DEXTERJS: Robust Testing Platform for DOM-Based XSS Vulnerabilities

Inian Parameshwaran Enrico Budianto Shweta Shinde
Hung Dang Atul Sadhu Prateek Saxena
National University of Singapore, Singapore
{inian, enricob, shweta24, hungdang, atulsadh, prateeks}@comp.nus.edu.sg

ABSTRACT

DOM-based cross-site scripting (XSS) is a client-side vulnerability that pervades JavaScript applications on the web, and has few known practical defenses. In this paper, we introduce DEXTERJS, a testing platform for detecting and validating DOM-based XSS vulnerabilities on web applications. DEXTERJS leverages source-to-source rewriting to carry out character-precise taint tracking when executing in the browser context — thus being able to identify vulnerable information flows in a web page. By scanning a web page, DEXTERJS produces working exploits that validate DOM-based XSS vulnerability on the page. DEXTERJS is robust, has been tested on Alexa's top 1000 sites, and has found a total of 820 distinct zero-day DOM-XSS confirmed exploits automatically.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming; D.2.5 [**Software Engineering**]: Testing and Debugging; D.4.6 [**Operating Systems**]: Security and Protection

Keywords

Web Security, Taint Analysis, DOM-based XSS

1. INTRODUCTION

JavaScript has become a scripting language which goes beyond the web platform. It now powers many popular web applications, HTML5-based mobile applications, and server-side scripts (e.g., NodeJS). However, presently, applications built with JavaScript are fraught with DOM-based XSS, a code injection vulnerability which is known to be highly pervasive and an elusive category of vulnerabilities for many commercial scanners to find [15]. A majority of popular web sites including Google, Twitter and Yahoo have recently been vulnerable to these attacks [1,2,4]. Nearly all available solutions for detecting DOM-based XSS vulnerability today, including the research proposals (e.g., FLAX [17], Lekies *et al*. [15]) and commercial tools (e.g., DominatorPro [5]), are designed as modifications to a JavaScript interpreter (implemented in a web browser). This approach has practical limitations as the analysis infrastructure is restricted to specific browsers or platforms which fail to capture the massive variation in program behavior that exists on the real end-user device/system.

In this work, we develop an end-to-end testing platform called DEXTERJS. DEXTERJS robustly analyses and finds DOM-based XSS vulnerabilities in client-side web applications, with the ability to produce working exploits for vulnerable applications. DEXTERJS instruments JavaScript of a website that client visits to carry out character-precise taint tracking — thus guaranteeing the analysis engine is browser-agnostic. Once the instrumented application runs on the client's browser, it dynamically analyses the portions of tainted strings that are used in the sensitive code evaluations functions. The instrumented application then records such potentially vulnerable flows. Based on the information provided by the *logs* (Section 3), DEXTERJS then constructs candidate exploits and tests them on the actual website to verify its exploitability. Once they have been confirmed, the exploits are then reported to the user via a web interface.

We carry out a large-scale scan on Alexa's top 1000 websites to evaluate the robustness of DEXTERJS. Our analysis of JavaScript from 228,541 URLs suggests that there are 777,082 distinct dynamic code evaluation instances constructed from untrusted data — which can potentially be exploited. Among these potentially vulnerable flows, we find a total of 820 distinct zero-day DOM-based XSS vulnerabilities across 89 different domains along with sample exploits. Our practical detection of DOM-based XSS attacks requires no modification of browsers or server-side code, and does not require users to install any plug-ins or extensions to web browsers, as compared to existing detection tools [5, 15].

2. TECHNICAL CHALLENGES

In designing DEXTERJS, one of the key capabilities we developed is a robust source-to-source transformation engine for JS applications. DEXTERJS enables intrusive instrumentation to the JavaScript application, specifically character-precise dynamic taint analysis, for identifying vulnerable sinks. This approach allows the instrumented code to execute on any number of browsers (or HTML5-based) backends. Further, the analysis implementation is easy to maintain for a variety of browser versions and can gracefully be extended to handle future language changes without requiring re-implementation.

Designing a robust, semantics-preserving JavaScript transformation system has been technically challenging in our experience. Though a number of frameworks provide APIs for general-purpose JavaScript rewriting, DEXTERJS handles a variety of practical challenges arising from many dimensions: the dynamic nature of JavaScript, the pervasive use of reflection, unconventional or browser-specific scoping rules, browser limitations on code size blowup,

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2803191

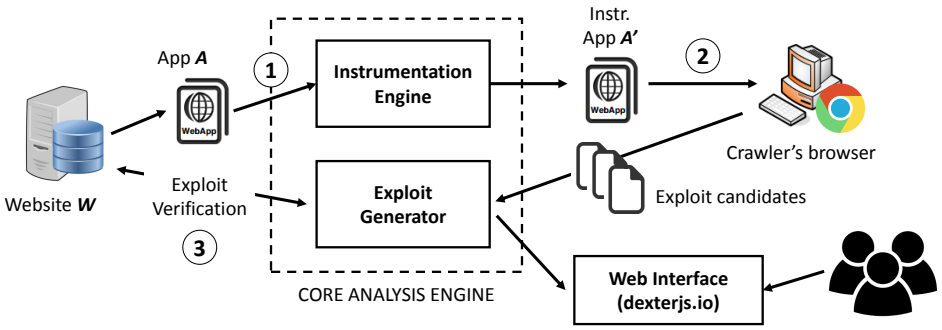

Figure 1: Overview of DEXTERJS's design. After receiving a URL of website W, DEXTERJS instruments the client-side code of the web page (Step 1). It then instruments the client-side code and executes it on various browsers (Step 2). DEXTERJS then analyzes vulnerable flows in the application and reports those to the exploit generator module as *exploit candidates*. The module then verifies whether such flows are exploitable by constructing sample exploits and testing it against the scanned web page of W (Step 3).

and completeness in handling browser built-in objects. Handling these features robustly by source rewriting, rather than as modifications to specific browser versions allows DEXTERJS to perform instrumentation that can run on multiple browser backends.

3. CORE ANALYSIS ENGINE

Figure 1 illustrates the design overview of DEXTERJS's core analysis engine. In this section we detail two core modules of DEXTERJS namely the instrumentation engine and the exploit generator. A detailed description of the techniques underlying DEXTERJS can be found in [16].

3.1 Instrumentation Engine

DOM-based XSS can be seen as an information flow problem where portions of strings — potentially under attacker's control (e.g., URL, cookie value) — are being evaluated as code through JavaScript code evaluation functions like `eval()` or being used in unsafe dynamic DOM constructions, such as via `document.write` or `innerHTML`. Therefore, taint tracking can be reliably used to detect such vulnerabilities. DEXTERJS acts as a trusted man-in-the-middle proxy that intercepts any HTTP(S) requests from the browser (Figure 1 Step 1), identifies scripts in the responses, and rewrites them to perform character-precise taint tracking. The taint propagation logic and metadata is kept within the website's execution context in the browser. Once the user submits the URL to be scanned, DEXTERJS automatically crawls and instruments the application using a selenium-based crawler (Figure 1 Step 2). We use dynamic taint analysis to detect all flows from unsafe input sources (i.e., can be controlled by attacker) to code execution sinks. The list of all the sources and sinks used by DEXTERJS is the same as in [16].

Storage and Propagation of Taint Information. DEXTERJS stores taint information along with each string and uses boxing to accomplish this as shown in Listing 1. This approach is efficient in looking up taint information and does not lead to a memory blow up. In our experience other approaches such as storing the taint information in a separate namespace or in the global namespace is not a scalable approach [16].

Listing 1: Conversion to Objects to track taint information.

```
1 var a = new String("foo"); // a is string object
2 a.taint = true;
3 console.log(a.taint); // prints true
```

The taint information is propagated using immediately invoked function expressions (or IIFE [9]) as shown in Listing 2.

Listing 2: IIFE used to group taint analysis statements

```
1 var a = (function() {
2   var rhs = b();
3   return rhs + "foo";
4 })();
```

3.2 Exploit Generation

Our exploit generator module utilizes input from the instrumentation engine that reports potentially exploitable taint flows. This module analyses the tainted flows and generates context-based exploits, which can be easily verified to the original vulnerable web pages. Currently, the module supports exploit generation for URL-based DOM-XSS, similar to what has been proposed in [15]. We explain how this module works based on a real-world example illustrated in Figure 2.

3.2.1 *Flow Parser*

This module accepts the logs supplied by the instrumentation engine as its input. The logs contain all the tainted flows discovered during the execution of instrumented web application by the crawler. The flow parser extracts following information from the logs: taintID for storing a unique ID for each tainted flow, benignURL which is the URL of the page where the taint flow was discovered, taintSource which contains the type of source and the characters in the source string which are used in the sink, taintSink which is the type of sink and the exact characters which are controlled by the attacker. In the example shown in Figure 2, taintSource is the function `getAttribute()` and taintSink is `document.write()`.

3.2.2 *Context Identifier*

Tainted strings can exist in various parts of a web page, such as in HTML contents and attributes (e.g., `href, onload`), CSS properties, or JavaScript (e.g., `<script>` tag). We term parts of a web page where the tainted string was injected as *context*. The context parser is responsible for finding the context of the injection using the taintSource and taintSink information. It generates a HTML parse tree based on the given input. Using this parse tree and the character-precise taint information, it figures out the exact context in which the injection takes place (Figure 2 Step 3).

3.2.3 *Exploit Generator*

In order for an exploit to work, the tainted string is then replaced with an attack vector. The attack vector to be injected must be adjusted to the context where the attack vector is going to be injected in a web page (see Section 3.2.2). This module first tries

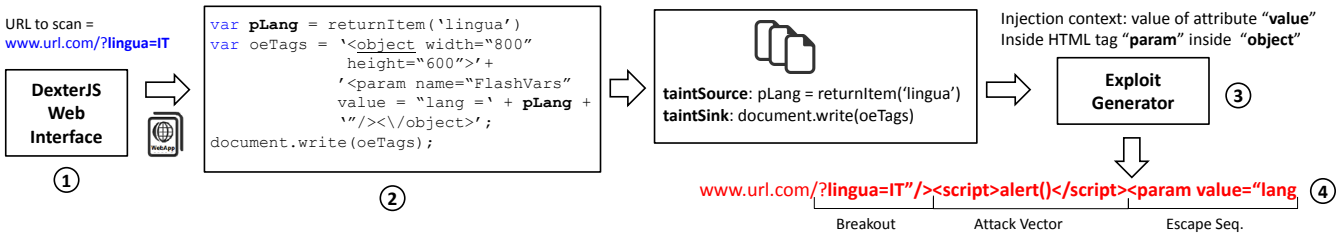

Figure 2: Overview of DEXTERJS's exploit generation module. First, DEXTERJS receives URL of a web page to be scanned (Step 1) and get the source code of the web page (Step 2). DEXTERJS then instruments the code and run it on various browsers. The browsers then report any vulnerable flows found to the exploit generator module by attaching the following information: taintSource, taintSink, and the tainted value itself. The exploit generator module then decides the context where the sample exploits must be injected (Step 3). It then generates the final exploit URL, which will be verified to the original website (Step 4).

to break out into the JavaScript context from the context identified by the Context Identifier. This is necessary to make sure that the attack vector is always injected in a script context. Using the parse tree from *Context Identifier*, DEXTERJS is able to generate the required sequence to break into the JavaScript context. After generating the breakout sequence, DEXTERJS generates browser specific attack vectors based on the type of sink, the context information, taintSink and taintSource. Our attack vectors are based on publicly known strategies such as the XSS filter evasion cheat sheet [13, 15]. To make sure that the rest of the page renders properly and our attack vector is executed, we generate escape sequence and concatenate it to the existing exploit candidate. The final exploit URL is constructed by combining the output of the sub-modules as follows.

Exploit = BreakOutSequence + AttackVector + EscapeSequence

A sample of resulted exploit URL is shown in Figure 2 Step 4. Since our focus is on DOM-based XSS carried out via URLs, the end result of this step is a set of candidate exploit URLs containing the attack vector that needs to be validated.

Exploit Verifier. This module is responsible for validating whether candidate exploit URLs successfully inject the attack code into the scanned web page (Figure 1 Step 3). If the attack is successful, the injected code will send out the taintID of the tainted flow to the exploit verifier webpage. The exploit verifier webpage stores these taintIDs in a file. Once a web page is confirmed as exploitable, the module sends the vulnerable URL, the exploits found in that URL, and the different browsers in which the page is exploitable.

3.3 Noticeable Features

DEXTERJS is designed to instrument and scan both HTTP and HTTPS websites. DEXTERJS use `mitmproxy` to generate valid certificates for the websites visited on the fly and use the analysis engine to rewrite the scripts in these pages. DEXTERJS can scan pages which require login by accepting valid cookies from the user. These cookies are then passed on to the server when making the request to get the authenticated version of the page to be scanned.

4. IMPLEMENTATION

We implement DEXTERJS as a proxy server based on mitmproxy [11]. We build DEXTERJS's instrumentation engine and exploit generator modules using the Node.js platform. We implement our source-to-source rewriting logic with 1,896 lines of JavaScript code using Esprima [6] and Cheerio [7] node modules.

We utilize the Selenium framework [12] to implement a web crawler and a simple GUI fuzzing tool. Our Python based crawler is written in 808 lines of code. We have tested the crawler on mainstream browsers such as Chrome, Firefox, Opera and Safari. It supports Internet Explorer in only a limited manner due to the instability of the existing Selenium driver. Our crawler is capable of configuring the browser proxy, enforcing a page-load time-out, browsing in private-mode and controlling specific browser features like the Chrome's XSS-auditor. It performs a wide range of interactions on websites, such as extracting data from DOM nodes, filling forms or input boxes, triggering different events — which are necessary to dynamically analyze features in the instrumented web application [14].

5. EVALUATION

We demonstrate the robustness of DEXTERJS by instrumenting real-world websites and publicly-available benchmarks for DOM-based XSS. All the experiments were conducted on an Intel Xeon® 2.0 Ghz CPU with 64 GB RAM.

Accuracy. We test the accuracy of DEXTERJS instrumentation engine against two DOM-based XSS benchmarks namely IBM JavaScript test suite [10] and Google's Firing Range [8]. We report that DEXTERJS is able to detect all DOM-based XSS vulnerabilities and has zero false negative rates with respect to these benchmarks.

Scalability. We crawl 228,541 page URLs starting from the Alexa Top 1000. In total, DEXTERJS instruments 13,255,378 HTML and 15,769,329 JS files. Our evaluation shows that the DOM tree of the web page remains unchanged after instrumentation after comparing the DOM tree of the instrumented and non-instrumented applications using Selenium driver. This indicates that the instrumentation does not alter the original functionality of the client-side applications. Of these sites, DEXTERJS automatically generates working exploits for 820 vulnerable points of web pages on 89 different domains, meaning that there are 820 exploitable web pages. These domains include many high-profile websites such as *comodo.com* and *washingtonpost.com*. The majority of the vulnerable flows come from a source-sink pair `location.href` and `document.write`, which accounts for 55% of the observed vulnerabilities.

Robustness. The browser-agnostic nature of DEXTERJS allows us to measure browser-specific taint flows, i.e., taint flows that exhibit only on some browsers. We collect a random sample of 100 taint flows and found 3 instances of flows that exhibit only on Firefox and 2 only on Chrome, out of those 100 flows. Browser-specific taint flows justifies the need for browser-agnostic testing platform like DEXTERJS.

6. TOOL AVAILABILITY

We deploy DEXTERJS as a web service located at `https://dexterjs.io`. This service is accessible to user after completing registration process. During the registration process, user needs to supply her e-mail address which will later be used. Once logged in,

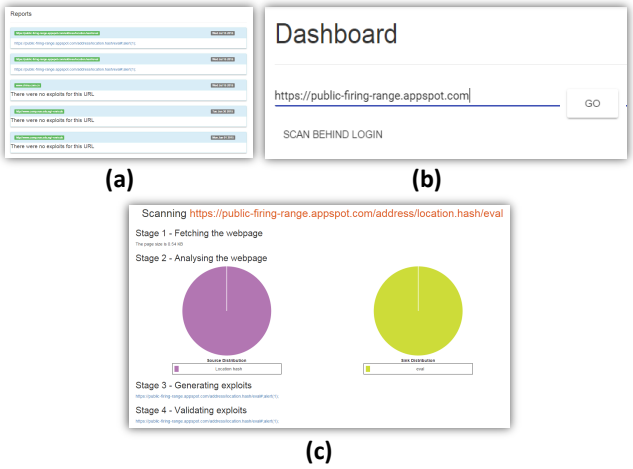

Figure 3: Screenshot DEXTERJS Web Interface. (a) Record of previous scans. (b) Interface to scan a web page. (c) Analysis result, including source-sink distribution, generated exploits, and validated exploits.

users can simply enter the URL of a web page that they would like to scan, as shown in Figure 3. Once completed, a report of the vulnerable DOM-based XSS flows and a URL containing exploits of the scanned web page will be shown to the user. We release a video demonstrating step-by-step guideline of using DEXTERJS [3]. The website has been running since February 2015 and garnered 2,872 pageviews (according to Google Analytics) and 88 registered users.

7. RELATED WORK

Dynamic taint tracking has been employed to perform information flow analysis of JavaScript-based applications. The closest study on JS dynamic analysis to our work are FLAX [17] and Jalangi [18]. FLAX uses character level taint tracking for dynamic analysis in JavaScript applications. Jalangi employs a technique to record and replay a user-selected part of the program in their dynamic analysis tool called Jalangi. Unlike our technique, FLAX and Jalangi perform dynamic analysis after the actual program has finished execution. Our technique encompasses *in-situ* dynamic analysis to give immediate results. In order to achieve this, we implement character-level taint tracking as opposed to FLAX and complete code instrumentation as opposed to partial instrumentation carried out by Jalangi.

Even though DOM-based XSS was first mentioned nearly a decade ago [14] and is very much prevalent, there are not many studies proposing solution to detect such vulnerabilities. Lekies et al. [15] proposed an approach which automatically detects and validates DOM-based XSS vulnerabilities similar to DOMinatorPRO [5], a commercial tool. Both these tools modify specific browser versions to achieve their purpose. DEXTERJS, on the other hand, is browser-agnostic and can be directly adopted to perform web security testing without installing additional software or plug-ins.

8. CONCLUSION

We presented DEXTERJS, a web security testing platform for finding DOM-based XSS vulnerabilities in web applications, as well as generating working exploits for vulnerable web pages of the applications. DEXTERJS features character-precise taint tracking that is of wide interest in many security analyses. Our tool is robust and scales to the Alexa Top 1000 sites on multiple browser backends. Using DEXTERJS, we find hundreds of zero-day DOM-based XSS exploits. DEXTERJS is available through a web interface at `https://dexterjs.io/`

9. ACKNOWLEDGEMENT

We thank Abhinit Ambastha for his assistance. This research is supported in part by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate. This work is also supported in part by a university research grant from Intel.

10. REFERENCES

[1] A Twitter DomXSS, a wrong fix and something more. `http://goo.gl/dHF457`.

[2] Analyzing a Dom-Based XSS in Yahoo! `http://goo.gl/yXKtf4`.

[3] DexterJS Video Demo. `https://www.youtube.com/watch?v=73BsDij5Fu4/`.

[4] DOM XSS on Google Plus One Button. `http://goo.gl/ohRAkM`.

[5] DominatorPro: Securing Next Generation of Web Applications. `https://dominator.mindedsecurity.com/`.

[6] ECMAScript Parsing Infrastructure for Multipurpose Analysis. `http://esprima.org/`.

[7] Fast, Flexible, and Lean Implementation of Core jQuery Designed Specifically for the Server. `https://github.com/cheeriojs/cheerio`.

[8] Firing Range. `http://public-firing-range.appspot.com/`.

[9] Immediately-invoked Function Expression. `http://benalman.com/news/2010/11/immediately-invoked-function-expression/`.

[10] LaBaSec: Language-based Security. `http://researcher.watson.ibm.com/researcher/view_group_subpage.php?id=1598`.

[11] Mitmproxy: a man-in-the-middle proxy. `http://mitmproxy.org/`.

[12] SeleniumHQ Browser Automation. `http://seleniumhq.org/`.

[13] XSS Filter Evasion Cheat Sheet. `https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet`.

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

[15] S. Lekies, B. Stock, and M. Johns. 25 Million Flows Later - Large-scale Detection of DOM-based XSS. In *2013 ACM SIGSAC Conference on Computer and Communications Security, Berlin, Germany, November 4-8*. ACM, 2013.

[16] I. Parameshwaran, E. Budianto, S. Shinde, H. Dang, A. Sadhu, and P. Saxena. Auto-Patching DOM-Based XSS at Scale. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015.

[17] P. Saxena, S. Hanna, P. Poosankam, and D. Song. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 2010*, 2010.

[18] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 488–498. ACM, 2013.