped elements from the parsed EJS template, that constitute potential XSS vulnerabilities. These elements are then reported using our ejsFilesAnalyzer script.

AngularJS. For AngularJS templates we decided to extend the ESLint tool [6] by adding a specific rule called "trust_Angular" that identifies calls to AngularJS $sce service that return unescaped input, such as $sce.trustAs() and $sce.trustAsHtml(). To programmatically run the modified ESLint code, we used the Node API for ESLint [2] and modified the code to only report one type of finding to minimize the noise in the output. The potential XSS vulnerabilities are then reported using our angularFilesAnalyzer script.

Table 1: Accuracy of Automated Analysis

Template Engine	Vulnerabilities discovered automatically	True positives	Accuracy Rate
Jade/Pug	212	72	33.96%
EJS	140	96	68.57%
AngularJS	26	12	46.15%

3 RESEARCH FINDINGS

3.1 Limitations

To ensure the accuracy of our results, we performed manual analysis of the automatically reported potential vulnerabilities. If we compare the numbers of vulnerabilities discovered by our automated analysis and by the manual analysis, we can see that some of the template types yield a higher accuracy rate, while others have a lower accuracy rate, demonstrating the fact that some template formats are easier to parse automatically, while others present more complex structures (see Table 1).

The accuracy rates show that the fully automated analysis of applications (without manual triage) does not provide exact results. We identified the following three reasons for that:

- *Inaccuracy of the parsing tools.* While working on modifying the open source engines for parsing Jade/Pug and EJS templates we improved them by catching several types of exceptions and adding logic to parse corner cases.
- *Lack of dataflow analysis.* All the tools used in this research do not perform any dataflow analysis of JavaScript. In some

[1]Here and further we call the reported findings "potential vulnerabilities" because our tools may include false positives and, therefore, some reported findings may not be true vulnerabilities.

Table 2: Number of Vulnerable Projects Based on Template (After Manual Analysis)

Template Engine	Num. of Projects	Num. of Vulnera-bilities	Num. of Vuln. Projects	% of Vuln. projects
Jade/Pug	65	72	25	38.46%
EJS	54	96	23	42.59%
AngularJS	51	12	6	11.76%

cases our tools will flag a sink function that does not perform proper encoding. However, after manually analyzing the dataflow, we discovered that the source of data is a static file, such as a configuration file or a hard-coded blog post. Data from that source cannot be influenced by an attacker, and therefore, the finding is a false positive. This result is in line with the findings of other researchers stating that dataflow analysis improves the accuracy of automated security defects discovery [27].

- *Use of third-party libraries to perform HTML sanitization.* Since EJS and Jade/Pug do not have built-in functionality to sanitize HTML output, some developers do use third-party libraries, such as *markdown*, that is, they use the solution level L2. Since our tools do not perform dataflow analysis and do not have any knowledge of sanitization routines, they would flag variables that have been previously sanitized with *markdown* as sources of taint, resulting in a false positive. However, such cases were very rare in our set of applications.

Based on these findings, we conclude that any future static analysis of JavaScript applications vulnerabilities must include manual verification of results. Although dataflow analysis would improve the accuracy of the automated analysis, automated dataflow in JavaScript remains a challenging problem, due to the flexibility of the language. Specifically, JavaScript is dynamically and loosely typed, uses prototypes to implement inheritance, and has a very forgiving syntax. The problem of automated dataflow analysis in JavaScript is currently being explored by several researchers [16, 29].

3.2 Findings

After validating all findings with manual analysis, we counted the number of XSS vulnerabilities in each subset of applications, based on the template engine used (see Table 2). The 65 Jade/Pug projects contained a total of 72 vulnerabilities that occurred in 25 projects, resulting in 38% of all Jade/Pug applications being vulnerable. Out of 54 EJS projects, 23 contained at least one vulnerability, with the total number of vulnerabilities equal to 96. Thus 43% of all EJS applications were vulnerable to cross-site scripting. Out of 51 AngularJS projects 6 had vulnerabilities, with a total of 12 vulnerabilities discovered, resulting in only 12% of projects being vulnerable.

As we mentioned in section 2, all three frameworks do not have any plugins that support safe-sanitization (L3), and Jade/Pug and EJS also do not have any built-in controls for safely outputting HTML on the page (L4). Therefore, in EJS and Jade/Pug developers are bound to either write custom sanitization routines (mitigation level L1) or use third-party libraries (mitigation level L2). Below is an example of using the *markdown* library to prevent XSS attacks found in one of the Jade/Pug applications we analyzed:

```
if posts.length > 0
  each post in posts
    div.post
      h1.post-title= post.title
      article.post-content!=
            markdown(post.content)
else
  p No New Post
```

In this case, the post.content is returned in an insecure way using the "!=" syntax. However, the value of post.content is already sanitized with *markdown*. Nevertheless, the majority of the analyzed applications did not use any sanitization routines. For example, the application below returned the value of article.body without any sanitization using the insecure !{} syntax.

```
br
textarea(name='body', class='form-control',
  rows=10, id='editArticleBody')
  !{article.body}
script.
  CKEDITOR.replace('editArticleBody');
br
```

These examples demonstrate that using mitigation level L2 requires extra steps taken by a developer. On the other hand, the AngularJS template engine has L4 mitigation implemented – it outputs a safe subset of HTML as a built-in feature, which does not require any additional work performed by a developer. This is demonstrated in one of the AngularJS applications we analyzed, where the content of post.title and post.description will be automatically sanitized.

```
<div class="media-body">
  <h4 class="media-heading">
    <a href="{{post.slug}}">{{post.title}}</a>
  </h4>
  <p>{{post.description}}</p>
</div>
```

The results of our study demonstrate that when a framework has a security control built-in (mitigation level L4), the percentage of vulnerable projects written using this framework is considerably lower (12% in AngularJS vs. 38% in Jade/Pug or 43% in EJS). Although security controls that perform sanitization based on a safe HTML subset can be used with Jade/Pug and EJS as external third-party libraries (for example, using *markdown* [17] or Google Caja Sanitizer [28]), they are not part of the framework itself and require additional effort on the developers' side to implement the control as level L2. When such mitigations are built into the framework itself, as defined in the L4 approach, the applications created on top of the framework tend to be inherently more secure. Our data proves our hypothesis: the closer the implementation of the security control is to the framework itself, the fewer vulnerabilities the applications would have.

Table 3: P-values for Confounding Variables

Criteria	p-value
Developer's overall experience	0.319279
Developer's JavaScript experience	0.132049%
Project size	0.431335%
Project popularity	0.200649%
Project reuse	0.211615%
Template engine	0.001021%

After obtaining these results we also ran ANOVA statistical tests on the application sample that we selected. We performed confounding variables analysis to ensure that the numbers of vulnerabilities we obtained were not a result of a different factor. We analyzed the following factors: 1) developer's overall experience (measured by the number of projects on GitHub); 2) developer's JavaScript experience (number of JavaScript-related projects on GitHub); 3) project size; 4) project popularity (number of stars); project reuse (number of forks). ANOVA tests for each of these factors demonstrated that none of these factors have statistically significant impact on the number of vulnerabilities that the projects have. The only factor that did show statistically significant difference was the type of the template engine used by the application, and consequently, the choice of the mitigation level: L4 vs. L1 or L2 (see Table 3).

Based on our hypothesis, proved by data, our recommendation to the framework developers and maintainers is that security controls must be implemented into the framework itself and also must be turned on by default, if possible. Having security controls implemented as plugins or as third-party libraries makes them less likely to be used at all or used correctly be the developers. If we look at XSS vulnerability specifically, AngularJS implements output encoding for all fields out of the box (using the `ngBind` directive or double curly braces syntax). For fields that need to output HTML constructs AngularJS turns on the safe HTML sanitization by default, when the `ngBindHtml` directive is used. And only when developers need to output all HTML without any sanitization or filtering, they have to explicitly call the `$sce.trustAsHtml()` function and then send its output to the `ngBindHtml` directive. Therefore, the framework protects applications from XSS and developers from making mistakes inherently. These recommendations for creating secure frameworks should be used not just for XSS protection, but for other vulnerabilities, such as cross-site request forgery (CSRF) and different injection issues (SQL injection, OS command injection, mail injection, etc.).

The main message to developers choosing a framework for their applications is to evaluate the security of the framework by the number of mitigations or security controls already build into the framework. If a framework inherently implements protections from injection issues, CSRF, XSS, authentication and authorization controls, and these features have secure default settings, i.e. they are turned on by default, it is highly likely that applications developed on top of these frameworks will have fewer vulnerabilities.

4 RELATED WORK

Previous work on the security of JavaScript applications and frameworks concentrated in the following three areas.

JavaScript Security. Studying security of JavaScript applications themselves regardless of the frameworks used often focused on client-side JavaScript, identifying JavaScript inclusions, dynamic generation, and other bugs [18, 20, 30, 33]. Not much attention has been paid to the security of server-side JavaScript code written with frameworks like Node.js or Express.js.

JavaScript Analysis and Testing. Saxena and Li researched symbolic execution approach for testing JavaScript code [16, 29]. Artzi and Jensen analyzed automated testing of client-side JavaScript [3, 13]. However, these approaches did not concentrate on the security aspects.

Frameworks Analysis with security focus includes analysis of frameworks for other languages, not including JavaScript. For example, J. Wenberger analyzed XSS sanitization in web frameworks in PHP, Ruby, Python, Java [31, 32]. Previous analysis of JavaScript frameworks looks at performance, quality, and documentation, but not at security features [10, 11].

Therefore, to the best of our knowledge, this is the first study investigating the impact of a framework choice on the security of a full-stack JavaScript application.

5 CONCLUSION

In this paper, we considered four possible levels of mitigation for XSS vulnerabilities that can be implemented in web applications. We focused our analysis on JavaScript frameworks, selecting three popular template engines implementing security controls at different levels: Jade/Pug and EJS applications require controls at level L1 (custom function) and L2 (third-party library), where AngularJS has the control at level L4 (built-in feature). We analyzed how the choice of the framework and, in turn, the level of mitigation control, influences the number of XSS vulnerabilities found in open source projects. We used statistical analysis to confirm that no other confounding variables affected the produced results.

The results we achieved demonstrate that the level of the mitigation control has a significant impact on the security of the application. When the mitigating control is built into the framework itself, the applications built on top of that framework have a much lower rate of vulnerabilities. In the use case studied in this paper, i.e., when a developer needs to output only "safe" HTML tags in user input (in order to avoid XSS), only 11.76% of analyzed AngularJS projects had XSS vulnerabilities, compared to 38.46% of Jade/Pug projects and 42.59% of EJS projects.

For the use-case studied in this paper, our preliminary results suggest that framework maintainers should consider building security controls into their frameworks to improve the overall security of applications, instead of putting the burden of choosing security controls on the developers. Nevertheless, our results cover only one common web vulnerability (XSS) and only in a rather specific scenario. To demonstrate this point on a wider range of applications, vulnerabilities, and security controls, and to make broader conclusions on how the choice of a framework affects the security of an application, we need to perform broader research that covers

other security vulnerabilities and a broader range of mitigation strategies.

REFERENCES

[1] AngularJS. 2017. Documentation: $sce. (2017). Retrieved October 15, 2017 from https://docs.angularjs.org/api/ng/service/$sce

[2] ESLint Node.js API. 2017. (2017). Retrieved October 17, 2017 from http://eslint.org/docs/developer-guide/nodejs-api

[3] S. Artzi, J. Dolby, S. Jensen, A. Moeller, and F. Tip. 2011. A framework for automated testing of JavaScript web applications. In *Proc. of the 33rd International Conference on Software Engineering*. 571–580.

[4] EJS. 2017. (2017). Retrieved October 17, 2017 from https://github.com/mde/ejs

[5] EJS-Lint. 2017. (2017). Retrieved October 17, 2017 from https://github.com/RyanZim/EJS-Lint

[6] ESLint. 2017. (2017). Retrieved October 17, 2017 from https://github.com/eslint/eslint

[7] ESLint-plugin-pug. 2017. (2017). Retrieved October 15, 2017 from https://github.com/myfreeweb/eslint-plugin-pug

[8] XSS Filter. 2017. (2017). Retrieved October 15, 2017 from https://msdn.microsoft.com/en-us/library/dd565647(v=vs.85).aspx

[9] Fis-parser-ejs. 2013. (2013). Retrieved October 17, 2017 from https://github.com/fouber/fis-parser-ejs

[10] A. Gizas, S. Christodoulou, and T. Papatheodorou. 2012. Comparative evaluation of JavaScript frameworks. In *Proc. of the 21st International Conference on World Wide Web*. 513–514.

[11] D. Graziotin and P. Abrahamsson. 2013. Making sense out of a jungle of JavaScript frameworks: towards a practitioner-friendly comparative analysis. In *Proc. of the 14th International Conference on Product-Focused Software Process Improvement (PROFES)*. 334–337.

[12] C. Heinrich. 2017. Comparison of 2003, 2004, 2007, 2010 and 2013 Releases. (2017). Retrieved October 15, 2017 from https://raw.githubusercontent.com/cmlh/OWASP-Top-Ten-2010/Release_Candidate/OWASP_Top_Ten_-_Comparison_of_2003,_2004,_2007,_2010_and_2013_Releases-RC1.pdf

[13] C. Jensen, A. Moeller, and Z. Su. 2013. Server interface descriptions for automated testing of JavaScript web applications. In *Proc. of the 9th Joint Meeting on Foundations of Software Engineering*. 510–520.

[14] V. Kashyap et al. 2014. JSAI: a static analysis platform for JavaScript. In *Proc. of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 121–132.

[15] ksdmitrieva. 2017. Analysis Pipeline. (2017). Retrieved December 10, 2017 from https://github.com/ksdmitrieva/AnalysisPipeline

[16] G. Li, E. Andreasen, and I. Ghosh. 2014. SymJS: automatic symbolic testing of JavaScript web applications. In *Proc. of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 449–459.

[17] Markdown. 2017. (2017). Retrieved October 15, 2017 from https://github.com/evilstreak/markdown-js

[18] N. Nikiforakis et al. 2012. You are what you include: large-scale evaluation of remote JavaScript inclusions. In *Proc. of the ACM Conference on Computer and Communications Security*. 736–747.

[19] NoScript. 2017. (2017). Retrieved October 15, 2017 from https://noscript.net/

[20] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. 2013. An empirical study of client-side JavaScript bugs. In *IEEE International Symposium on Empirical Software Engineering and Measurement*.

[21] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. 2015. Detecting inconsistencies in JavaScript MVC applications. In *ACM 37th IEEE International Conference on Software Engineering (ICSE)*.

[22] OWASP. 2017. Top 10 - 2017 Release Candidate. (2017). Retrieved May 15, 2017 from https://github.com/OWASP/Top10/raw/master/2017/OWASP%20Top%2010%20-%202017%20RC1-English.pdf

[23] OWASP. 2017. Top Ten Project. (2017). Retrieved September 29, 2017 from https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[24] I. Parameshvaran, E. Budianto, and S. Shinde. 2015. Auto-patching DOM-based XSS at scale. In *Proc. of the 10th Joint Meeting on Foundations of Software Engineering*. 272–283.

[25] Pug-lexer. 2016. (2016). Retrieved October 15, 2017 from https://github.com/pugjs/pug-lexer

[26] Pug-parser. 2017. (2017). Retrieved October 15, 2017 from https://github.com/pugjs/pug-parser

[27] Donald Ray and Jay Ligatti. 2012. Defining Code-injection Attacks. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 179–190. https://doi.org/10.1145/2103656.2103678

[28] Caja HTML Sanitizer. 2017. (2017). Retrieved October 15, 2017 from https://github.com/theSmaw/Caja-HTML-Sanitizer

[29] P. Saxena et al. 2010. A symbolic execution framework for JavaScript. In *Proc. of the IEEE Symposium on Security and Privacy*. 513–528.

[30] A. Taly, U. Erlingsson, J. Mitchell, M. Miller, and J. Nagra. 2011. Automated analysis of security-critical JavaScript APIs. In *Proc. of the IEEE Symposium on Security and Privacy*. 363–378.

[31] Weinberger et al. 2011. An empirical analysis of XSS sanitization in web application frameworks. In *EECS Department, University of California, Berkeley, Technical Report No. UCB/EECS-2011-11*.

[32] Weinberger et al. 2011. A systematic analysis of XSS sanitization in web application frameworks. In *Proc. of the 16th European Conference on Research in Computer Security*. 150–171.

[33] C. Yue and H. Wang. 2009. Characterizing insecure JavaScript practices on the web. In *Proc. of the 18th International Conference on World Wide Web*. 961–970.