

Hybrid Analysis for JavaScript Security Assessment

Omer Tripp, Omri Weisman
IBM Rational Software
Herzliya, Israel
+972 (9) 9629800
{omert, weisman}@il.ibm.com

ABSTRACT

With the proliferation of Web 2.0 technologies, functionality in web applications is increasingly moving from server-side to client-side code, primarily JavaScript. The dynamic and event-driven nature of JavaScript code, which is often machine generated or obfuscated, combined with reliance on complex frameworks and asynchronous communication, makes it difficult to perform effective security auditing of client-side JavaScript using existing static- and dynamic-analysis techniques.

We present a commercial-grade hybrid-analysis solution for automated security assessment of client-side JavaScript code. Our approach brings together the advantages of the white-box and black-box methodologies while overcoming their weaknesses. A black-box component interacts with the subject web application and collects pages that contain client-side JavaScript code. The pages are then analyzed using static taint analysis to detect security vulnerabilities. The black-box component provides URLs and other pieces of dynamic information that contribute toward specializing the static analysis, making it much more precise and effective than its baseline version, as we demonstrate empirically.

General Terms

Algorithms, Security, Verification.

Keywords

Web application security, cross-site scripting, JavaScript, black-box, static analysis, hybrid analysis.

1. BACKGROUND & MOTIVATION

There are several known JavaScript security vulnerabilities, the most significant one being a variant of *cross-site scripting (XSS)* [1] called *DOM-based XSS* [2].

There are three primary types of XSS. In *reflected* and *stored* XSS, malicious attackers take advantage of insufficient server-side input validation to inject a client-side script into web pages viewed by other users. DOM-based XSS is different in that the injection manifests in client-side code, without ever reaching the server. This vulnerability generated significant interest when discovered in 2005, partly because its client-side nature allows it to bypass the defense measures of web-application firewalls. Since then, DOM-based XSS vulnerabilities have been discovered

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Conference'10, Month 1–2, 2010, City, State, Country.
Copyright 2010 ACM 1-58113-000-0/00/0010...\$10.00.

in many real-world applications, including Adobe Flex [3], Yahoo Mail [4], and Dojo [5]. Here is a simple example:

```
<html>
  <script type="text/javascript">
    var pos=document.URL.indexOf("name=")+5;
    document.write(
      document.URL.substring(pos,document.URL.length));
  </script>
</html>
```

An attacker can send a link such as [http://hostname/welcome.html#name=<script>alert\(1\)</script>](http://hostname/welcome.html#name=<script>alert(1)</script>) to the victim resulting in the victim's browser executing the injected client-side code.

Detection of DOM-based XSS is challenging, even for seasoned security auditors, since it usually requires in-depth review of JavaScript source code, which is complicated in several respects. First, JavaScript code is often obfuscated or compressed by developers concerned about protecting intellectual property or conserving bandwidth. Second, web development frameworks, such as Microsoft .NET and Java Server Faces, introduce sizeable amounts of complex machine-generated client-side JavaScript code. Use of JavaScript frameworks, such as Dojo or jQuery, further aggravates the problem, since code that depends on a framework can only be fully understood given the source code of the framework itself, which is typically several orders of magnitude larger than the client code.

All this makes the case for automation, but the classic approaches for automated security scanning – based either on black-box testing or on static analysis, but not on a combination of these two methodologies – are also limited, as we now discuss.

Black-box testing. Traditional black-box methods for the detection of reflected or stored XSS work by sending HTTP requests containing test payloads to the subject web server and looking for the reflected payload in the HTTP response. This approach does not work for DOM-based XSS, since the test payload is not included in the response, but rather gets embedded in the DOM as the JavaScript code executes. This is discussed in detail by Klein [2].

In response to this challenge, JavaScript execution capabilities have been added to black-box scanners. There are two primary ways in which JavaScript execution can be used to detect client-side vulnerabilities.

The first way is to inject test payloads into the URL and parameter values, and then observe the side effects of sandboxed JavaScript execution. For example, the test might attempt to trigger a pop-up alert box, in which case its validation would be to check whether such a pop-up box indeed appeared during JavaScript execution.

The second alternative is to perform dynamic taint analysis via instrumentation of the client-side code [6]. When used in

combination with symbolic analysis, this approach is favorable in that it improves both the coverage of the client-side state space, and the detection of successful attacks.

Still, the enormity of the client-side state space – due to event-driven behaviors, complex dependencies between the HTML DOM and JavaScript code, and asynchronous communication with the server side – leaves coverage as a fundamental challenge for both variants of JavaScript execution.

Static analysis. The problem of partial coverage is not resolved by white-box scanners, which typically analyze JavaScript content in the web application's source code; in some cases, coverage is even poorer. It is common for JavaScript content to be loaded dynamically from independent resources that may reside outside the application's codebase. JavaScript code may also be generated dynamically as a side effect of server-side or client-side code execution. Here is a real-world example ([7]):

```
<script type="text/javascript">
document.write('<scr'+ipt ');
document.write('src="http://affinity-numerology.com/cgi-
bin/EmailThisLink.cgi?g'+Email_This_Link+'");
document.write(' type="text/javascript">');
document.write('</scr'+ipt>');
</script>
```

In this example, not only is JavaScript code generated dynamically, but the target URL that determines the inclusion of a separate JavaScript source file is itself dynamic.

Another limitation of static analysis is imprecision due to client-side frameworks, such as jQuery and Dojo, whose runtime behavior is hard to model statically due to their inherent complexity, proprietary syntax, and tight interaction with the DOM.

2. THE HYBRID APPROACH

We have developed a hybrid solution for security analysis of client-side JavaScript where a black-box crawler interfaces with a static security analysis. In this section, we discuss the makeup of our solution with special emphasis on aspects of our system that are innately hybrid.

2.1 System architecture

In our architecture, a web crawler is used to explore the pages of a target web application and collect JavaScript content to be analyzed. The web crawler applies JavaScript execution to the retrieved web pages to maximize coverage and simplify the ensuing analysis. The web crawler also records other pieces of dynamic information, such as the concrete URL values for each page and sub-element therein.

Once a page has been retrieved, and JavaScript execution has completed, we perform security assessment using static analysis. Since common client-side vulnerabilities – and in particular, DOM-based XSS – are injection vulnerabilities, our security assessment is encoded as a taint-analysis problem [8].

The taint analysis is parameterized by a set of security rules. Each rule contains a specification of "sources", "sinks", and "sanitizers". *Sources* are points in the program that read untrusted information; *sinks* are security-sensitive operations; and *sanitizers* are operations that endorse untrusted data, thereby making it trusted.

Flow of untrusted information from a source to a sink that is not mediated by an appropriate sanitization operation is considered a security violation. Our analysis detects injection vulnerabilities by following this reasoning, which reduces security analysis to a data-flow reachability problem.

An exciting feature of the hybrid approach is the ability to specialize the static analysis based on information provided by the black-box scanner. Our system currently supports two forms of specialization, which we describe in turn.

2.2 Hybrid elimination of false findings

Sound static analysis makes conservative assumptions about the environment in which a program is run. Thus, in the absence of any further information, the following real-world code might be considered vulnerable to Open Redirect [9] attacks, where the attacker can influence the target URL of a redirection operation:

```
var str = document.URL;
var url_check = str.indexOf('login.html');
if (url_check > -1) {
  result = str.substring(0, url_check);
  result = result + 'login.jsp' +
    str.substring((url_check+search_term.length),
      str.length);
  document.URL = result;
}
```

The vulnerable flow extends from the statement reading field `URL` of the document object to the assignment of `result` to that field, since data flow between these two endpoints suggests that the attacker can potentially influence the target URL to which the web page redirects.

In practice, this code does not contain an Open Redirect vulnerability. While part of the target URL can indeed be controlled by an attacker (the request parameters), the target hostname of the redirection is beyond the attacker's control.

We have designed and implemented an effective string analysis [10] for eliminating false findings of this sort. Seeded by concrete URLs provided by the black-box crawler, our string analysis conservatively models URL strings as a concrete prefix and an unknown suffix. This is a natural fit for taint analysis: The part controlled by the attacker is unknown, but the uncontrolled prefix is modeled precisely.

For example, a safe approximation of the (anonymized) URL where the above sample code was found is:

https://mySite/release/jsp/sso/login.html?.*

where `.*` is a conservative regular approximation of the part of the URL controlled by the attacker that contains the parameters set on the HTTP GET request.

At the sink statement, the analysis checks whether the host and path parts of the URL can be controlled by an attacker. If it is determined that both these parts are fixed, and not controllable by an attacker, then the issue is eliminated.

Our string-analysis algorithm is implemented as an IFDS [11] problem, where data-flow facts represent string abstractions. On top of its regular representation of the string content (which abstracts away only the string suffix, as exemplified above), the abstraction also records

- environment and heap pointers into the string (in a manner akin to TVLA's shape abstractions [1]) for sound handling of aliasing between strings; and
- relationships between strings and integral variables defined via `indexOf` calls for precise handling of index-based string operations, such as `substring` and `charAt`.

Here is an illustrative example:

```

1: var url = document.location.url;
2: var loginIdx = url.indexOf('login');
3: var loginSuffix = url.substring(loginIdx);
4: url = 'https://mySite/html/sso/' + loginSuffix;
5: document.location.url = url;

```



The statement at label 5, which is theoretically vulnerable to Open Redirect attacks, is proved safe by our string analysis, which establishes that the controlled portion of any concrete string flowing into this statement is strictly beyond an uncontrolled prefix containing the `?` character.

We emphasize that the above analysis critically relies on the availability of the concrete URL provided by the black-box component. This information is accessible in a hybrid setting, but cannot, in general, be recovered by a purely static analysis.

2.3 Hybrid DOM modeling

The HTML DOM is an important channel of data propagation. JavaScript code can manipulate the values of DOM elements, which enables communication between different JavaScript programs in the same HTML page (as well as between invocations of the same program). In particular, `onload` events frequently trigger major DOM transformations, which are hard to approximate in a purely static setting.

In our hybrid solution, the static analysis operates on a fully resolved DOM, which is the result of import resolution as well the execution of load events. This enables accurate tracking of vulnerable flows through the DOM.

Our analysis further leverages the fact that all attribute values are set and the full DOM structure is available for "DOM reduction": The original DOM is replaced by a reduced DOM where redundant data is omitted. For example, if there is a data table with 50 symmetric records in the original DOM, then the reduced DOM defines the same table but only with 1 record. The performance benefit of this transformation is significant, since real-world HTML pages with prohibitively large DOMs (translating to $>10^5$ text lines) that exhibit a great deal of redundancy are not infrequent.

Technically, DOM reduction is accomplished by inducing an equivalence relation on DOM elements, where two elements are equivalent iff all their children are pair-wise equivalent, and

- the (unique) paths between the HTML root and both elements are identical;
- both elements define the same set of attributes; and
- both elements agree on the values of all JavaScript-related attributes (e.g., `onClick`, `onmouseover`, etc).

3. IMPLEMENTATION & EVALUATION

Our algorithm is featured in a commercial black-box security-scanning product [12]. To evaluate the efficacy of the hybrid solution, we conducted two sets of experiments. The purpose of the first experiment (Section 3.1) was to assess the effect of the hybrid specialization techniques described in Section 2 on the performance and precision of the static security analysis. The second experiment (Section 3.2) was designed to measure the merit of augmenting the baseline black-box scanner with hybrid analysis capabilities.

3.1 1st Experiment: impact of specialization

For the first experiment, we used a sample set of 675 real-world websites. These consisted of all the Fortune 500 companies' websites [13], the top 100 websites list [14], and other handpicked websites including IT, security vendors, and social-networking sites.

We harvested content from each site using the commercial scanner's built-in non-intrusive web crawler. The crawler starts at the site's homepage and performs bounded DFS indexing. In this way, we collected between 200 and 500 pages and JavaScript files per site. To avoid the risk of nonstandard interaction with the site, our web crawler did not log into the application, nor did it submit any HTML forms.

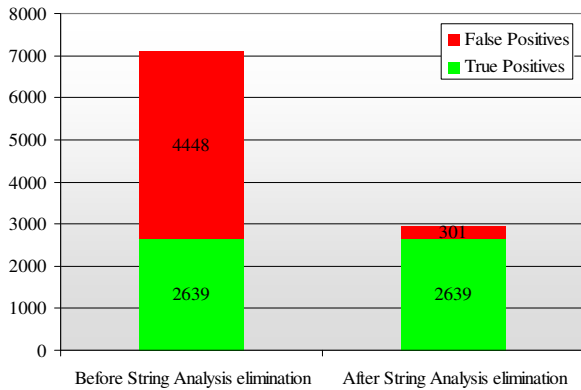
The collected pages were then analyzed by our system running in the four configurations induced by either enabling or disabling string-prefix analysis and DOM modeling. The results were manually reviewed and classified by a security expert to determine whether they were exploitable security issues or false findings.

Results. DOM modeling proved to be essential. With this feature disabled, the analysis crashed on a substantial portion of the pages, which prevented us from completing the experiments where this option is turned off.

As for string-analysis elimination, with this feature enabled, the hybrid engine reported 2940 JavaScript security vulnerabilities in 188 of the 675 websites. Manual review concluded that approximately 10% of the reports were false. The impact of string-analysis elimination (shown in Figure 1) proved to be

highly significant: Over 4,000 false reports were correctly suppressed by our algorithm, and – as we expected thanks to its soundness – the algorithm did not suffer from false negatives.

Figure 1. Total number of JavaScript security vulnerabilities detected for 675 websites



3.2 2nd experiment: merit of hybrid features

Of the 675 websites used in the previous experiment, we selected 60 sites at random. We ran the black-box scanner twice on all these sites, with and without the hybrid capabilities.

Table 1. Client-side vulnerabilities found by black-box scanner with and without hybrid capabilities

Number of websites tested	60
Websites found to be vulnerable by baseline scanner (without hybrid capabilities)	8 (0 false positives)
Websites found to be vulnerable by scanner with hybrid capabilities	33 (4 false positives)

Results. The results, which are summarized in Table 1, indicate that the hybrid solution significantly improves the scanner's ability to detect client-side vulnerabilities. Manual review by a security expert determined that four of the websites were falsely reported by the hybrid analysis as vulnerable. This is consistent with the 10% false-positive rate computed for the benchmarks in Section 3.1. Analysis of the false reports indicates that they were all due either to infeasible or to extremely rare path conditions, which – from a security viewpoint – renders the reported flows unexploitable.

4. CONCLUSION

We presented a hybrid solution that combines static- and dynamic-analysis capabilities toward automated detection of client-side security vulnerabilities. We demonstrated that our solution provides highly accurate results on a diverse set of real-world applications, and considerably improves a state-of-the-art standalone black-box scanner. We also confirmed that specialization of the static analysis via dynamic hints, which is only viable in a hybrid setting, has a dramatic impact on the quality of the analysis by allowing the elimination of the vast majority of false reports and boosting the analysis' scalability.

5. ACKNOWLEDGMENTS

The authors would like to thank Yinnon Haviv, Daniel Kalman, Dmitri Pikus, Lotem Guy, Julian Dolby, Marco Pistoia, Takaaki

Tateishi, Ory Segal, Adi Sharabani, and Yair Amit for their contribution to the work presented in the paper. Additionally, we extend our thanks to Yishai Feldman, Ory Segal, and Jonathan Cohen for their help in reviewing the paper.

6. REFERENCES

- [1] CERT. Advisory CA-2000-02, 2000: Malicious HTML Tags Embedded in Client Web Requests <http://www.cert.org/advisories/CA-2000-02.html>
- [2] A. Klein, 2005: DOM Based Cross Site Scripting or XSS of the Third Kind <http://www.webappsec.org/projects/articles/071105.shtml>
- [3] O. Segal, A. Sharabani, and A. Yogev, 2008: JavaScript Code Flow Manipulation, and a real world example advisory - Adobe Flex 3 Dom-Based XSS <http://blog.watchfire.com/wfblog/2008/06/javascript-code.html>
- [4] P. Agrawal, 2010: Yahoo! Mail Dom Based XSS Vulnerability <http://seclists.org/fulldisclosure/2010/Jun/289>
- [5] Gotham Digital Science (GDS) 2010: Multiple DOM-Based XSS in Dojo Toolkit SDK <http://www.securityfocus.com/archive/1/510093>
- [6] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis (ISSTA '07)*. ACM, New York, NY, USA, 196-206.
- [7] The Affinity Numerology website <http://affinity-numerology.com/Lucky-Numbers/index.php>
- [8] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI '09)*. ACM, New York, NY, USA, 87-97.
- [9] Open Web Application Security Project (OWASP): Open redirect http://www.owasp.org/index.php/Open_redirect
- [10] Yasuhiko Minamide. 2005. Static approximation of dynamically generated Web pages. In *Proceedings of the 14th international conference on World Wide Web (WWW '05)*. ACM, New York, NY, USA, 432-441.
- [11] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '95)*. ACM, New York, NY, USA, 49-61.
- [12] IBM Rational AppScan Standard Edition <http://www-01.ibm.com/software/awdtools/appscan/standard/>
- [13] Fortune 500 2010: Annual Ranking of America's Largest Corporations from Fortune <http://money.cnn.com/magazines/fortune/fortune500/2010/>
- [14] Web 100 <http://www.web100.com/web-100>
- [15] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. 1999. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '99)*. ACM, New York, NY, USA, 105-1