DomXssMicro: A Micro Benchmark for Evaluating DOM-Based Cross-Site Scripting Detection

Conference Paper · August 2016

DOI: 10.1109/TrustCom.2016.0065

2 authors, including:

Xiaoguang Mao
National University of Defense Technology
96 PUBLICATIONS **642** CITATIONS

SEE PROFILE

DomXssMicro: A Micro Benchmark for Evaluating DOM-based Cross-Site Scripting Detection

Jinkun Pan
College of Computer
National University of Defense Technology
Changsha, China
pan_jin_kun@163.com

Xiaoguang Mao
College of Computer
Laboratory of Science and Technology on Integrated Logistics Support
National University of Defense Technology
Changsha, China
xgmao@nudt.edu.cn

***Abstract*—With the prevalence of JavaScript, Cross-site Scripting based on Document Object Model (DOM-based XSS) has become one of critical threats to client-side Web applications. To detect DOM-based XSS vulnerabilities, a variety of tools have been developed, providing different features and abilities. Both for developers and tool users, the benchmark plays an important role in evaluating the effectiveness of detection tools. However, no widely used standard benchmark exists in the domain of DOM-based XSS. In this paper, we present a micro benchmark named DomXssMicro. DomXssMicro is constructed based on a template extracted from representative vulnerabilities, consisting of six orthogonal components (i.e. Source, Propagation, Transformation, Sink, Trigger and Context). In DomXssMicro, there are 175 test cases in total, each one of which aims at testing a specific property of DOM-based XSS. To illustrate our micro benchmark, an empirical study is performed to evaluate six state-of-the-art DOM-based XSS detection tools, including both commercial and open-source ones. The results demonstrate that our micro benchmark is helpful in providing guidance and insight for tools selection and further improvement.**

***Keywords* - micro benchmark; DOM-based XSS; tool evaluation**

I. INTRODUCTION

Since the development of Web 2.0, rich client applications become more and more popular. More functionality moves from the Web server to the client. Complex and sophisticated applications written in JavaScript can be run in modern browsers, having various features such as good user experience, high efficiency and flexibility that are more competitive than equivalents implemented in desktops or traditional Web servers. As a consequence, the amount of JavaScript code has a rapid growth. Moreover, the abilities of JavaScript are being augmented as the prevalence of HTML5.

Along with the ever increasing amount and complexity of the Web client, a number of security issues that are specific to the client are emerging [1–3]. One of them is called DOM-based XSS [4], which is a subtype of Cross-site Scripting (XSS) problems that results from unsafe processing of untrusted data from sources controlled by the attacker. In a recent study, it has been shown that nearly one tenth of Alexa Top 5000 contain at least one DOM-based XSS vulnerability [5]. Compared to reflected and stored XSS, DOM-based XSS is more challenging to deal with. DOM-based XSS occurs on the browser of the user rather than the server, which is out of the control of the service provider. Furthermore, the dynamic code generation and evaluation features of JavaScript making new code can be generated and executed purely on the client which is not visible to the server. That is to say, approaches proposed for standard XSS [6–8] are no longer effective for DOM-based XSS.

Recent researches have focus on techniques aiming at DOM-based XSS [5, 9–14], and a variety of tools have been developed to detect DOM-based XSS vulnerabilities. Developers might want to know which features or components limit the performance of detection in order to improve the tool. Users might want to evaluate different tools to get a guidance in choosing which tool to use. As a point of reference for measurement, a benchmark is a good solution to these questions. A well-designed benchmark captures the dominant research direction and leads the community to problems worthy of study [15]. Unfortunately, in the area of DOM-based XSS, there is no widely used standard benchmark. In this paper, we propose our benchmark DomXssMicro, intended to fill this gap. DomXssMicro is a kind of micro benchmark instead of real-world benchmark. Micro benchmarks have been adopted widely and proved useful in various domains [16–21]. DomXssMicro is comprised of a number of test cases with desired properties generated based on a template, which is extracted from representative real-world applications. Compared with real-world application benchmark, DomXssMicro has the following features:

- Each test case focuses on a specific property, thus having a clear test goal which is more convenient in evaluation.
- The template consists of six abstract components representing different aspects of the DOM-based XSS vulnerability. Therefore, the benchmark is easy and flexible to extend by adding extra instances for each component.
- The components of the template are orthogonal. Test cases are generated by instantiating each component iteratively, thus having a better coverage of tested properties.
- The test cases omit all vulnerability-irrelevant language features and external dependencies, making them easier to deploy.
- The test cases scale down from realistic applications to small ones, which reduces the cost of detection and

facilitates rapid testing.

Our goal is to complement rather than compete real-world vulnerability benchmark, providing researchers and practitioners with diverse options for evaluating DOM-based XSS detection tools. At present, DomXssMicro is far from comprehensive. However, it is still useful as a baseline. Passing all test cases in DomXssMicro does not mean the tool is perfect. But failing in any test case will reveal certain weakness of the tool, which is helpful in guiding future improvement. We hope DomXssMicro can provide the basis for further development and discussion in the community.

This paper makes the following contributions:

- A template modeling common DOM-based XSS vulnerabilities is proposed, which generalizes various aspects of the vulnerability, including Source, Propagation, Transformation, Sink, Trigger and Context.
- A micro benchmark DomXssMicro consisting of 175 test cases is proposed based on the template for evaluation of DOM-based XSS detection.
- An empirical study is conducted to evaluate six state-of-the-art DOM-based XSS detection tools using DomXssMicro.

The rest of this paper is organized as follows. In Section II, the technique details of DOM-based Cross-site Scripting are introduced. In Section III we describe the construction process of our micro benchmark and implementation details. Section IV presents the setup and results of our empirical evaluation to illustrate the use of our micro benchmark. Related work is reviewed in Section V and we conclude in Section VI.

II. DOM-based Cross-Site Scripting

A. Definition

The Document Object Model (DOM) is a kind of data model which is widely adopted in the Web and used for representing and interacting with objects in a document, especially in the HTML document of the Web page. Generally speaking, every HTML document has a relevant DOM which comprises of document objects in a structural format, and each object is associated with document properties used for the browser. The DOM enables scripts such as JavaScript to interact with objects of an HTML document. Once a dynamic script is run in the browser, it will be provided with the DOM of the Web page where it executed by the browser. Therefore, the script can access to the properties of its host page, even change some of them, thus enhancing the flexibility of the script to a great extent.

Besides the benefits it offers, the DOM also suffers from several security issues. One of them is called DOM-based XSS vulnerability. DOM-based XSS is a specially type of cross-site scripting. It occurs due to inappropriate handling of user input. The input is then used for some parts of dynamic scripts, typically JavaScript, which manipulate the DOM of the Web page. With specially crafted requests, the attacker can modify the DOM maliciously, leading to execution of arbitrary scripts in the victim's browser. The process of exploiting a DOM-based XSS vulnerability [22] is shown in Fig. 1. The DOM-based XSS is distinct from standard XSS, namely the reflected

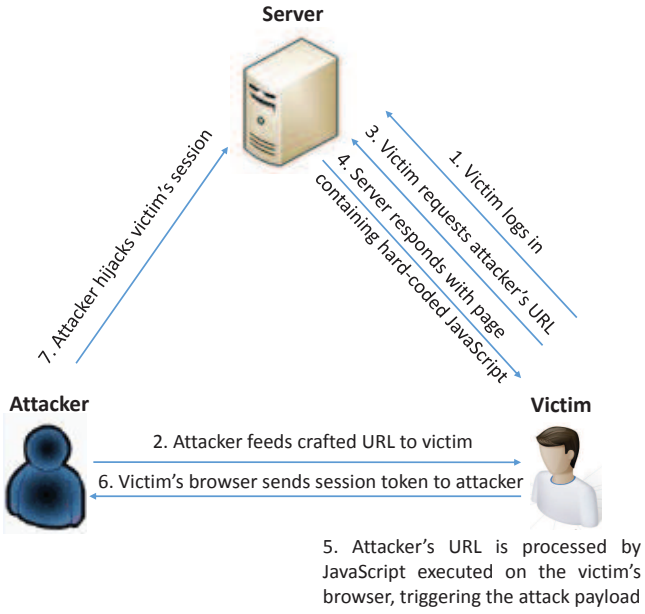


Fig. 1. Process of exploiting a DOM-based XSS vulnerability

XSS and the stored XSS, mainly because the server is not participating in the occurrence of the DOM-based XSS. The page is not a new one returned by the server. Instead, the page performs differently only because of the modifications in the DOM environment caused by malicious input requests.

B. Example

To illustrate the DOM-based XSS more specifically, a representative example is presented in Fig. 2, which is the HTML code for a vulnerable page. In this example, the Web page is able to provide different users with customized greetings by using dynamic scripts. The user name is contained in the URL, which is used by the script to generate the resulting page. Assume the address of the Web page is *http://www.ourwebsite.com/welcome.html*. Normally, a benign URL is something like:

http://www.ourwebsite.com/welcome.html?name=Peter

In this case the greeting becomes "Hello, Peter!". However, the attacker might also inject malicious script into the URL as follows:

http://www.ourwebsite.com/welcome.html?name=<SCRIPT>SomethingEvil</SCRIPT>

Once the URL is sent to the victim and requested by the victim's browser, the response from the server is the same static HTML page shown in Fig. 2 without any special processing of the malicious script embedded in the URL. When the browser builds the DOM using the URL, the malicious script is then injected into the page content and executed immediately, thus accomplishing the DOM-based XSS attack. In reality, a variety of advanced techniques might be adopted by the attacker such as encoding to hide its malicious intent and to evade the detection.

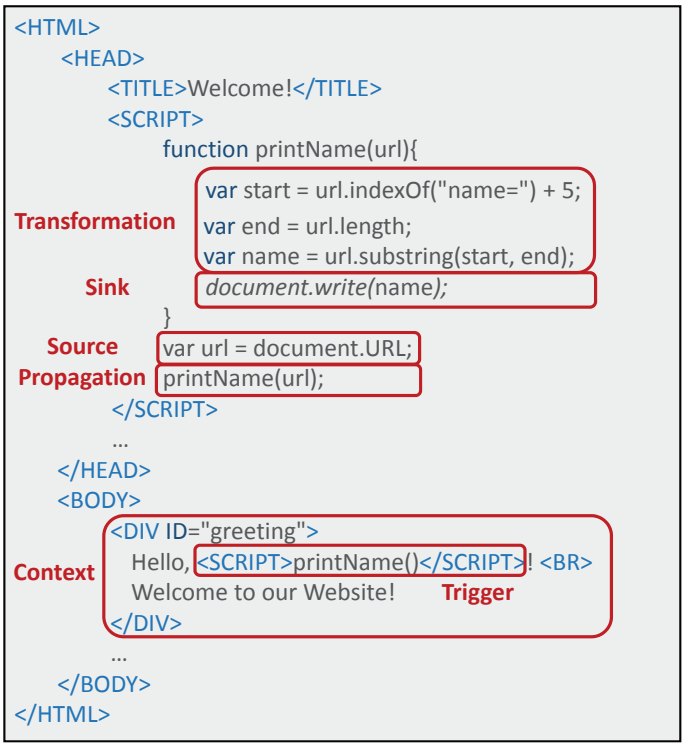


Fig. 2. An example and the template of the HTML page which is vulnerable to the DOM-based XSS

III. Micro Benchmark Construction

In order to evaluate existing detection tools and offer guidance for further improvement, benchmarks are needed. In this section, we will introduce how our micro benchmark is constructed.

A. Overview

The overview of our micro benchmark construction process is illustrated in Fig. 3. First of all, we summarize several representative real-world DOM-based XSS vulnerabilities and extract an abstract template from them. The template contains six components, each one of which have several instances. Then, a benchmark test case can be generated by instantiating each component with one concrete instance.

B. Benchmark Template

Our micro benchmark construction is a template-based approach which extracts a common template from typical cases and generates test cases based on the template. As a representative example, the HTML code shown in Fig. 2 includes most of the essential parts of the HTML page that are vulnerable to the DOM-based XSS. From this example, a set of primary components of the template can be extracted, including Source, Propagation, Transformation, Sink, Trigger and Context. These components orthogonally represent different aspects of the DOM-based XSS vulnerability. By instantiating these components, various test cases can be generated, aiming at evaluating different detection abilities supported by a tool.

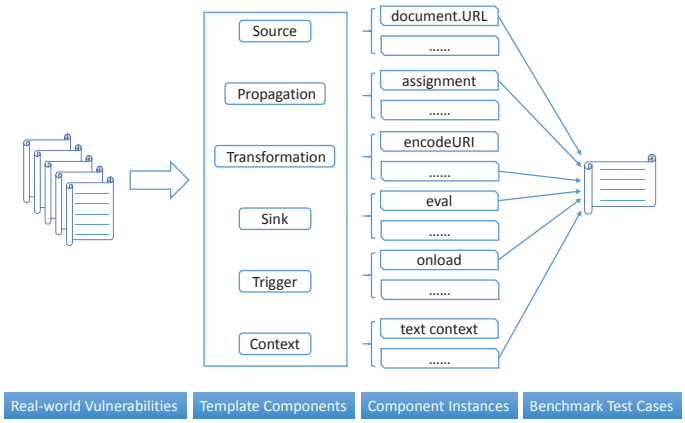


Fig. 3. Overview of micro benchmark construction process

C. Benchmark Components

To generate test cases in our micro benchmark, the template is concretized by various instances of each component [23] [24]. We introduce instances of each component in detail.

1) Source: The Source component represents the source locations of incoming data. In our micro benchmark, we considered sources as follows.

- URL and location information such as *document.URL*, *window.location.href*, *window.location.search*, *window.location.hash*.
- Cookie information from *document.cookie*.
- Client storage such as *localStorage* and *sessionStorage*.
- Navigation related information such as *document.referer* identifying the address of the page that is linked to the resource being requested and *history.state* representing the navigation history of the browser.
- User submitted data by the *form*.
- Messages passed from other domains such as *postMessage*, *XMLHttpRequest* and *JSONP*.
- Safe instances such as *window.status* and *document.domain*.

2) Propagation: The Propagation component represents the statements which propagate tainted values from the source to the sink without modifying them. This component mainly considers the language features and information flows that affect the propagated tainted values, including:

- Variable assignment.
- Object field alias.
- Function parameter alias.
- Array access.
- Control flow decision.
- Dead code.

Each one of them has its vulnerable and safe form.

3) Transformation: The Transformation component represents the statements which modify tainted values through operations such as string concatenation, modification and truncation. Transformations considered in our micro benchmark are as follows.

- Escaping or removing special characters such as backslash, quotations, parentheses, the great-than and less-than symbols, and white spaces.
- Removing dangerous HTML tags such as *script* and dangerous attributes such as *href* and *src*.
- Sanitization functions provided by JavaScript such as *escape()*, *encodeURI()* and *encodeURIComponent()*.
- Customized sanitization functions using *replace* function.

4) Sink: The Sink component represents the statements that actually perform the malicious behaviors using the tainted values propagated from the Source component. In our micro benchmark, we considered:

- Script execution function such as *eval()*, *setTimeout()*, *setInterval()*, *addEventListener()* and *execScript()*.
- Modifying script source attributes such as *script.src*, *script.text*, *script.textContent* and *script.innerText*.
- Location-changing functions such as *location.assign()* and *location.replace()*.
- Modifying location and address attributes such as *location.href*, *location.pathname*, *src* and *href*.
- Document modification function such as *document.write()* and *document.writeln()*.
- Document modification attributes such as *innerHTML* and *outerHTML*.
- Modifying cookie information with *document.cookie*.
- Modifying client storage such as *localStorage* and *sessionStorage*.
- Passing messages to other domains such as *postMessage* and *XMLHttpRequest*.
- Safe instances such as *document.getElementById()*, *window.moveBy()* and *window.scrollTo()*.

5) Trigger: The Trigger component represents how the scripts are triggered and executed. There are three types of triggers:

- The scripts are executed immediately while the Web page is loaded.
- The scripts are executed only when certain actions (e.g. mouse clicking, hovering and keyboard typing) are performed or certain events (e.g. timer event, DOM modification event and page load event) are triggered.
- The scripts are not executed.

6) Context: The Context component represents the surrounding context where the tainted content locates. Contexts considered in our micro benchmark include:

- Text context where the tainted content is part of the text value of an HTML tag.
- Attribute context where the tainted content exists in an attribute of an HTML tag. The attribute might be surrounded by single quotations, double quotations or no quotations.
- Comment context where the tainted content is part of a piece of comment.
- Style context where the tainted content locates in the style attribute which permits CSS syntax.
- URL context where the tainted content is part of an URL determining an address.
- Script context where the tainted content is part of JavaScript code.

D. Vulnerability Verdict

In a vulnerable page, all six components should satisfy certain conditions which are listed in Table I. Any unsatisfied component instance will result in an invulnerable page. According to these conditions, we can classify a component instance into three classes: vulnerable, safe and conditionally vulnerable. A component instance is conditionally vulnerable means it is vulnerable under certain conditions depending on other component instances. In order to determine whether a page is vulnerable conveniently, only vulnerable and safe component instances are selected in our benchmark. In this way, a page is safe so long as one component instance is safe; a page is vulnerable only when all component instances are vulnerable.

E. Benchmark Generation

Since the components of template are orthogonal, test cases can be produced by different combination of component choices. To make each test case have a clear test target, we adopt control variate method. First, we choose a set of default instances for each component, shown in Table I. All these instances are vulnerable as well as the most easily detected ones. Then, test cases are generated by iteratively varying the instance of one component while keeping instances of other components. In this way, every component instance is included in one test case where the feature represented by the instance will be evaluated. Considering that all Context instances are

TABLE I. DEFAULT SETTINGS AND VULNERABLE CONDITIONS OF EACH COMPONENT

Component	Default Setting	Vulnerable Condition
Source	docoment.URL	string inputs controlled by the user
Propagation	identical assignment	propagate the flow from source to sink
Transformation	none	not sanitize the tainted data
Sink	document.write	operations that might cause security issues
Trigger	onload	execute the malicious script
Context	text context	exploitable by certain methods

exploitable by certain methods, negative test cases are extended by changing the Transformation instances according to the Context instances. Eventually, 175 test cases including 115 vulnerable ones and 60 invulnerable ones are produced, which consist our micro benchmark.

IV. Evaluation

In this section, we describe our empirical experiments conducted to explore the usefulness of DomXssMicro for the evaluation and benchmarking of DOM-based XSS detection tools, and discuss how to guide tool users and developers.

A. Evaluated Tools

To demonstrate our micro benchmark, we chose a set of DOM-based XSS detection tools as our evaluation objects. These evaluated tools cover both commercial tools and open-source tools, which apply various approaches to identify DOM-based XSS vulnerabilities including static, dynamic and hybrid analysis. All tools were evaluated with their default settings without further optimization because our goal is demonstrating the use of micro benchmark rather than comparing these tools comprehensively. The detailed techniques applied by each tool are introduced as follows.

1) IBM AppScan: IBM AppScan (*appscan* for short) [25] adopts a hybrid JavaScript analysis, which enhances static string analysis by partially evaluating the code and concretizing dynamic contexts based on concrete information retrieved during crawling. It targets for various client-side vulnerabilities including DOM-based XSS.

2) StaticBurp: StaticBurp (*burp*) [26] is a passive scan plugin of Burp Suit for DOM-based XSS detection. It uses regular expressions from various sources and tuned extensively to eliminate false positives.

3) IronWASP: IronWASP (*ironwasp*) [27] analyzes the JavaScript source codes to identify tainted sources and track traces forward, or identify tainted sinks and track traces backward. The intersection between traces in two directions is the tainted flow vulnerable to DOM-based XSS.

4) ScanJS: ScanJS (*scanjs*) [28] first converts JavaScript source codes into Abstract Syntax Tree (AST), then walks through the AST matching patterns to detect DOM-based XSS based on a set of predefined rules.

5) OWASP Xenotix: Xenotix (*xenotix*) [29] adopts various regular expressions representing a comprehensive set of tainted sources and sinks patterns that are vulnerable to DOM-based XSS. Both StaticBurp and Xenotix use regular expressions to detect vulnerabilities, but the patterns represented by regular expressions are different.

6) Tainted PhantomJS: Tainted PhantomJS (*tpjs*) [30] is built based on the headless browser engine PhantomJS to emulate the browser navigation and interaction. The inbuilt WebKit and JavaScriptCore are instrumented to enable taint tracking and DOM-based XSS detection. Tainted flows are tagged “Suspicious”, and an optional exploitation step can be performed to make the suspicious flow “Confirmed”.

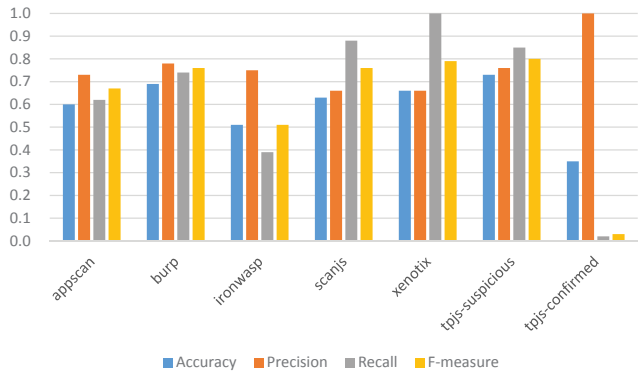


Fig. 4. The overall detection performance of each tool

B. Evaluation Metrics

To evaluate the performance of each DOM-based XSS detection tool, we adopt detection results in the format of confusion matrix [31]. There are four possible outcomes for each test case. A Web page might be detected as vulnerable to DOM-based XSS when it is truly vulnerable (true positive, TP); it might be detected as vulnerable when it is actually safe (false positive, FP); it might be detected as safe and it is truly safe (true negative, TN); or it might be detected as safe when it is actually safe (false negative, FN).

According to the confusion matrix, we can compute four widely adopted metrics, that is Accuracy, Precision, Recall and F-measure [32]. The accuracy measures the number of correctly identified Web page, including both vulnerable and safe ones, over the total number of test cases. The precision measures the percentage of correctly detected vulnerable Web pages over all of the pages identified as vulnerable. The recall measures the percentage of correctly detected vulnerable Web pages over all of the actually vulnerable pages. The F-measure is a composite measure that measures the weighted harmonic mean of precision and recall. The definitions of these four metrics are as follows:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

$$Precison = \frac{TP}{TP + FP}$$

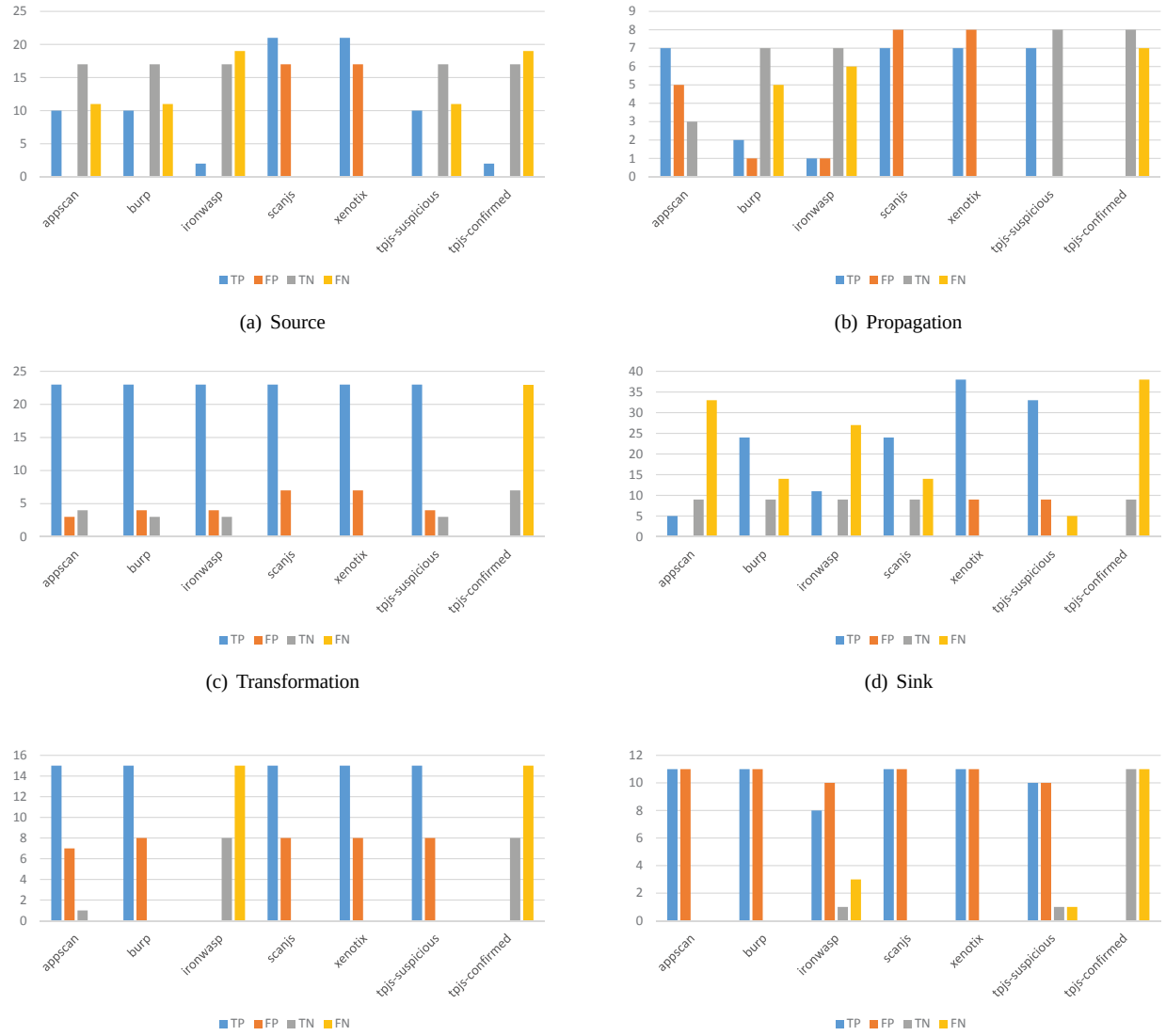
$$Recall = \frac{TP}{TP + FN}$$

$$F - measure = \frac{2 * Precision * Recall}{Precision + Recall}$$

C. Result and Discussion

The overall detection performance of each tool in evaluating our micro benchmark is presented in Fig. 4. In Accuracy and F-measure, *tpjs-suspicious* performs the best. In Precision, *tpjs-confirmed* is the best. In Recall, *xenotix* achieves the best performance.

From the overall results, we can also observe the strategy applied by each tool, which may guide us in choosing detection tools. Generally speaking, Precision and Recall are a pair of



(a) Source (b) Propagation

(c) Transformation (d) Sink

(e) Trigger (f) Context

Fig. 5. Detailed detect results of each tool in each component

contradictory metrics, a tool has to sacrifice one metric to achieve a better value of the other. *Ironwasp* and *tpjs-confirmed* are Precision-preferred tools which have significant higher Precision than Recall. *Tpjs-confirmed* has 100% Precision because it confirms every suspicious page by exploiting it, therefore eliminating all false positives. Its low Recall illustrates that the exploitation techniques adopted by it is not powerful enough thus missing a lot of vulnerable pages. *Scanjs* and *xenotix* are Recall-preferred tools which sacrifice Precision for Recall. *Xenotix* would rather to get a thousand killed than to get one missed by identifying all test cases as vulnerable, resulting in zero true negative and false negative, thus achieving 100% Recall. *Appscan*, *burp* and *tpjs-suspicious* are balance tools which try to make a tradeoff between Precision and Recall. Among them, *tpjs-suspicious* achieves the best balance, which can be seen from its high Accuracy and F-measure values. Tool users should choose an appropriate tool of the right type according to their application contexts and requirements.

For further analysis, detailed detection results are reported in Fig. 5, which provide the confusion matrix values of each tool partitioned into six categories by component. By inspecting what kind of false positives and false negatives are produced, the shortcomings of each tool are revealed as a guidance for further improvement.

Source. Some simple and straightforward sources are identified by all tool. But some advanced or indirect sources are missed by most tools, such as client-side storage and sources from message passing mechanism, producing false negatives. *scanjs* and *xenotix* are able to detect them due to their conservative strategy at the cost of many false positives.

Propagation. Except for *tpjs-suspicious*, all tools have several false positives or false negatives in Propagation. The good performance of *tpjs-suspicious* is owed to its taint analysis module built into the JavaScript interpretation engine, which is worth other tools using for reference.

Transformation. All tools are not able to distinguish built-in sanitization functions and customized sanitization functions. *Tpjs-confirmed* suffering from many false negatives because of not able to exploit vulnerable customized sanitization functions, while other tools produce several false positives due to not able to filter out safe customized sanitization functions. Symbolic execution and SMT (satisfiability modulo theories) solver [33] might be candidate techniques for improving existing tools in this direction.

Sink. Just like the sources, a lot of advanced or indirect sinks are not detected by most tools. The reason might be these sources and sinks are not very common and not easy to exploit, thus these tools choose efficiency at the cost of coverage. However, in security-critical applications, these tools will be not competent enough. The tradeoff should be made by the tool users.

Trigger. All evaluated tools are either not able to execute scripts that have to be triggered under certain events which causes false negatives, or execute all scripts even when the event can never be triggered, producing false positives. For improvement, the tool should adopt real events to try to execute possible scripts, therefore triggerable and untriggerable events can be distinguished.

Context. All tools except for *tpjs-confirmed* do not consider whether the context is exploitable, resulting in many false positives. *Tpjs-confirmed* considers the exploitation problem, but its ability of exploitation is not powerful enough to exploit many vulnerable pages, therefore a lot of false negatives are produced. A more sophisticated exploitation technique is required for further improvement in the future.

V. Related Work

In the area of DOM-based XSS, several researches have been proposed to deal with this vulnerability. The term DOM-based XSS was first introduced by Klein [4] in 2005. In that paper, the concept, examples, evading techniques and effective defenses are discussed in detail. Lekies et al. [5] proposed a fully automated system to detect DOM-based XSS vulnerabilities, consisting of a dynamic byte-level taint-aware JavaScript engine covering all JavaScript features and full DOM API along with a fine-grained context-aware payload generation technique. A large scale empirical experiment was also conducted, finding around 25 million suspicious flows. Based on the conceptual shortcoming analysis of current client-side XSS filters and empirical validation by automatic filter bypass generation, Stock et al. [11] proposed a precise client-side protection approach against DOM-based XSS using character-level taint tracking along with the taint-enhanced HTML and JavaScript parsers. In this way, all flows of attacker-controlled data will be tracked and detected, and malicious injection attempts will be stopped during parse time. In their recent work [12], they conducted an empirical study investigating a large set of real-world clientside XSS vulnerabilities and characterizing features of them. Based on these findings, a set of metrics measuring complexity are proposed to gain insights of the relationship between the complexity and vulnerability. Parameshwaran et al. [9] developed a system called DexterJS, which synthesizes patches for DOM-based XSS vulnerabilities automatically. DexterJS performs fine-grained taint analysis by a JavaScript rewriting engine without any assumption of the browser environment. Saxena et al. [10] proposed FLAX, aiming at discovering a serial of client-side validation vulnerabilities including DOM-based XSS. FLAX uses taint enhanced black box fuzzing, which combines automated random fuzzing with dynamic taint analysis. Targeting for hardening client-side validation vulnerabilities, ZigZag [14] performs dynamic invariant detection through instrumentation of client-side code, and derives models of normal behaviors. Deviation from these models are highly suspicious of vulnerable to client-side valid. This approach is able to defend against previously unknown attacks. Tripp et al. [13] proposed a hybrid security analysis combining static analysis and dynamic analysis. In their approach, string analysis is used to evaluate the JavaScript content partially based on the dynamic information obtained during crawling. Thus the dynamic contexts can be concretized, which facilitates further analysis. However, none of the existing researches investigates benchmark issues of DOM-based XSS systematically.

VI. Conclusion

We have introduced DomXssMicro, a micro benchmark for evaluating DOM-based XSS vulnerabilities, consisting of 175 test cases. We have also presented an evaluation of six DOM-based XSS detection tools to demonstrate the use of DomXssMicro. In the future, we plan to extend our benchmark with more comprehensive instances of each component, such as more complex language features, the involvement of Web frameworks and libraries and the impact of browser quirks. We hope our DomXssMicro makes useful contribution to the community in further ongoing development and discussion of a standard benchmark for DOM-based XSS detection.

Acknowledgment

This research was supported in part by grants from National Natural Science Foundation of China (Nos. 61379054, 61502015 and 91318301), and Program for New Century Excellent Talents in University.

References

[1] S. Hanna, R. Shin, D. Akhawe, A. Boehm, P. Saxena, and D. Song, "The emperors new apis: On the (in) secure usage of new client-side primitives," in *Proceedings of the Web*, vol. 2, 2010.

[2] S. Lekies and M. Johns, "Lightweight integrity protection for web storage-driven content caching," in *6th Workshop on Web*, vol. 2, 2012.

[3] S. Son and V. Shmatikov, "The postman always rings twice: Attacking and defending postmessage in html5 websites." in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2013.

[4] A. Klein, "Dom based cross site scripting or xss of the third kind," *Web Application Security Consortium*, vol. 4, 2005.

[5] S. Lekies, B. Stock, and M. Johns, "25 million flows later: large-scale detection of dom-based xss," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 1193–1204.

[6] M. Johns, B. Engelmann, and J. Posegga, "Xssds: Server-side detection of cross-site scripting attacks," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. IEEE, 2008, pp. 335–344.

[7] R. Pelizzi and R. Sekar, "Protection, usability and improvements in reflected xss filters." in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2012, p. 5.

[8] A. Doupé, W. Cui, M. H. Jakubowski, M. Peinado, C. Kruegel, and G. Vigna, "dedacota: toward preventing server-side xss via automatic code and data separation," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 1205–1216.

[9] I. Parameshwaran, E. Budianto, S. Shinde, H. Dang, A. Sadhu, and P. Saxena, "Auto-patching dom-based xss at scale," *Proceedings of the Foundations of Software Engineering (FSE)*, 2015.

[10] P. Saxena, S. Hanna, P. Poosankam, and D. Song, "Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications." in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2010.

[11] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns, "Precise client-side protection against dom-based cross-site scripting," in *Proceedings of the 23rd USENIX security symposium*, 2014, pp. 655–670.

[12] B. Stock, S. Pfistner, B. Kaiser, S. Lekies, and M. Johns, "From facepalm to brain bender: Exploring client-side cross-site scripting," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 1419–1430.

[13] O. Tripp, P. Ferrara, and M. Pistoia, "Hybrid security analysis of web javascript code via dynamic partial evaluation," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 49–59.

[14] M. Weissbacher, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Zigzag: automatically hardening web applications against client-side validation vulnerabilities," in *Proceedings of the 24th USENIX Security Symposium*, 2015, pp. 737–752.

[15] S. E. Sim, S. Easterbrook, and R. C. Holt, "Using benchmarking to advance research: A challenge to software engineering," in *Proceedings of the 25th International Conference on Software Engineering*. IEEE Computer Society, 2003, pp. 74–83.

[16] L. Afanasiev, I. Manolescu, and P. Michiels, "Member: a micro-benchmark repository for xquery," in *Database and XML Technologies*. Springer, 2005, pp. 144–161.

[17] R. Taylor and X. Li, "A micro-benchmark suite for amd gpus," in *Proceedings of the 39th International Conference on Parallel Processing Workshops (ICPPW)*. IEEE, 2010, pp. 387–396.

[18] M. Banikazemi, J. Liu, S. Kutlug, A. Ramakrishnan, P. Sadayappan, H. Shah, and D. Panda, "Vibe: A micro-benchmark suite for evaluating virtual interface architecture (via) implementations," in *Proceedings of the 15th International Parallel and Distributed Processing Symposium*. IEEE, 2001, pp. 8–pp.

[19] D. Bureddy, H. Wang, A. Venkatesh, S. Potluri, and D. K. Panda, *OMB-GPU: a micro-benchmark suite for evaluating MPI libraries on GPU clusters*. Springer, 2012.

[20] N. S. Islam, X. Lu, M. Wasi-ur Rahman, J. Jose, and D. K. D. Panda, "A micro-benchmark suite for evaluating hdfs operations on modern clusters," in *Specifying Big Data Benchmarks*. Springer, 2014, pp. 129–147.

[21] S. Ray, B. Simion, and A. D. Brown, "Jackpine: A benchmark to evaluate spatial database performance," in *Proceedings of the 27th International Conference on Data Engineering (ICDE)*. IEEE, 2011, pp. 1139–1150.

[22] D. Stuttard and M. Pinto, *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. John Wiley & Sons, 2011.

[23] W3school. [Online]. Available: http://www.w3school.com.cn/index.html

[24] Dom xss test cases wiki project. [Online]. Available: https://code.google.com/p/domxsswiki

[25] Ibm security appscan standard. [Online]. Available: http://www.ibm.com/software/products/en/appscan

[26] burp static scan. [Online]. Available: https://github.com/tomsteele/burpstaticscan

[27] Iron web application advanced security testing platform. [Online]. Available: https://ironwasp.org

[28] Static analysis tool for javascript code. [Online]. Available: https://github.com/mozilla/scanjs

[29] Owasp xenotix xss exploit framework. [Online]. Available: https://www.owasp.org/index.php/OWASP-Xenotix-XSS-Exploit-Framework

[30] Tainted phantomjs. [Online]. Available: https://github.com/neraliu/tainted-phantomjs

[31] S. V. Stehman, "Selecting and interpreting measures of thematic classification accuracy," *Remote sensing of Environment*, vol. 62, no. 1, pp. 77–89, 1997.

[32] J. Huang, *Performance measures of machine learning*. University of Western Ontario, 2006.

[33] S. Prateek, A. Devdatta, H. Steve, M. Feng, M. Stephen, and S. Dawn, "A symbolic execution framework for javascript," in *IEEE Symposium on Security and Privacy*, 2010.