XSS Vulnerability Detection Technology Based on EBNF and Two-stage Crawling Strategy (Postprint)

Authors: Huang Wenfeng, Li Xiaowei, Huo Zhanqiang

Date: 2018-05-24T00:00:00+00:00

Abstract

Cross-Site Scripting (XSS) attacks represent one of the most significant threats to Internet security today. To address issues in traditional penetration testing-based vulnerability detection methods, such as low-complexity attack vectors that are easily filtered and cumbersome overall detection workflows, this paper proposes an automatic attack vector generation method based on Extended Backus-Naur Form (EBNF) and a secondary crawling strategy for XSS vulnerabilities. By defining EBNF rules to generate a rule parse tree, high-complexity attack vectors are obtained through hierarchical traversal. During the initial page crawling phase, input point information is embedded into attack vectors for injection requests; subsequently, secondary crawling is performed by requesting legitimate parameters to obtain response pages. Finally, a prototype system was designed and implemented, and vulnerability detection was conducted across two platforms. Comparative experiments demonstrate that the system simplifies the detection process, improves vulnerability detection counts to a certain extent, and reduces the false positive rate.

Full Text

Preamble

Title: XSS Vulnerability Detection Technology Based on EBNF and Twice-Crawling Strategy

Authors: Huang Wenfeng[1], Li Xiaowei[2], Huo Zhanqiang[2]†
[1]Henan Provincial Institute of Scientific & Technical Information, Zhengzhou 450003, China
[2]College of Computer Science & Technology, Henan Polytechnic University, Jiaozuo, Henan 454000, China

Abstract: Cross-site scripting (XSS) attacks have been one of the biggest threats to Internet security. Traditional vulnerability detection methods based on penetration testing technology suffer from issues such as low-complexity attack vectors that are easily filtered and cumbersome overall detection processes. This paper proposes an automatic attack vector generation method based on Extended Backus-Naur Form (EBNF) and an XSS vulnerability twice-crawling strategy. By defining EBNF rules, the method generates a rule-parsing tree and then traverses it hierarchically to obtain high-complexity attack vectors. During the first page crawling, input point information is embedded into attack vectors before requesting injection, followed by a second crawling that requests legitimate parameters to obtain the returned page. Finally, this paper designs and implements a prototype system and uses two platforms for vulnerability detection. Comparative experiments demonstrate that the system has a simple detection process, improves the number of detected vulnerabilities to a certain extent, and reduces the false positive rate.

Keywords: XSS; EBNF; attack vector; penetration testing

0 Introduction

Since the birth of the Internet, cybersecurity issues have always existed. With the widespread adoption of Web applications, these issues have received increasing attention from security researchers. The Open Web Application Security Project (OWASP) 2017 Top 10 list of the most critical Web application security risks ranks cross-site scripting (XSS) at seventh place [1]. WhiteHat Security' s 2017 Application Security Statistics Report [2] shows that XSS accounts for 33% of all security risks, second only to information leakage risks at 37%. As the second most common security issue, nearly two-thirds of Web applications face XSS risks, which can cause significant damage once exploited. This demonstrates that XSS security threats to Internet security cannot be ignored.

Currently, domestic and international researchers have developed XSS vulnerability detection tools that fall into two main categories: black-box testing tools [3] and white-box testing tools [4]. Black-box testing tools primarily use dynamic analysis techniques [5], while white-box testing tools mainly use static analysis techniques [6], with some researchers employing a combination of both [7]. Dynamic analysis techniques detect vulnerabilities during Web application execution, commonly using methods such as penetration analysis, dynamic taint analysis, traffic analysis, and monitoring/filtering. Static analysis techniques detect vulnerabilities by reviewing Web application source code or bytecode, employing methods such as static taint analysis, symbolic execution, and string analysis. To avoid XSS vulnerabilities during Web development, researchers have proposed solutions such as secure programming and modeling.

Li Wei et al. [8] proposed a detection method specifically for stored XSS vulnerabilities, using BNF to automatically generate initial attack vectors and then performing mutation operations. They used a focused Web crawler to crawl

pages, submitted probe vectors to form forms, performed network-wide scans to find output points, and finally injected attack vectors based on matched input-output points for vulnerability detection. Although this method reduces the number of tests during vulnerability detection, it makes the pre-detection preparation process overly cumbersome. After automatically generating initial attack vectors, mutation operations are still required, and using probe vectors to find output points for each form requires network-wide scanning. Gu Mingchang et al. [9] improved penetration testing methods by proposing an XSS attack vector automatic generation method based on symbolic sets and using a decision tree classification algorithm to categorize attack vectors for direct use by category during the testing phase. This method also uses probe vectors before penetration testing to eliminate some URLs without vulnerabilities, reducing unnecessary tests and improving detection efficiency. However, probe vectors complicate the crawling process, and the generated attack vectors are relatively simple and easily filtered by the server side. Wu Zijing et al. [10] proposed an anti-filter rule set to transform XSS attack vectors to bypass server-side filtering mechanisms for different types of malicious code and implemented automatic injection of XSS attack vectors through an automatic crawler, representing another improvement to penetration testing methods. However, this method only studied how to make attack vectors bypass server-side filtering mechanisms, and during vulnerability detection, it only injected attack vectors once through the crawler before starting detection, which can easily cause a large number of stored vulnerabilities to be missed.

This paper proposes a new automatic attack vector generation method and designs a vulnerability detection system based on penetration testing technology. The system uses EBNF to define attack vector generation rules, increasing the randomness of XSS attack vector generation, improving attack vector complexity to bypass servers, and embedding input point information in malicious script fragments to record input points, ultimately obtaining mutated attack vectors directly. As a black-box testing tool, the system performs two page crawling operations during the target system' s operation. During the first crawling, carefully designed attack vectors are injected, followed by a strategic second crawling based on the first crawling results. Finally, regular matching methods are used to analyze and detect vulnerabilities in the saved pages, directly locating vulnerabilities based on input point information in malicious script fragments.

When designing and implementing the XSS vulnerability detection system proposed in this paper, we did not consider the impact of WAF firewalls and anti-crawling measures on detection results for two main reasons: a) The system' s target users are website developers and maintainers, whose goal is to detect as many XSS vulnerabilities as possible in their own Web systems for repair. If a Web system has WAF firewalls or anti-crawling measures, they need to be temporarily disabled; b) This is not a professional crawler system, with the focus on designing high-complexity attack vectors to bypass server-side filtering mechanisms. For page crawling, we improved existing open-source crawler tools and used a twice-crawling strategy to reduce the false negative rate for stored

vulnerabilities.

1 Related Knowledge

1.1 XSS Vulnerabilities

To date, researchers have classified XSS vulnerabilities into three main types [11-13]: reflected XSS vulnerabilities, stored XSS vulnerabilities, and DOM-based XSS vulnerabilities.

Reflected XSS and stored XSS vulnerabilities are server-side vulnerabilities. Their formation principle is that attackers submit malicious code from the browser side to the server side. When the server fails to effectively filter and validate this code, the malicious code appears in the returned page. When users access this page, the malicious code loads, creating a vulnerability. DOM-based XSS is a variant of reflected XSS and stored XSS. It differs significantly from stored XSS but has subtle differences from traditional reflected XSS, which lead to completely different attack characteristics. In DOM-based XSS, the server does not directly return malicious scripts as part of the page. Instead, when the browser loads the page, legitimate scripts directly output user input data as HTML content to the page, so after the legitimate script executes, malicious scripts are also inserted into the page.

1.2 Penetration Testing

Penetration testing technology is a technique that simulates hacker vulnerability discovery techniques and attack methods as completely as possible to conduct in-depth testing of target network security and discover any weaknesses, technical defects, or vulnerabilities. There are many types of penetration testing methods, but the two most common and widely accepted methods in the industry are black-box testing and white-box testing.

Black-box testing: Also known as external testing, testers are completely unaware of the system. They use information collection tools to obtain information from DNS, Web, Email, and various publicly accessible servers, simulating real hacker techniques to conduct organized and step-by-step penetration and intrusion of the target network to discover known or unknown security vulnerabilities and perform security assessments.

White-box testing: Also known as internal testing, testers obtain all internal and underlying information about the target network in advance and can view and evaluate the most serious security vulnerabilities in the target network at minimal cost. White-box testing can eliminate almost all security risks existing in the target network's internal code and facilities, making it more robust against external malicious attacks.

1.3 EBNF

EBNF (Extended Backus-Naur Form) is a metasyntax notation primarily used to formally define the syntax of computer programming languages and many other languages. It is an extension of the basic Backus-Naur Form (BNF). EBNF defines production rules that assign sequences of symbols to non-terminals, such as `letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g';`. This production rule defines the non-terminal `letter` on the left side of the assignment. The vertical bar indicates alternatives, terminals are enclosed in quotes, and a semicolon serves as the termination character. Therefore, `letter` is an English letter from a or b through g.

2 Attack Vector Design Based on EBNF

This chapter first introduces how to design attack vector seeds, then describes how to define EBNF attack vector generation rules based on these seeds, and finally generates mutated attack vectors according to the rules.

2.1 Attack Vector Design

During XSS vulnerability detection, the effectiveness of designed attack vectors is key to comprehensively detecting XSS vulnerabilities. Attack vectors have many types and conform to certain composition rules: a) they contain executable JavaScript scripts; b) they conform to HTML syntax and can be embedded in HTML pages; c) they can be triggered for execution. Typically, one type of attack vector is insufficient to detect all possible vulnerabilities, so multiple types must be used for repeated detection.

This paper primarily uses the following types of attack vectors to detect XSS vulnerabilities:

a) Direct use of scripts as attack vectors: `<script>alert('xss')</script>`, `<script src="xss.js"></script>`

b) Embedding scripts as HTML element attribute values: `<img src="javascript:alert('xss')"/>`

c) Embedding scripts as HTML element trigger events: `<body onload="alert('xss')">`

d) Embedding scripts as CSS style attribute URL values: `<body style="background-image: url('javascript:alert("xss")');">`, `<style>@import url("javascript:alert('xss')");</style>`

If the input point is within an HTML tag attribute value, the tag must first be closed by adding start strings like `">` or `">` to the attack vector. Since the JavaScript code portion of an attack vector can contain any valid JavaScript statement, this paper adds input point information to the code portion, which avoids recording input and output point information during detection and greatly simplifies the entire vulnerability detection process. For example, for the attack vector `<script>alert('xss')</script>`, this paper replaces the

`alert('xss')` portion with `alert(URL)`, where the URL content actually identifies the input point information.

2.2 EBNF Rule Definition

Since existing EBNF syntax cannot meet requirements, this paper adds several new syntax rules to the original EBNF:

Definition 1: Define new syntax `<...>`, where each group of elements must be connected by `|`, containing the concept of grouping and allowing multi-level nesting. If a rule contains two or more identical `<...>` groups connected by `,`, only one identical group can be selected. If it appears only once, it follows the same syntax rules as `(...)`.

Definition 2: Define new syntax `:`, where elements connected by `:` are ordered and contain the concept of `|`. If a rule contains two or more such connection strings connected by `,`, values are taken in corresponding order. If it appears only once, it follows the same syntax rules as `|`.

Based on these mutation processing methods, this paper defines different forms of EBNF rules, dividing them into basic character rules and basic statement rules. Basic character rules are shown in Table 2 .

Basic statements are built upon basic character rules, as shown in Table 3 .

2.3 Attack Vector Mutation

Based on basic character rules and basic statement rules, various types of attack vector generation rules can be defined. According to the attack vector types introduced above, several rules are defined as shown in Table 4 .

After parsing according to these rules, highly complex mutated attack vectors can be generated. Rule parsing trees are used for this purpose, with each rule corresponding to a rule parsing tree. After generating the parsing tree, mutated attack vectors can be generated through hierarchical traversal according to certain rules. For example, the basic character rule `a = 'a'|'A'|('&#','97'|'65'|('x'|'X','61'|'41'),';')|'%61'|'%41';` corresponds to the rule parsing tree shown in Figure 1 [Figure 1: see original paper].

Similarly, the basic statement rule `payload_rule0 = left_script_label,js_strings,right_script_label` corresponds to the rule parsing tree shown in Figure 2 [Figure 2: see original paper]. Each leaf node of this parsing tree is the root node of a parsing tree generated by basic character rules.

The mutated attack vectors generated by the parsing tree corresponding to each attack vector generation rule are shown in Table 5 .

3 System Design

3.1 Overall Framework

Based on the previous research and analysis of various XSS vulnerability principles, this paper designs a detection system for stored and reflected XSS vulnerabilities aimed at generating high-complexity attack vectors and simplifying the XSS vulnerability detection process. The system consists of three major modules, with the overall structure shown in Figure 3 [Figure 3: see original paper]. The attack vector generation module includes two sub-modules: rule parsing tree generation and mutated attack vector generation. The crawler module includes two sub-modules: page crawling and page parsing. The attack vector generation module is the foundational module of the detection system, providing services for the crawler module and vulnerability detection module, with the main function of generating mutated attack vectors. In this module, the rule parsing tree generation sub-module generates corresponding rule parsing trees based on defined EBNF rules, after which the mutated attack vector generation sub-module generates mutated attack vector strings under different rules. The crawler module is the core module of the detection system, providing services for the vulnerability detection module, with the main function of crawling and parsing all pages of the target system, with both page crawling processes performed in this module. The page crawling sub-module produces crawlers and crawls pages, while the page parsing sub-module parses crawled pages and feeds analysis results back to the crawler. The vulnerability detection module is the final module of the detection system, depending on services provided by preceding modules, and determines vulnerabilities by detecting the content of saved pages.

3.2 System Process

The detection method of this system is an XSS vulnerability detection method based on penetration testing technology. The entire detection process is shown in Figure 4 [Figure 4: see original paper]. The vulnerability detection process is divided into five sub-processes: attack vector generation, crawler initialization, first crawling and parsing, second crawling and parsing, and vulnerability detection. First, the EBNF-based attack vector generation method described above is used to generate different types of mutated attack vectors for subsequent operations. Then, crawler-related configurations are initialized, and carefully selected seed URLs are added to the to-be-crawled URL queue. After crawler initialization is completed, a vulnerability crawler performs the first page crawling and parsing process, saving some crawled pages. Next, a second page crawling and parsing process is executed to obtain pages not crawled during the first pass. Finally, vulnerability detection is performed on all crawled pages to determine whether XSS vulnerabilities exist. The specific execution process of each subprocess is detailed below.

3.2.1 Attack Vector Generation This sub-process implements the automatic generation of mutated attack vectors, producing high-complexity attack vectors that can effectively bypass server-side filtering mechanisms. The specific execution flow is shown in Figure 5 [Figure 5: see original paper]. First, configuration information for attack vector generation is initialized. Then, based on this configuration, the EBNF-formatted attack vector generation rule file is parsed to generate mutated attack vectors.

3.2.2 Crawler Initialization In this sub-process, to enable the crawler to reach deeper levels of the target system, login authentication must first be performed in the target system. After authentication, basic crawler configurations such as the number of crawlers and crawling depth are set. Finally, carefully selected seed URLs are added to the to-be-crawled URL queue. The detailed process is shown in Figure 6 [Figure 6: see original paper].

3.2.3 Crawling and Parsing When a crawler crawls pages in the target system for vulnerabilities, a specific situation arises: some URL requests have deletion functions that delete data stored in the server-side database. Due to page crawling order, the crawler may first inject attack vectors into the server-side database through input points of stored vulnerabilities, then crawl URLs with deletion functions before crawling the corresponding output point pages, causing previously injected attack vector information to be deleted from the database and affecting subsequent vulnerability detection processes, resulting in missed stored vulnerability reports.

Therefore, during page parsing, this system uses a combination of conventional parsing and semantic analysis to perform semantic judgment when extracting new URL requests, filtering URLs containing deletion semantics such as "del," "delete," "remove," and "rmv." Simultaneously, it extracts tag words corresponding to URLs and filters URLs containing words like "删除," "delete," "移除," "remove," and "删掉." For example, if an HTML page contains a hyperlink: `<a href="CompanyAction_delComByAdmin.do?id=00000001" onclick="javascript:if (!confirm('确认要删除?')) { window.event.returnValue = false;}"> 删除 </a>`, this URL' s tag word is "删除" and the URL contains the "del" string, so this URL is filtered and not crawled further. Since URL requests with deletion functions will not contain reflected or stored XSS vulnerabilities, whether they are crawled or not does not affect subsequent XSS vulnerability detection.

This system has two crawling and parsing processes, with the second crawling being much simpler than the first and performed based on the results of the first crawling. The two crawling processes target different vulnerability types. The first crawling primarily focuses on reflected vulnerabilities, possibly only capturing a small portion of stored vulnerabilities. The second crawling targets stored vulnerabilities and no longer performs reflected vulnerability crawling. The first crawling process is shown in Figure 7 [Figure 7: see original paper].

During the first crawling process, after extracting a URL from the to-be-crawled URL queue, a request is first sent with legitimate parameters to obtain and parse the returned page, with the purpose of extracting new to-be-crawled URLs to add to the queue. Then, request parameters are adjusted to inject mutated attack vectors, and the request is repeated with the returned page saved. This process is looped until all types of attack vectors have been injected. To successfully request and obtain returned pages, the system performs semantic analysis on URL request parameters during page parsing, using the tag word corresponding to the request parameter and the parameter' s key string to ensure request parameter legitimacy. For example, if a request parameter' s tag word is "预约时间" (appointment time), it can be determined that the legitimate input for this parameter should be in date format, or if a request parameter' s key string is "startSalesAmount," the legitimate input should be numeric format.

During the second crawling, request parameters are no longer adjusted, and URL parameters obtained by the system crawler are used directly for requests. Since attack vectors have already been injected into the server side through stored vulnerabilities during the first crawling, pages crawled during the second crawling must contain output point pages. Additionally, because the attack vectors injected during the first crawling contain input point information, this ensures consistency between input and output points.

3.2.4 Vulnerability Detection After completing the two crawling and parsing processes, the system has saved pages obtained from each request to the target system. The vulnerability detection process only needs to traverse these pages and analyze whether the injected attack vector information exists in the page content. If it exists, it is identified as an XSS vulnerability, and vulnerability information is recorded based on the output point information contained in the attack vector. If it does not exist, it is ignored. The specific process is shown in Figure 9 [Figure 9: see original paper].

4 Experimental Analysis

To verify the feasibility and effectiveness of this system in detecting XSS vulnerabilities, this paper designed two sets of comparative experiments. Experimental metrics include vulnerability detection count, vulnerability false negative rate, and vulnerability false positive rate. Detection count represents the actual number of vulnerabilities detected, calculated based only on URLs (not request parameters), meaning a URL is counted only once even if multiple request parameters can inject attack vectors. The false negative rate is the ratio of vulnerabilities not discovered by the detection system to the actual number of vulnerabilities, while the false positive rate reflects the accuracy of detected vulnerabilities.

The first comparative experiment used this detection system and the AWVS (Acunetix Web Vulnerability Scanner) tool to test an inter-bank business information sharing and exchange platform. AWVS is a commercial vulnerability

scanning software with paid and free versions; this experiment used the free version for comparison. The information sharing and exchange platform is a standard JSP+Struts2+MySQL+Hibernate B/S system currently in use. Since the vulnerability status of this system is unknown, vulnerability detection count and false positive rate were used as experimental metrics. The experimental data comparison results are shown in Table 6 .

The second comparative experiment used this detection system and the AWVS tool to test an XSS vulnerability experimental platform. The XSS vulnerability experimental platform is a self-built platform for stored and reflected vulnerability detection with known vulnerabilities. Experimental metrics included vulnerability detection count, false negative rate, and false positive rate. The comparative experiment results are shown in Table 7 .

The results from Experiment 1 show that under unknown vulnerability conditions, this system' s detection count is much higher. Experiment 2 results show that this system' s false negative rate is lower than the AWVS tool. These results occur because the attack vectors generated by the EBNF paradigm have very high complexity and can effectively bypass server-side validation and filtering mechanisms to inject malicious code. Since the system filters URLs with deletion functions, attack vectors injected for stored vulnerabilities will not be deleted during the crawling process, allowing detection of a large number of stored vulnerabilities during the second crawling process and significantly reducing the false negative rate. The results from both sets of experiments show that this system' s false positive rate is 0 in both tests because the attack vectors contain input point information, ensuring consistency between input and output points during vulnerability detection and thus preventing false positive vulnerability reports.

5 Conclusion

This paper studied the principles of XSS vulnerabilities in Web systems and related detection technologies, designed an XSS vulnerability detection system based on penetration testing technology, and implemented a system prototype. To address the problems of low-complexity attack vectors that are easily filtered and redundant detection processes in existing detection systems, this system proposes a high-complexity attack vector automatic generation method based on the EBNF paradigm, which can effectively bypass server-side filtering. By embedding input point information in malicious script fragments of attack vectors, the detection process is simplified and false positives are almost completely avoided. Simultaneously, through twice-crawling, false negatives for stored vulnerabilities are further reduced, lowering the overall false negative rate. Finally, two sets of comparative experiments demonstrated the feasibility and effectiveness of this system in detecting XSS vulnerabilities in Web systems.

Currently, the system does not address how to counter WAF firewalls and anti-crawling measures, limiting its use to developers' own Web systems. Therefore,

certain limitations exist. In future work, we will optimize and improve this issue to expand the usage scope and perfect the XSS vulnerability detection system.

References

[1] Jeff W, Dave W. OWASP top 10-2017 [EB/OL]. (2018-03-27) [2018-04-12]. https://www.owasp.org/index.php/Top_10-2017_Top_10.

[2] Ryan O. WHS 2017 application security report FINAL [EB/OL]. (2017) [2018-04-12]. https://www.whitehatsec.com/resources-category/premium-content/web-application-stats-report-2017/.

[3] Azmi S A, Khan A R. A comprehensive research on XSS scripting attacks on different domains and their verticals [C]//Proc of the 4th International Conference on Computer Science and Network Technology. Piscataway, NJ: IEEE Press, 2015: 677-680.

[4] Barus A C, Hutasoit D I P, Siringoringo J H, et al. White box testing tool prototype development [C]//Proc of the 5th International Conference on Electrical Engineering and Informatics. Piscataway, NJ: IEEE Press, 2015: 298-303.

[5] Cao Libo, Cao Tianjie. Research on cross-site scripting vulnerability detection method based on dynamic testing [J]. Computer Applications and Software, 2015, 32(8): 272-275.

[6] Medeiros I, Neves N, Correia M. Detecting and removing Web application vulnerabilities with static analysis and data mining [J]. IEEE Trans on Reliability, 2016, 65(1): 54-69.

[7] Yu Xueyong, Jiang Guohua. A method for detecting cross-site script [J]. Journal of Chinese Computer Systems, 2015, 36(8): 1763-1768.

[8] Li Wei, Li Xiaohong. Detection method for stored-XSS vulnerability in Web applications and implementation [J]. Computer Applications and Software, 2016, 33(1): 24-27, 37.

[9] Gu Mingchang, Wang Dan, Zhao Wenbing, et al. A XSS vulnerability penetration test method based on attack vector automatic generation [J]. Software Guide, 2016, 15(7): 173-177.

[10] Wu Zijing, Zhang Xianzhong, Guan Lei, et al. Technique for deep discovering XSS vulnerability based on anti-filter rules set and automatic crawler program [J]. Transaction of Beijing Institute of Technology, 2012, 32(4): 395-401.

[11] Rathore S, Sharma P K, Park J H, et al. XSS classifier: an efficient XSS attack detection approach based on machine learning classifier on SNSs [J]. Journal of Information Processing Systems, 2017, 13(4): 1014-1028.

[12] Gupta S, Gupta B B. Cross-site scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art [J]. International Journal of System Assurance Engineering & Management, 2017, 8(Suppl 1): 512-530.

[13] Dong G, Zhang Y, Wang X, et al. Detecting cross site scripting vulnerabilities introduced by HTML5 [C]//Proc of the 11th International Joint Conference on Computer Science and Software Engineering. Piscataway, NJ: IEEE Press, 2014: 319-323.

Note: Figure translations are in progress. See original paper for figures.