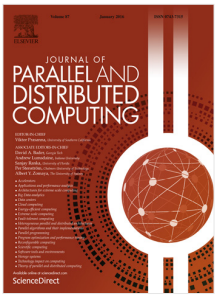
Accepted Manuscript



TT-XSS: A novel taint tracking based dynamic detection framework for DOM Cross-Site Scripting

Ran Wang, Guangquan Xu, Xianjiao Zeng, Xiaohong Li, Zhiyong Feng

PII: S0743-7315(17)30218-6
DOI: http://dx.doi.org/10.1016/j.jpdc.2017.07.006
Reference: YJPDC 3719

To appear in: *J. Parallel Distrib. Comput.*

Received date : 19 January 2017
Revised date : 9 June 2017
Accepted date : 28 July 2017

Please cite this article as: R. Wang, G. Xu, X. Zeng, X. Li, Z. Feng, TT-XSS: A novel taint tracking based dynamic detection framework for DOM Cross-Site Scripting, *J. Parallel Distrib. Comput.* (2017), http://dx.doi.org/10.1016/j.jpdc.2017.07.006

TT-XSS: a Novel Taint Tracking Based Dynamic Detection Framework for DOM Cross-Site Scripting

Ran Wang, Guangquan Xu*, Xianjiao Zeng, Xiaohong Li, Zhiyong Feng

[a] *Tianjin Key Laboratory of Advanced Networking (TANK), School of Computer Science and Technology, Tianjin University, Tianjin, China 300350.*

Abstract

Most work on DOM Cross-Site Scripting (DOM-XSS) detection methods can be divided into three kinds: black-box fuzzing, static analysis, and dynamic analysis. However, black-box fuzzing and static analysis suffer much from high false negative rates and high false positive rates respectively. Current dynamic analysis is complex and expensive, though it can obtain more efficient results. In this paper, we propose a dynamic detection framework (TT-XSS) for DOM-XSS by means of taint tracking at client side. We rewrite all JavaScript features and DOM APIs to taint the rendering process of browsers. To this end, new data types and methods are presented to extend the semantic description ability of the original data structure, based on which we can analyze the taint traces through tainting all sources, sinks and transfer processes during pages parsing. In this way, attack vectors are derived to verify the vulnerabilities automatically. Compared to AWVS 10.0, our framework detects more 1.8% vulnerabilities, and it can generate the corresponding attack vectors to verify 9.1% vulnerabilities automatically.

Keywords:
DOM Cross-Site Scripting, Static Analysis, Dynamic Analysis, vulnerabilities

1. Introduction

Cross-site Scripting(XSS) attack is one of the most important security issues in today's Web applications, such as cloud storage[1]. It will lead to

*losin@tju.edu.cn

serious user privacy leaks, and even users will be infected with worms. XSS is often included in OWASP[2](Open Web Application Security Project) TOP 10. XSS vulnerabilities are usually classified into three types: Reflected XSS, Stored XSS and DOM-XSS. This is because the principle of DOM-XSS is completely different from others. Therefore, the detection methods of different types are not the same. There are some researches on Reflected XSS and Stored XSS vulnerabilities detection, however, they are not suitable for the DOM-XSS. On the other way, increasing Web application makes the DOM-XSS popular. Hence, an appropriate DOM-XSS detection method is necessary

Most current work focuses on black-box fuzzing and static analysis, but there are limits to accuracy in theory. Black-box fuzzing relies heavily on fuzz datasets, which is difficult to cover all cases due to a variety of situations. Static analysis encountered many difficulties during the process, such as code ambiguity, code obfuscation[3]. Dynamic analysis can obtain more efficient results, however, dynamic detection method is complex and expensive to be implemented. To this end, we propose a new dynamic detection framework based on taint tracking[4][5]. Our contributions are as follows. (1) New data types and methods are advanced to be used in tainting browsers' rendering process. (2) Attack vectors are derived to verify the vulnerability automatically. (3) A prototype system is implemented to verify our proposed DOM-XSS detection method by modifying the JavaScriptCore and Webkit in WebCore engine.

The rest of the paper is organized as follows. Section 2 reviews the related work. Section 3 describes the framework for dynamic DOM-XSS detection, and modifies browsers' rendering engine to verify vulnerabilities automatically. Section 4 implements a prototype system. Our method is verified by experimental results in Section 5. Section 6 are conclusion and future work.

2. Related work

On the one hand, part functions, once realized at the server in the traditional Web application, have been moved to the client gradually. On the other hand, JavaScript code becomes more and more popular, and the functions of JavaScript code are more complex since modern browsers have supported HTML5 API. All of these cause more and more serious security problems[5][6]. DOM-XSS is a pure client-side security issue, which is also known as the third kind of XSS. In other words, it does not need to be di-

rectly involved in the analytical response of the server, but triggers XSS only by the DOM parsing of browsers. Compared with the server, Web client applications are facing great challenges in the static and dynamic security testing process, whose main reasons include three facts. First, all server code is completely under control of Web application operators, which can be processed, monitored and analyzed directly, while the client code can only be run on the user computer. Second, compared with C# and Java commonly used by the server, JavaScript code often regards string data as executable code by *eval* or other API[7], during which the server is completely invisible. Third, a large number of modern Web applications use the third party JavaScript code, which is not controllable and visible for the server.

Hydara[8] et al. analyzed lots of related work on XSS before 2013, but they found the study about DOM-XSS only accounted for 0.9%. A static analysis and dynamic information flow tracing method were proposed by Vogt[9] to reduce the risk of XSS. However, they focused on sensitive information, such as user cookie, or potential information leakage, rather than tracking unreliable data, and can not find out cause either vulnerabilities. Ra.2[10] is a Firefox plug-in for DOM-XSS detection, which adopts black box fuzzing. DOMinator is the first tool for DOM-XSS detection based on taint tracking, which is implemented by modifying SpiderMonkey JavaScript engine of Firefox. However, it can not realize automatic vulnerabilities scanning. Saxena[11] proposed a method of black box Fuzzing[12] based on taint enhancement. In this work, dynamic taint analysis is combined with automatic random Fuzzing technology. At the same time, the corresponding prototype tool FLAX was developed. Criscione[13] proposed a automation tool based on black box to find XSS, and it used a real browser to test and verify vulnerabilities. Lekies[14] et al. implemented a DOM-XSS detection and verification method based on dynamic taint tracking by modifying the Chromium. Parameshwaran[15] et al. implemented a testing platform called DEXTERJS. It is a kind of dynamic taint tracking method based on byte level to obtain and verify potential dataflow of vulnerabilities caused by Web page. In order to patch DOM-XSS automatically, they proposed a scheme[16] which do not need to modify the browsers' code. DOMXSS Scanner[17] is a tool to check web pages source code with DOM XSS sources and sinks without vulnerabilities detection.

At present, there are several directions in the research of DOM-XSS, including black box testing, static analysis[18] and dynamic analysis. Because black box testing is limited to coverage of attack vector, it suffers much from

false negative. Static analysis method can solve the common problems, when it applies to the complex situation, its accuracy rate is low, and the false positive rate is higher. In this paper, on the basis of dynamic analysis, we leverage dynamic taint tracking technology to the detection and verification of DOM-XSS vulnerabilities.

3. Detection framework for DOM-XSS

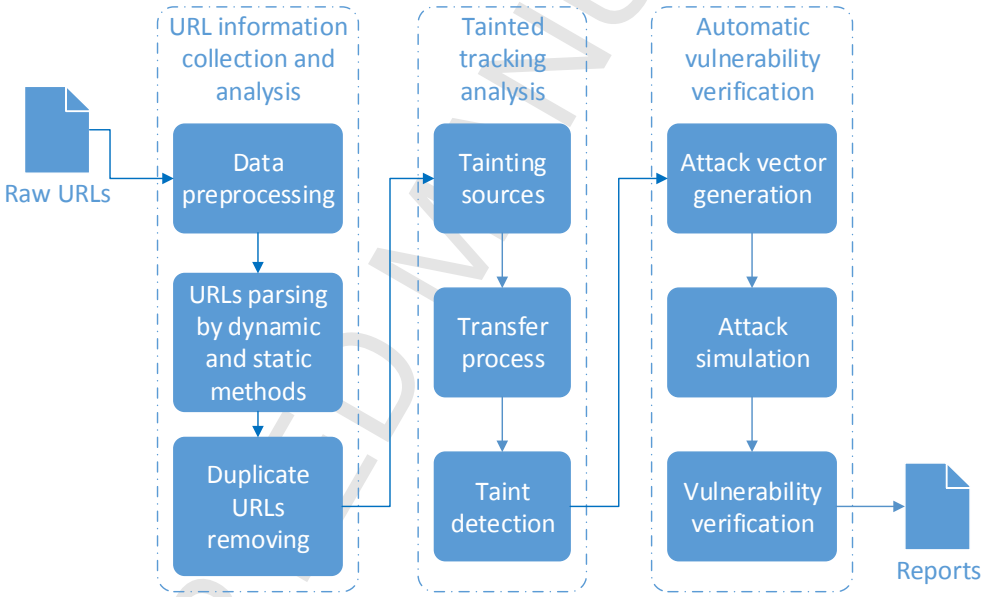


Figure 1: Detection process of TT-XSS

In order to detect DOM-XSS in Web applications automatically, this paper simulates real browsers to render Web pages. To tag the input data, our method modifies the JavaScriptCore and WebKit engine of PhantomJS. As shown in Figure 1, our dynamic DOM-XSS detection framework is mainly composed of three modules: URL information collection and analysis, taint tracking analysis and automatic vulnerability verification.

Raw URLs are entered into the framework as input. Then, the URLs are parsed by dynamic and static methods after being preprocessed. Next, taint

traces are obtained from pages of URLs in taint tracking analysis module. Finally, vulnerabilities are automatically verified and reported.

3.1. URL information collection and analysis

This module is similar to the website crawler. It maintains a URLs queue initialized with input URLs. A browser is called to parse URLs, render pages and gather the information of pages when the queue is not empty. The information of all relevant pages is collected, as URLs in new pages are added to the queue.

This module focuses on (1) parsing the URLs by dynamic and static methods to obtain more comprehensive URLs, (2) removing duplicate URLs according to URLs and scripts of pages. Static parsing method only deals with URLs in HTTP response, while dynamic parsing method will parse pages first to catch further data. Therefore, crawled URLs are more comprehensive. Moreover, Bloom Filter[19] is used to deal with duplicate URLs efficiently.

3.2. Taint tracking analysis

Taint tracking method is the basis of taint tracking analysis processes. To detect potential controllable source-to-sink data flows, a taint tracking method is proposed. As DOM-XSS is triggered entirely in the browser side, our method modifies the JavaScriptCore and WebKit engine. The modification covers JavaScript language features and the DOM APIs associated with DOM-XSS. Therefore, our method can tag the input data and transfer tags with modified DOM APIs. According to tags, taint traces are obtained. The entire process is done while the browser runs, so it is a dynamic detection process.

Code obfuscation is an approach of protecting program. Our method will not be affected by it, as functions, e.g. *eval*, from code do not change at the client side. Moreover, a third party JS library, e.g. *JQuery*, will not affect taint tracking accuracy.

The taint tracking method mainly contains four steps: (1) Analyze and number all DOM APIs associated with DOM-XSS. (2) Design taint transfer methods and taint information storage. (3) Modify the related DOM APIs. (4) Deal with taint information.

3.2.1. Analysis of related DOM APIs

To get the full taint trace, controllable sources, sinks and transfer processes need to be modified. Any unmodified operations related to DOM-XSS will cause the suspension of taint trace. Related DOM APIs are analyzed as following.

Table 1: Possible controllable sources for DOM-XSS

JS Function	Description	Id
location	return window.location object	1
location.href	return URL	2
location.pathname	return pathname	3
location.search	return search string	4
location.hash	return anchor	5
window.name	return window name	6
document.documentURI	return document URI	7
document.referrer	return referrer URL	8
document.URL	return URL	9
document.cookie	return cookie	10

Controllable sources, as vulnerability source, refer to some of the DOM APIs user can control in the browser. The input from source will lead to vulnerability without proper treatment, so controllable source should be fully marked.

According to different objects, types of sources mainly include Location, Document, Window, etc. As shown in Table 1, most of controllable sources are from the URLs. Users are likely to bring a malicious script into to DOM by opening a link. However, doucument.referrer, window.name and doucument.cookies need to induce users for other functions, such as opening a jump page to hijack the referrer.

Taint data may be modified during transfer process, so the functions of data also need to be covered to transfer taint as expected. As shown in Table 2, there are several functions, such as strings modification functions, regular match functions, etc.

All dangerous input will not work until it turns into output, so marking sinks is also crucial. As shown in Table 3, dangerous sink DOM URIs are listed. The table mainly includes execution functions, DOM element functions, address bar modification functions, etc.

Table 2: Possible transfer functions for DOM-XSS

JS Function	Description	Id
decodeURI	decode with URI encoding	1
decodeURIComponent	decode part with URI encoding	2
Unescape	decode with URI encoding	3
-	create string	4
replace	replace string	5
concat	string concatenation	6
match	string match	7
slice	string slice	8
split	string split	9
substr	string substring	10
substring	string substring	11
lowerCase	string to lower	12
upperCase	string to upper	13
trim	remove spaces both ends	14
trimLeft	remove left spaces	15
trimRight	remove right spaces	16
RegExp.exec	regular match	17
htmlelement.innerText	set, get value from tags	18
htmlelement.outerText	set, get the value of object	19
input.value	set, get the value of input	20
textarea.value	set, get the value of textarea	21

Table 3: Possible sink functions for DOM-XSS

JS Function	Description	Id
eval	calculate and execute string	1
setTimeOut	call function after some time	2
setInterval	call function according to specified period	3
innerHTML	set or get HTML in the tags	4
window.location	browser redirect to a new page	5
location.href	browser redirect to a new page	6
location.replace	replace with a new page	7
document.write	write into document	8
document.writeln	write into document	9

3.2.2. Design of data types and methods

To track taint, we add the data types and methods for Webkit engine. Key parameters of data types are shown in Table 4. They are divided into three parts according to main function: (1) data types and methods for marking taint strings, (2) data types and methods for storing information of taint, (3) methods for outputting taint information.

Table 4: Key parameters of data types

Data Type	Parameter	Description
ExecState	Taint	JavaScript string with taint strings
TaintInfo	TaintTrace	TaintTrace instances with sink api
TaintTrace	Trace_id	current TaintTrace serial number
	Trace_func	JavaScript function execution sequence
	Trace_detail	function execution detail information
	ExecState	ExecState instance for the TaintTrace
Counter	-	singleton type as TaintTrace counter

The first part is taint strings. JavaScript string is the carrier of taint strings, and the value can be obtained from *ExecState* instance while JavaScriptCore executes functions. In order to transfer taint successfully, it is required to add *Taint* attribute to the class, add *set* and *get* methods for this attribute. At the same time, *Taint* is used to store trace number of current taint JavaScript string.

The second part is taint information. In order to achieve taint information tracking, $TaintInfo$ class, $TaintTrace$ class and $Counter$ class are designed for recording taint information, taint trace, and taint trace counter. $TaintTrace$ includes three attributes: $Trace\_id$, $Trace\_func$ and $Trace\_detail$. $Trace\_id$ is responsible for the serial number of current taint trace. Whenever there is a controllable source operation, the function of $Counter$ will be called to generate a new number. The number is used to not only mark taint trace, but assign to taint strings during the process of taint transfer. $Trace\_func$ is responsible for function execution information with string format. It is empty initially. Whenever a function is executed during process, the function number will be added to it. Through the analysis of it, we can obtain the trace of current process. Two digit hexadecimal can be used for numbering, and "ff" added to $Trace\_func$ indicates the successful output. $Trace\_detail$ is responsible for operation details with string vector format.

It is empty initially. As there is a function is executed during process, the detail information will be stored in the vector, such as parameters, function calls and string content. Through the analysis of it, detail information can be got by comparing with the same function name in $Trace\_detail$. The method for $Trace\_detail$: (1) Parameters and the number of parameters will be stored (only to keep a record of transfer operation). (2) Function number which refers to function table above will also be stored. (3) String content after operation will be the last information. That is to say, every controllable source operation generates a $TaintTrace$ instance representing a trace. Each *Document* in the process will be instantiated into a $TaintInfo$ instance. *Counter* class takes the single case model and it will generate new number when controllable source function is called.

The third part is outputting information. We add an interface called *showTaint* to *Document* object in WebCore to output information. So it is convenient to get taint information during the process of parsing on the client side.

Taint trace reaching sinks is the real potential vulnerability trace. According to design above, "ff" end presents the trace reached sink. So the trace may be cause DOM-XSS vulnerability.

3.3. Automated vulnerability verification

3.3.1. Attack vectors

The structure of attack vector is defined as follows:

$Vector ::= [PaddingBlock][ClosingBlock]Payload[ExtraBlock]$

As shown in Table 5, the content of the $PaddingBlock$ is a long string. In the process of taint transfer, there will be some string operations, such as substr, substring, etc. The block will prevent attack vector from being truncated and keep the vector correct. $ClosingBlock$ is to keep the grammar before *Payload* correct, so that malicious script can be executed. *Payload* is the key to attack vectors, and its trigger proves the existence of DOM-XSS vulnerabilities. There are three types of *Payload*, according to different output positions. "alert(1);" in JavaScript, "$\langle svg/onload = alert(1)\rangle$" in HTML, and "javascript:alert(1)" in URL. $ExtraBlock$ prevents the code after *Payload* affecting the execution of the *Payload*.

Transfer operations and sink positions determine the content of attack vectors, source positions determine the position of attack vector, and attack vectors are put in the right place to generate final test cases. Due to the variety of input types, the second order input (e.g., referrer, cookies) can

Table 5: Structure of attack vector

Definition
$PaddingBlock ::= \{a \mid b \mid c \mid ... \mid x \mid y \mid z\}$
$ClosingBlock ::= \{'); \mid ''); \mid ''>\mid '>\mid ''>\} < /a >$
$Payload ::= \{alert(1); \mid < svg/onload = alert(1) > \mid javascript : alert(1)\}$
$ExtraBlock ::= \{// \mid < \text{ " } \mid <'\}$

not be verified automatically. As shown in Table 6, the $Vector$ represents the generated attack vector. In the case of the whole URL as input, attack vectors are chosen to put in the anchor. Content of anchor does not need to interact with the server, so they are not easily influenced by the server. Moreover, due to the different URL address encoding, the string of anchor will be less encoded.

Table 6: Regulation of attack vector

JS Fuction	Sample
location location.href document.documentURI document.URL document.baseURI document.URLUnencoded	protocol://hostname[:port]/path/ [;parameters][?query]#Vector
location.pathname	protocol://hostname[:port]/Vector/ [;parameters][?query]#hash
location.search	protocol://hostname[:port]/path/ [;parameters][?Vector]#hash
location.hash	protocol://hostname[:port]/path/ [;parameters][?query]#Vector

3.3.2. Automated verification

First of all, taint trace information of will be analyzed. Then according to analysis results, attack vectors will be generated. Effective HTTP requests can be constructed to be sent to target server. Next, client will receive HTTP responses from target server, and client can render the JavaScript and simulate browser to verify vulnerabilities. If successful, taint information

report will be generated with attack vectors. Otherwise, taint information will be preserved for further manual analysis.

4. Prototype system

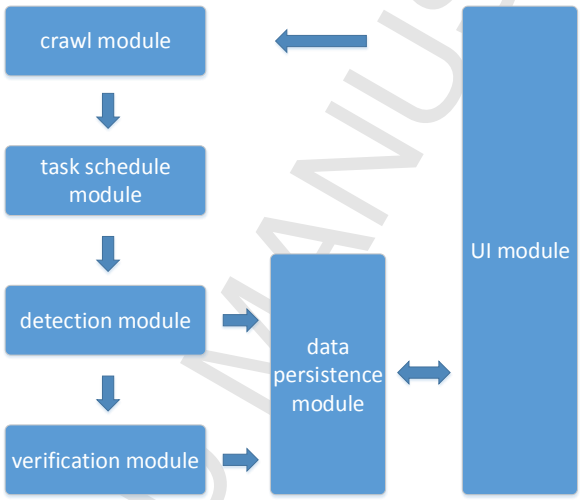


Figure 2: The taint tracking based dynamic detection framework for DOM-XSS

According to the dynamic detection method based on taint tracking in this paper, we designed and implemented a prototype framework called TT-XSS, whose architecture was shown as Figure 2. The framework is mainly composed of crawler module, task scheduling module, detection module, verification module, data persistence module and UI module.

Crawler module obtains URL information to find out where to submit malicious scripts by crawling and parsing URLs. Task scheduling module distributes task to handle the URL information collected in crawler module, and calls detection module for analyzing the URLs. Detection module will render pages to get taint traces. Then verification module processes URLs of pages which contain potential vulnerabilities. Next data persistence module stores the information. The verification module and the detection module are connected in series. According to the taint trace, it can generate corresponding attack vectors, simulate attacks, verify vulnerabilities, and store

the verification information. The UI module provides a GUI interface for the entire framework to distribute task to crawler module. And the UI module can interact with the data persistence module to view and modify the current detection.

Next, we will introduce the detection module and verification module in detail.

4.1. Detection module

By modifying the open-source browser PhantomJS, detection module can obtain taint trace for automatic verification module to generate attack vectors as expected. Although PhantomJS is a browser without a user interface, rendering the DOM, JavaScript, network access and Canvas/SVG drawing functions are complete. PhantomJS has wide applications in terms of page crawling, page output, automated tests, etc. QtWebkit is the rendering engine of PhantomJS with JavaScriptCore to parse JavaScript code. So when PhantomJS loading a Web page, like a browser with UI, JavaScript code will be executed. During implementation process, PhantomJS's WebCore and JavaScriptCore are mainly modified to achieve the goal of dynamic based on taint tracking.

Modifications occur mainly in source, transfer, and sink functions as shown in Table 1, Table 1, Table 1. At the very beginning, each trace starts with a source function and gets an instance of $TaintTrace$. So, initialize function will be added into source function. $ExecState$ will be set tainted in initialize function. At last, all sink function will call the report function and pass the instance to verification module when the trace is set tainted. In order to obtain information of taint trace, information (such as id, parameters, string content) will be added to the instance in function on the trace. Therefore, all possible trace will be reported and wait to be verified.

4.2. Verification module

After obtaining one or more dangerous traces in the detection module, these potential vulnerabilities page will conduct vulnerability verifications.

Through the analysis of dangerous trace information in detection module, detection reports will be stored in the form of XML. The detection reports include the controllable source operations, transfer process and sink operations. For the XML content, it will firstly extract the controllable source operations and sink operations. Then match and analyze the transfer process through the regular expression to record variables after string truncation

and concatenation operation. According to Table 5 and Table 6, the attack vector and the attack position will be determined. Finally, it will generate test URLs.

During verification process, this module calls the PhantomJS API to implement a validation script. Then it submits the corresponding test URL, *Cookie* to verify the script. The script will resolve the URL and call *page.onAlert*. The vulnerability is verified when alert event is caught. Finally, report will be given and stored in databases.

5. Experiments

To verify the feasibility of the method, Firing Range[20], a special testing tool Web application, will be used for testing the function of each module of the prototype detection framework. Firing Range covers the classic Web application vulnerabilities, which are representative. Through the analysis of process and result, it can effectively prove the reliability of the detection framework.

We have covered static analyzers and dynamic analyzers, too. For example, Ra.2, JSPwn[21], and AWVS[22]. Moreover, the comparison will be given in rest part.

5.1. Experiment processs

We used 2 PCs during experiment. One crawled URL to add tasks, another detected the Web application to get results. We prepared environment for compiling PhantomJS, and Google App Engine for running Firing Range. Results database was deployed on the second server.

For different software, we also wrote some scripts to make the automated detection process smooth.

As shown in Table 7, we use different types of DOM-XSS vulnerabilities raised by Firing Range to simulate the actual situation. As shown in Listing 1, the sample code shows: first, it truncates part of the URL address string to get the anchor; then, the createElement function is used to add a div in the current page in the document; finally, the truncated string is to add div with innerHTML. According to our method, one of attack vectors, $http : //192.168.211.1/address/location.hash/innerHtml.html\# < svg/onload = alert() >$, is verified automatically. If necessary, it is also possible to carry out manual verification conveniently.

Table 7: Types of DOM-XSS vulnerabilities

Type	Count
LocationHash	14
Location	8
Cookies	4
Referrer	3
WindowName	3
LocalStorage	10
PostMessage	3
EventTriggering	3
Others	7

Listing 1: Code of sample vulnerability

```
<html>
  <head></head>
    <body>
      <script>
        var payload = window.location.hash.substr(1);
        var div = document.createElement('div');
        div.id = 'divEl';
        document.documentElement.appendChild(div);
        var divEl = document.getElementById('divEl');
        divEl.innerHTML = payload;
    </script>
  </body>
</html>
```

5.2. Results and Comparison

We use Ra.2, JSPwn and Acunetix Web Vulnerability Scanner 10.0 to detect the Firing Range. Ra.2 is a black box DOM-XSS detection plugin based on Firefox, JSPwn is an open source project of static analysis and AWVS 10.0 with DeepScan technology to detect DOM-XSS is a famous international Web application security testing software.

Ra.2 relies relies heavily on fuzz datasets. So it is expensive to construct a fuzzing dataset and Ra.2 suffers from high false negative rates, when the logic is complicated. Furthermore, Ra.2 cannot detect some source APIs such as document.referrer, window.name, since Ra.2 is a plugin based on firefox.

JSPwn relies on a lot of parameter settings for detection rules, and additional manual testings from the drawbacks of static analysis. Moreover, JSPwn will be affected by code ambiguity and code obfuscation.

For comparison, we use the results from dynamic analysis method of this paper and AWVS 10.0.

Table 8: Detection results by dynamic analysis

Type	This paper	AWVS 10.0
LocationHash	4	8
Location	3	0
Cookies	2	0
Referrer	2	3
WindowName	0	3
LocalStorage	0	0
PostMessage	0	0
EventTriggering	0	0
Others	5	1
SUM	17	16

As shown in Table 8, detection framework in this paper detected 17 possible vulnerabilities, while AWVS 10.0 detected 16. The experimental results are satisfactory, as our framework detects more 1.8% vulnerabilities than AWVS 10.0. Without sacrificing the accuracy, our method is easy to be implemented. Most vulnerabilities with $LocationHash$, all vulnerabilities with $Referrer$ and $WindowName$ are detected by AWVS 10.0. Since it is a well-structured long-term updated software and its scanning of specific vulnerabilities is very comprehensive. However, uncommon vulnerabilities (like

other type) are hardly detected by it, while our framework is less affected. According to the results, we found that all vulnerabilities with APIs covered by our methods were detected. As our framework covers more APIs, it will do better. In addition, that it modifies APIs costs less than others modify the logical expression.

Table 9: Automatic verification of vulnerabilities

Case	This paper	AWVS 10.0
detect	17	16
verify	5	0

Another comparison, as shown in Table 9, shows that 5 of the 17 (9.1%) vulnerabilities were verified automatically by our method, while AWVS 10.0 verified 0. That is to say, all need to comfirm by manual for AWVS 10.0. AWVS 10.0 found a lot of potential vulnerabilities by tracking the source, and only gave the position of the input and output without string variable information. So still much work needs to be done by manual.

There are also some detected vulnerabilities can not be confirm with method in this paper. They come from document.cookie, document.referrer, window.name, etc., which need two input phases. However, the detection framework gave the trace. The experiment results show that the method can effectively detect DOM-XSS vulnerabilities with lower rate of false positive and false negative.

6. Conclusion and Future Work

In this paper, we have proposed a dynamic detection framework called TT-XSS of DOM-XSS based on taint traking. In addition, we implement the prototype detection framework. During the detection process, our framework analyzes pages which maybe cause DOM-XSS and obtains taint traces. According to these traces, our method can generate attack vectors automatically, which is quite significant for developers to detect and repair vulnerabilities effectively. Experimental results show the effectiveness and the superiority of our framework over other representative work, as our framework detects more 1.8% vulnerabilities and verifies 9.1% than AWVS 10.0. In a word, our framework can find more comprehensive vulnerabilities and more abundant vulnerabilities information.

However, there are some limits in our framework. (1) In verification module of TT-XSS, we regularly select attack vectors to make fuzzing verifying, which can not handle two-order inputs. Therefore, we will modify the verification module for variety of inputs in our future work. (2) When payloads or paddings are complex, it will spend a lot of time to construct attack vectors. Therefore, we will use heuristic search or other algorithms to speed up the build process.

7. Acknowledgment

This work has partially been sponsored by the National Science Foundation of China (No. 61572355, 61572349) and 985 funds of Tianjin University, Tianjin Research Program of Application Foundation and Advanced Technology under grant No. 15JCYBJC15700 and No. 14JCTPJC00517.

References

[1] Y. Ren, J. Shen, J. Wang, J. Han, S. Lee, Mutual verifiable provable data auditing in public cloud storage, Journal of Internet Technology 16 (2) (2015) 317–323.

[2] D. Wichers, The 2017 owasp top 10, https://raw.githubusercontent.com/OWASP/Top10/master/2017/OWASP Top 10 - 2017 RC1-English.pdf, 2017.

[3] A. Hellal, L. B. Romdhane, Minimal contrast frequent pattern mining for malware detection, Computers & Security 62 (2016) 19–32.

[4] J. Newsome, D. Song, Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software, in: Network and Distributed System Security Symposium, 2005, pp. 720–724.

[5] D. Mitropoulos, P. Louridas, M. Polychronakis, A. D. Keromytis, Defending against web application attacks: Approaches, challenges and implications, IEEE Transactions on Dependable and Secure Computing 99 (2017) 1–1.

[6] S. Sivakorn, I. Polakis, A. D. Keromytis, The cracked cookie jar: Http cookie hijacking and the exposure of private information, in: Security and Privacy (SP), 2016 IEEE Symposium on, IEEE, 2016, pp. 724–742.

[7] K. Z. Snow, R. Rogowski, J. Werner, H. Koo, F. Monrose, M. Polychronakis, Return to the zombie gadgets: Undermining destructive code reads via code inference attacks, in: Security and Privacy (SP), 2016 IEEE Symposium on, IEEE, 2016, pp. 954–968.

[8] I. Hydara, A. B. M. Sultan, H. Zulzalil, N. Admodisastro, Current state of research on cross-site scripting (xss) c a systematic literature review, Information & Software Technology 58 (2014) 170–186.

[9] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krgel, G. Vigna, Cross site scripting prevention with dynamic data tainting and static analysis., in: Network and Distributed System Security Symposium, NDSS 2007, San Diego, California, Usa, February - March, 2007.

[10] Google, Ra.2, Google, https://code.google.com/archive/p/ra2-dom-xss-scanner/issues (2012).

[11] P. Saxena, S. Hanna, P. Poosankam, 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, February - March, 2010.

[12] M. Sutton, A. Greene, P. Amini, Fuzzing: Brute force vulnerability discovery, 2007.

[13] C.Criscione, Drinking the ocean-finding xss at google scale, 2013.

[14] S. Lekies, B. Stock, M. Johns, 25 million flows later - large-scale detection of dom-based xss, in: ACM SIGSAC Conference on Computer & Communications Security, 2013, pp. 1193–1204.

[15] I. Parameshwaran, E. Budianto, S. Shinde, H. Dang, A. Sadhu, P. Saxena, Dexterjs: robust testing platform for dom-based xss vulnerabilities, in: Joint Meeting, 2015, pp. 946–949.

[16] I. Parameshwaran, E. Budianto, S. Shinde, H. Dang, A. Sadhu, P. Saxena, Auto-patching dom-based xss at scale, in: Joint Meeting, 2015, pp. 272–283.

[17] R. Gomez, DOMXSS Scanner, https://github.com/yaph/domxssscanner (2016).

[18] Y. Xie, A. Aiken, Static detection of security vulnerabilities in scripting languages, in: Conference on Usenix Security Symposium, 2006.

[19] Z. Fu, X. Wu, C. Guan, X. Sun, K. Ren, Toward efficient multi-keyword fuzzy search over encrypted outsourced data with accuracy improvement, IEEE Transactions on Information Forensics and Security 11 (12) (2016) 2706–2716.

[20] C. Criscione, firing-range, Google, https://github.com/google/firing-range (2016).

[21] D. Monteiro, JSPwn, Github, https://github.com/dvolvox/JSpwn (2015).

[22] Acunetix, AWVS, Acunetix, http://www.acunetix.com/company/contact/ (2016).

*Author Biography & Photograph

Ran Wang is a postgraduate student in the school of computer science and technology, Tianjin University now and graduated from Tianjin University with a bachelor's degree. His main research field is the network attack and defense.

Guangquan Xu is an associate professor of the school of computer science and technology, Tianjin University. He is also deputy director of the Institute of software and information security engineering, ACM member and senior member of China Computer Society.The main research directions of hime are the network and information security and service computing.

Xianjiao Zeng is a postgraduate student in the school of computer science and technology, Tianjin University now and graduated from Yanshan University with a bachelor's degree. Her main research field is android security.

Xiaohong Li is a full tenured professor of the School of Computer Science and Technology, Tianjin University, Tianjin, China. Her current research interests include knowledge engineering, trusted computing, and security software engineering.

Zhiyong Feng is a full tenured professor and head of the School of Computer Software, Tianjin University, Tianjin, China. His current research interests include knowledge engineering, service computing, and security software engineering.

1) A dynamic detection framework for DOM-XSS by taint tracking is proposed.

2) New data types and methods used in tainting process are advanced.

3) Attack vectors are derived to verify the vulnerability automatically.