I.J. Engineering and Manufacturing, **2012, 5, 52-61**
Published Online October 2012 in MECS (http://www.mecs-press.net)
DOI: 10.5815/ijem.2012.05.08

Available online at http://www.mecs-press.net/ijem

Identifying Cross-Site Scripting Attacks Based on URL Analysis

Zhi'hua Tang [a], Ning Zheng [a], Ming Xu [a]

[a] *Institute of Computer Application Technology, Hangzhou Dianzi University, Hangzhou, China, 310018*

Abstract

Cross-site scripting (XSS) is one of the major threats to the security of web applications. Many techniques have been taken to prevent XSS. This paper presents an approach to identify Cross-Site Scripting attacks based on URL analysis. The fundamental assumption of our method is that the URL contains a part that can produce a valid JavaScript syntax tree. First, we extract the parameters of the URL to produce a valid JavaScript syntax tree and weight its parsing depth. If its depth exceeds a user-defined threshold, the URL is considered suspicious. Second, to the exception URLs, a second level of defense is formed by analyzing its structure. The experimental results demonstrate that our approach can effectively distinguish most of the malicious URLs from the benign ones.

Index Terms: Cross-site scripting, depth level, structure

1. Introduction

According to the Open Web Application Security Project (OWASP) [1], cross-site scripting (XSS) is already one of the top two vulnerabilities. Cross site scripting attack is a class of web application vulnerabilities in which an attacker cause a victim's browser to execute JavaScript crafted by the attacker to gain elevated access privileges to sensitive page-content, session cookies, and a variety of other information. These attacks can steal confidentiality of sensitive data, undermine authorization schemes, defraud users and defame web sites. Even some XSS attacks can be self-propagating [2] and have the potential to rapidly victimize millions of people. For example, the input contains scripting commands like <script>…document.cookie…</script>. When such injected code is executed in the client browser, it might result in stealing cookies, defacing the document or unauthorized submission of forms. We refer to such JavaScript code as *unauthorized* code and to distinguish it from code that was *authorized*. There are basically two large categories of XSS attacks, a) reflected and b) stored. During a reflected XSS attack the injected code is placed in a URL, upon the user clicks on the malicious URL, the injected code executes. On the other hand, during a stored XSS attack, the adversary injects the malicious payload in some form of storages utilized by a

* Corresponding author.
E-mail address: tangzhihua1101@126.com

web application. In view of the mentioned above, it is an important approach to identify the URLs for preventing XSS attacks.

1.1. Non-persistent XSS

The non-persistent (reflected) XSS vulnerability results from the data commonly in HTTP query parameters or HTML form submissions which are provided by web client. The attacker uses social engineering to convince a victim to click on a disguised URL that contains malicious HTML/JavaScript code. The user's browser then displays HTML and executes JavaScript which was a part of the attacker crafted malicious URL. This can result in stealing of browser cookies and other sensitive user data. To prevent first-order XSS attacks applications need to reject or filter input values that may contain script code.

1.2. Persistent XSS

The *persistent (stored)* XSS vulnerability results from the application that store part of the attacker's input in a database, and then inserting it in an HTML page that is displayed to multiple victim users. It is harder to prevent *stored* XSS than *reflected* XSS for two reasons, a) social engineering is not required, the attacker can directly supply the malicious input with tricking users into clicking on a URL, b) a single malicious script once planted into a database can execute on the browsers of many victim users later. And applications need to use variety techniques to reject or sanitize input values that may contain script code which are used in database commands.

Due to the prevalence of XSS attacks and current trend in web applications, there exists a strong need for preventing these attacks. Fixing XSS vulnerabilities in a large web site is a very challenging task if it is not impossible. In this paper, we make the following contributions:

- Presenting a system that can actively identify suspicious URLs in the network.
- Implementing a solution that can automatically learn the depth of attack vectors' syntax tree from practical data analysis, and modeling a parsing depth model of attack vectors.
- Giving a second level of defenses to the URLs which mismatch or miss the parsing depth model.

2. Related work

XSS defenses techniques can be largely classified into three categories: server side defenses, client side defenses and client-server cooperative defenses. This section describes current XSS defenses methods and their weakness.

2.1. Purely server side defenses

Server-side validation of untrusted content has been the most commonly adopted defense in practice, and a majority of defense techniques [3-4] were proposed in looking for scripting commands or mate-characters in untrusted input, Employing filter provides a first layer of defense against XSS attacks, but this consistency has been missing and infeasible because there are many scenarios where filtering is difficult to deal with. Otherwise a common problem with purely server-side mitigation strategies is the assumption that parsing on the client browser is consistent with the process of server-side. This is troublesome due to the diversity of popular web browsers which contain subtle parsing quirks that allow scripts to evade detection.

2.2. Purely client side defenses

Client-side XSS defenses focus on ensuring confidentiality of sensitive data by analyzing the flow of data through the browser [5]. One main technique is preventing unauthorized script to execute [6-7]. First, such solutions can not distinguish untrusted data generated by the server from user-generated data, leading to high false negatives and false positives. Second, they are largely targeted towards attacks that steal sensitive information. Noncespaces [6] and Document Structure Integrity [7] are all related in the goal of preserving the integrity of document structure on the browser to defense XSS attacks. They are all designed to prevent untrusted content to be displayed on existing browsers but without any assurance about protecting from XSS attacks on these browsers. In general, only on the client-side without server-side specification, either raise false positives or tend to be too specific to certain attack vectors.

2.3. Client-server cooperative defenses

Client-server cooperative defenses [8-10] for XSS defense have emerged to deal with the inefficiencies of purely client and server based mechanisms. BEEP [8] proposed white list legitimate scripts policies in web application and then enforced by the browser to protect against the injection codes. But this approach has problem of scalability from the web application's point of view; every client user needs to have a copy of this specialized browser that can understand this non-standard communication. BLUEPRINT [10] reduced the web application's dependency on unreliable browser parsers and provides strong assurance.

3. Our approach

The goal of our approach is to learn the depth of attack vectors' syntax tree in the URLs and generate a model to prevent XSS attacks. The architecture of our system is shown in Fig.1. The Attack vector catcher extracts the URLs' parameters which then were sent to XSS vector depth parsing. A depth threshold was produced by analyzing the parameters which contain XSS attack vector. Depth learning of XSS vector' syntax tree is the kernel part and an algorithm is proposed to training the threshold depth. Structure analysis provides a second level of defense of XSS attacks.

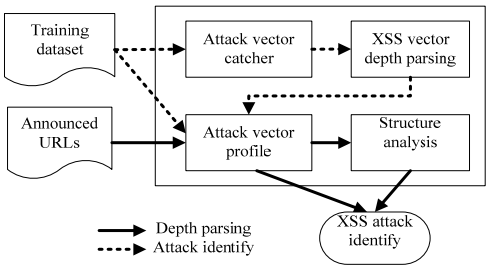

Fig.1.1 Architecture of our system

3.1. XSS Attack Vector Catcher

The attack vector catcher decodes the obtained XSS attack URLs and then attempts to extract the elements of attack vectors from the known XSS URLs. It involves two phases: decoding phase and extractor phase.

1) XSS Attack URLs Decoding

An existing exploit can be obfuscated to avoid the detection, such obfuscation can be achieved by encoding it in various ways such as UTF-8, HEX, special entities and so on. For example, “"” can be encoded as “&quot”, “&#34”and “&#x22” respectively by corresponding encoding approach. So it is necessary to decode the obfuscated XSS URLs first. In the decoding phase, it needs to examine the obfuscated source code of the URLs and use some regular expression to decode. Thus, the attack vector catcher is able to identify suspicious parts from XSS attack URL and extract the elements of attack vectors. In short, if the XSS Attack Vector Catcher processes these obfuscated XSS URLs directly, not only will it fail to identify the XSS attack, but also the token extractor will lose a lot of information about the attack vector. To obtain the original information of the attack vector, the Attack Vector catcher handles the URL encoding.

2) XSS Attack Token Extractor

After decoding phase, the XSS Attack Vector Catcher examines the entire XSS URL to identify where the attack vectors exist. According to URL syntax, the part of the URL follows the “?” character, named as the query string which isolated from the rest of the URL. The query string contains all parameters which take part in an HTTP GET/POST request. All parameters are separated using the “&” character and each parameter is in the form of “key = value”. The term “value” here is the value of a parameter in a URL. The delimiters “=” are used to separate the parameters from their values. The XSS Attack Vector catcher complies with the delimiters to obtain one or more values in a XSS URL and estimates the most possible value where a XSS attack exists by learning level features. At the same time, we limit minimum length of the malicious parameters, because only a certain length of code can exploit a script.

3.2. XSS Vector Depth Learning

After the process of the XSS Attack Vector Catcher, We extract all the parameters in the URLs. Our system refers to the JavaScript source code engine of Mozilla Spider Monkey [11] to generate parse tree. We weight the depth of the syntax tree depends on the tokens that the parameter composed. A URL is considered suspicious if it includes some part that can produces a JavaScript parse tree within a certain depth. We propose our algorithm of calculating the depth of the syntax tree in Fig.2.

The motivation of our approach is that, first benign URL usually did not compose the especially tokens which often exist in rich-content HTML, second the actual JavaScript code has a high probability to include certain tokens. For example, the"<" character needs to be present in hyperlinks and text formatting, and the "character needs to be present in generic text content. Someone can input a string prefixed double quote like ""<script>alert()</script>", through variable "alert" to trigger the vulnerability. However, the following attack “<script>alert()</script>”may not work,. It is clear that double quote is the critical character to introduce the attack body, and the attack vector is double quote in here. The string “"><script>alert()</script>"” is even better, as it is seamlessly embedded in the page. A typical example of weighting the parsing depth is depicted in Fig. 3. It is a URL selected from [13] that contains XSS attack codes.

3.3. Attack Vector Profile

Once the level learning model is built, an attack vector profile will be published. We introduce some formal definitions for the threshold.

```
1 url← File.Readline(String);
2 url1← decoding(url);
3 param ← getparameters(url1);
4 while(param){  //more than one parameter
5   if(param.length >K){
6    getDepthlevel() {
7       function1(){
8              find "((<)[^\n]+(>)).*(<)((<)//[^\n]+(>))");
9              depth++;
10               text ← ReplaceFunction(param);
11                    }
12   function2(text){
13               find("", "" ","">");
14               depth ++;  (each)
15               text2← ReplaceFunction();
16                    }
17   function3(tex2){
18               find("alert\\((.+?)\\)"));
19               depth++;
20               text3← ReplaceFunction();
21                    }
22   function4(tex3){
23            find((<)[^\n]+(>))| "(()[^\n]+())");
24            depth++; (each)
25                    }
26            array[]=depth;
27             }
28       Depth←Maxarray[];
29         if(Depth > threshold)
30            It is a suspicious URL!
31        else
32             Structual (param);
33       }
```

Fig.2. An example of operating on weighting the parsing depth

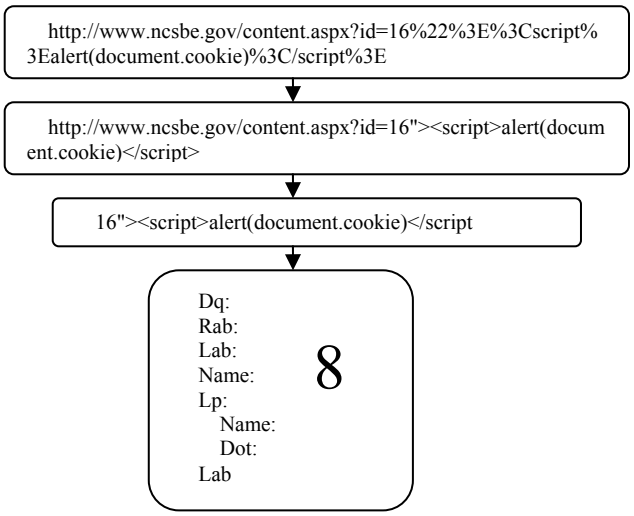

Fig.3. An example of operating on weighting the parsing depth.

- **Definition 1**. *U* denotes the set of URLs, *M* denotes the set of suspicious parts of a certain URL *u*, $M \in u$, $u \in U$.
- **Definition 2**. $S(t_i)$ denotes the parsing depth of M_i which is proposed in my algorithm, and $M_i \in M$, $i>1$.
- **Definition 3**. $S(t)$ stands for the maximum of the $S(t_i)$,which is means the parsing depth of the URL *u*.
- **Definition 4**. If $u \in U$, such that $S(t)$ exceeds a customized threshold *T*, we call u is suspicious of XSS attack.

The threshold *T* evaluated from our experimentation by analyzing the depth on 250 known attack URLs. The parsing depths are represented in Fig. 5. According to our definition, the URLs whose depth level exceeds the threshold will be considered as suspicious.

3.4. Attack vector structure detection

There are a few cases that mismatch and bypass our system. First, the URL don't have the "?" to split the domain from the search parameter. Second, the URL don't contain XSS attack vector directly but redirect to a hyperlink where the attacker want. Third, a few URLs might escape our threshold but are malicious. Considering those reasons, we design this component to provide a second level of prevention. First, many pairs of HTML tags, popular JavaScript constructs like document.write(), String.fromCharcode(), and event handlers like error(), onload(), were pre-stored in a file. Second, in order to find out whether there are any substrings of the URL existing in the file, a progress of comparing the file with URL is invited. If there are some substrings of the URL exist in the file also, the URL is considered as suspicious. For example, URL"http://www.mil.ee/?sisu="><iframesrc=http://xssed.com>" in http://www.xssed.com/mirror/66421/. Its depth is less than the defined threshold, so it can bypass our threshold method, but it will be detected by the structural analysis approach because it has HTML tags "iframe" and "src".

4. Challenges

We now proceed and present various challenges that we have to deal with. We present an example taken from xssed.com for each particular case.

4.1. Existing obfuscation methods:

An existing exploit can be obfuscated to avoid the detection, HTML entity codes, URL encoding, Base64 and double encoding are common obfuscation techniques used in practice. Encoded URLs not only lead human hard to read and write but also inflate its own size. Fig.4 shows an URL that is encoded twice by HEX encoding.

[http://xssed.com/mirror/274--- founder: ruzgar_18]
http://fr.netlog.com/go/search/go/search/view=people&q=%2522%253e%253c%2573%2563%2572%2569%2570%2574%253e%2561%256c%2565%2572%2574%2528%2522%2561%2563%2569%256b%2522%2529%253c%252f%2573%2563%2572%2569%2570%2574%253e&submit=Ara&v=g

Fig.4. An example of an obfuscated URL.

For this instance, the decode step is a necessary and we introduce some steps to pretreatment before extractor the attack vectors. We deal with the decoding process automatically. After our decoding process, the true appearance of the URL in the Fig.4 is *http://fr.netlog.com/go/search/go/search/view =people&q="><script>alert("acik")</script>&submit=Ara&v=g*. Some cases are impossible to handle since the URLs was encoded in a scheme only know by themselves.

4.2. Javascript relaxed syntax

In the paper we have mentioned that there are many attack vectors to be embedded scripts contents in URLs， the XSS exploit code can be partial or mixed up with other irrelevant text. During the statistical of the test data we found there are the following main classifications: (1) Use script code <script>….</script> directly. (2) Use JavaScript: pseudo-URL. (3) event handlers such as error(), onload(); And there are at least 94 event handers reported [12] (4) Tag attributes such as src, iframe and other content-rich html.(5) There are many URLs use HTTP POST instead of HTTP GET for attacking the web application. By this way, it is easier to draw into false positives.

4.3. Weighted the parse node level

Overview all the URLs were collected from XSSed.com, the repository classifies all attacks into two categories: Direct and Redirect XSS. We declare that it is easier to draw into false positives in the paper. For example, parse nodes such as "." (DOT) and """ (Dq), can be repeated several times in an expression and thus result to parsing nodes that contribute to the final syntax tree's depth. These tokens occur frequently in URLs, without being part of a JavaScript code. Although, there are tokens that are more likely to be part of valid JavaScript code, such as the LP token, which denotes a left parentheses occurrence. These tokens occur less frequently in URLs and much more frequently in JavaScript code. Which otherwise would be introduce false positives.

5. Experiment and Results

In this section, we evaluated the proposed approach that is designed for learning attack vector's levels to identify the vulnerabilities in URLs. The architecture of our system is shown in Fig.1. The training dataset are 250 known XSS URLs selected from XSSed.com. The accounted URLs are about 13,000 URLs also collected from [13] and other 800 URLs collected from social websites.

5.1. Statistics of the collected URLs

XSSed.com is a public XSS repository, and it document recently successful XSS attacks on major blog and social networking sites. A statistics of these collected URLs which were collected from July 2008 to October 2010 is shown in Fig.5.

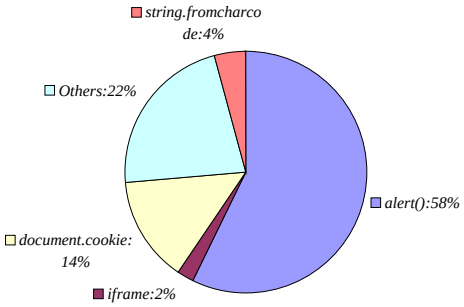

Fig.5. The distribution of announced XSS URLs.

5.2. Depth level threshold analysis

In theory, the malicious script codes usually can produce a valid JavaScript syntax tree. According to the algorithm which is proposed in previous section, we analyze on the 250 known URLs. And we exhibit the level of each URL in Fig.6. As we can see, 98% of the collected URLs' depth exceeds 6, so we set it as the threshold. In our experiment, some URL's depths even exceed 57 since it contains rich html- content.

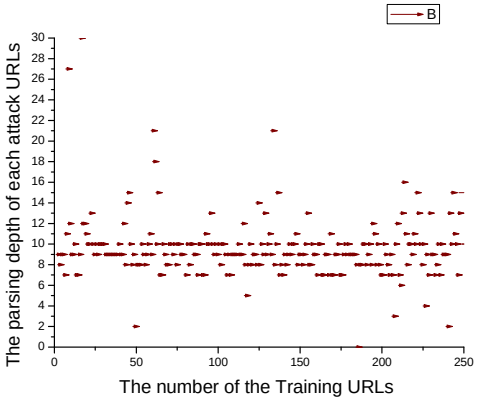

Fig.6. The distribution of parsing depth.

5.3. Results

Because the target programs have been examined by previous researches, we regarded to the effectiveness of our system for assisting vulnerability scanner in attack string generation by two metrics, that is, the false positive (FP) and the recall rate (Recall). We give the definitions as follows. The FP rate denotes the ratio of the number of false testing attacks to the number of successful testing attacks. Recall rate denotes the ratio of the number of success identifying attacks to the numbers of total vulnerabilities. How is the effectiveness of our system to find out the existed vulnerabilities showing in TABLE I.

Talbe.1. Result of the experiment

	Total URLs	Identified attacks	Recall %	FP %
Only depth parsing	13250+800	9845	74.3	2.3
Add structural analysis	13250+800	9845+848	80.7	2.1

There are some cases that our approach can't to deal with although the second level defense was introduced. For example, URL was obfuscated with unrecognizable techniques .Others like *http://bharat.gov.in/outerwin.php?id=http://pakbugs.com* in *http://www.xssed.com/mirror/66483/*, which just contains a link to redirect but without any attack vectors. Yet there are only a few cases.

6. Conclusion

In this paper, we describe the current XSS defenses and discuss some of their weakness, and then we present a depth level mechanism for identifying suspicious URLs that contain XSS attack exploits. We try to identify all parts contained in a URL that produce a valid JavaScript parse tree. If a fragment produces a syntax tree of a certain depth, then the URL is considered suspicious. To the exception URLs, we propose a second line of identifying and detect by analyzing its structure. Throughout this paper, we analyze main technical challenges and the implementation. We perform an evaluation using 13,000 URLs that contain XSS exploits, collected from XSSed.com, and 800 benign URLs collected from social platform websites. Although some cases will miss our approach, the results suggest that our approach has less than 2.1% false positives and more than 80.7% recall ratio.

Acknowledgements

This paper is supported by NSFC (No. 61070212, No.61003195), Natural Science Foundation of Zhejiang Province, China under Grant No. Y1090114, the State Key Program of Major Science and Technology (Priority Topics) of Zhejiang Province, China under Grant No 2010C11050.

References

[1] Van der Geer J, Hanraads JAJ, Lupton RA. The art of writing a scientific article. J Sci Commun 2000;163:51-9.
[2] Strunk Jr W, White EB. The elements of style. 3rd ed. New York: Macmillan; 1979.
[1] OWASP (2010). OWASP Top 10 Project , Available at http://www.owasp.org/index.php.
[2] S. Kamkar, "I'm popular," 2005, description and technical explanation of the JS.Spacehero (a.k.a. "Samy") MySpaceworm. [Online]. Available: http://namb.la/popular.
[3] P. Bisht and V. N. Venkatakrishnan. XSS-GUARD: precise dynamic prevention of cross-site scripting attacks. In Detection of Intrusions and Malware, and Vulnerability Assessment,2008.
[4] EnginKirda, Christopher Kruegel, Giovanni Vigna,and Nenad Jovanovic. Noxes: A client-side solution formitigating cross-site scripting attacks. In Proceedings of the 21st ACM Symposium on Applied Computing (SAC),Security Track, 2006.
[5] S. Nanda, L.-C. Lam , T. Chiueh. Dynamic multiprocess information flow tracking for webapplication security. In Proceedings of the 8th ACM/IFIP/USENIX international conference on Middleware, 2007.
[6] M. Van Gundy and H. Chen, "Noncespaces: Using randomization to enforce information flow tracking

and thwart crosssite scripting attacks," in 16th Annual Network & Distributed System Security Symposium, San Diego, CA, USA, Feb. 2009.
[7] P. Saxena, D. Song, and Y. Nadji, "Document structure integrity:A robust basis for cross-site scripting defense," in 16th Annual Network & Distributed System Security Symposium,San Diego, CA, USA, Feb. 2009.
[8] T. Jim, N. Swamy, and M. Hicks. Beep: Browser-enforced embedded policies. 16th International World World Web Conference, 2007.
[9] D. Bates, A. Barth, and C. Jackson. Regular Expressions Considered Harmful in Client-Side XSS Filters. In Proceedings of the 19th international conference on World Wide Web (WWW). ACM New York, NY, USA, 2010.
[10] M. Ter Louw and V. N. Venkatakrishnan. BluePrint: RobustPrevention of Cross-site Scripting Attacks for ExistinBrowsers. In Proceedings of the IEEE Symposium on Securityand Privacy, 2009.
[11] SpiderMonkey Engine.http://www.mozilla.org/js/spidermonkey.
[12] XSS (Cross Site Scripting) Cheat Sheet. Esp: for filter evasion. http://ha.ckers.org/xss.html.
[13] K. Fernandez and D. Pagkalos. XSSed.com. XSS(Cross-Site Scripting) information and vulnerablewebsites archive. http://www.xssed.com.