ISBN: 978-981-18-7950-0
2023 the 13th International Workshop on Computer Science and Engineering (WCSE 2023)
doi: 10.18178/wcse.2023.06.006

Enhancing Website Runtime Application Self-Protection by Using Tainting and Fuzzing Test

Rui Shi [1 +], Gaozhou Wang [2], Hang Yu [2], Long Zhang [3] and Haipeng Qu [1]

[1] College of Information Science and Engineering, Ocean University of China, Qingdao, Shandong, China

[2] Information and Telecommunications Company, State Grid Shandong Electric Power Company, Jinan, Shandong, China

[3] QI-ANXIN Technology Group Inc., Beijing, China

Abstract. More and more people are depending on web applications in their daily life as a result of the development of big data and cloud computing technologies. A lot of data is processed through web applications, however, security vulnerabilities in web applications provide a continuing threat to people's private data. Traditional black-box scanners and white-box analysis tools struggle to identify vulnerabilities for runtime web applications precisely and effectively in the production environment, and web application firewalls are unable to get the runtime context environment in which they are executed, leading to a large number of false positive reports. In this paper, we propose a novel fuzzing and tainting-based system for web application runtime self-protection, which simultaneously implements web application vulnerability detection and attack protection from adversaries at runtime and generates comprehensive vulnerability reports for website administrators along with potential fix recommendations and patch code. We implemented and evaluated a basic prototype system for XSS vulnerabilities in PHP-based web applications. Our results show that although RASP based on fuzzing and tainting causes about a 30% reduction in website efficiency, it is fully defendable against almost all XSS attacks, while the fuzzer finds new data flow paths that could trigger the same XSS vulnerability and provides corresponding remediation recommendations.

Keywords: rasp, fuzzing, tainting, website security

1. Introduction

With the advancement of big data and cloud computing technology, people cannot live without online web applications in areas such as communication, social networking, online shopping or media. A large amount of user data is transmitted and processed through web applications, and the security of web applications is receiving more and more attention. Depending on the type of vulnerabilities that exist in web applications, they can cause different levels of damage to websites. Some low-level vulnerabilities may result in a brief Denial of Service (DoS) or data leakage, while higher-level vulnerabilities may potentially allow the leakage of sensitive user data and complete control of the web server, leading to significant financial losses [1]. Web application vulnerability identification and runtime protection have been a focus of security research to prevent such losses.

Typically, black-box scanners or white-box code analysis tools are used by developers or security auditors to identify vulnerabilities in coding or debugging environments. Although these two methods for identifying vulnerabilities beforehand might lessen or even prevent unforeseen damage in production environments, they nevertheless have certain drawbacks. Black-box scanners are limited in their ability to detect vulnerabilities by relying solely on the response content of an HTTP request discovered by a website crawler, which frequently produces many false positives and false negatives. White-box analysis tools lack the access to the real web application runtime environment, which increases the likelihood of false positives

Compared to black or white-box tools, gray-box fuzzing is a better technique to explore program execution paths and detect vulnerabilities. More execution paths are likely to result in more vulnerabilities.

+ Corresponding author.
E-mail address: shirui@stu.ouc.edu.cn.

Fuzzer randomly mutates seeds from raw user inputs in bytes and re-sends them to web applications, expecting path feedback to guide the next round of mutation based on the genetic algorithm.

Vulnerability defense and remediation are as important as detection. Traditional Web Application Firewalls (WAF) intercept and detect whether user input contains illegal data, however, the lack of a web application runtime context environment frequently results in many false positive reports. Based on runtime vulnerability detection, Runtime Application Self-Protection (RASP) is a better option than traditional WAF.

In this paper, we present the architecture of a novel system for web application self-protection and runtime vulnerability detection. The basic concept behind our RASP system is to combine taint tracing for RASP into runtime web applications together with the coverage-guided gray-box fuzzing test. The RASP system taints all untrusted user input at first when the instrumented web application receives a new HTTP request, and inspects the arguments of the dangerous functions that might lead to vulnerabilities to check if these arguments contain any tainted data before the functions execute. The tainted input will be regarded as a new fuzzing seed for the gray-box fuzzer after a tainted data flow path from the HTTP input to the dangerous function is identified by tainting. Based on the specified data flow path, the gray-box fuzzer tries to discover additional paths that can trigger the same dangerous function and finally offers the website administrator a complete vulnerability report with potential patch codes and repair recommendations.

Our RASP system is more focused on website defense using tainting and fuzzing than vulnerability detection. A prototype for XSS vulnerability detection and defense system for PHP-based web applications is implemented and evaluated based on our design.

In summary, this paper makes the following contributions:

We propose a novel RASP system design that combines tainting with fuzzing for runtime vulnerability protection and detection.

We implement a basic prototype that targets the XSS vulnerability for PHP-based web applications and evaluate it on some real-world PHP websites.

We present preliminary results from defending real-world web application attacks using our RASP system design.

2. Background

2.1. Coverage-Guided Gray-Box Fuzzing

Traditional fuzzing test provides new malformed inputs to the runtime software by randomly mutating the original inputs and observing for unexpected behaviours such as errors, bugs, or even vulnerabilities [2]. Gray-box fuzzing is a popular automated software testing technology, and coverage-guided fuzzing has become the mainstream of automated vulnerability discovery, where large amounts of raw data are used to fuzzing through some special mutation strategies and guide the next mutation by feedback information. The basic idea of fuzzing is to repeat the process by gathering feedback information from instrumented program execution, processing the mutated samples based on the feedback data, eliminating the low-quality test cases, and then guiding the mutated samples in the subsequent round of testing to produce new samples for input into the target testing program.

The fuzzing has been widely used in web application vulnerability detection in recently. To improve the efficiency of crawlers for fuzzing, A. Doupé et al. [3] implemented black-box state exploration and rollback to improve code coverage based on DFA to determine the state of web applications by parsing HTML DOM-tree and locate HTTP url in the tree. O. V. Rooij et al. [4] applied gray-box fuzzing to web applications and successfully detected XSS vulnerabilities through crawler analysis of front-end JavaScript AST in PHP applications, covering more source code quickly compared to other black-box scanners. Gauthier et al. [5] implemented a grey-box fuzzing model for HTTP interfaces in REST format combined with static analysis, by trying to infer the RESTful API and its parameters from the HTTP interface information collected by the crawler in front-end page, and improving efficiency of fuzzing through coverage feedback.

2.2. Taint Tracing

Taint tracing, or tainting for short, is a dynamic testing technology for runtime vulnerability detection. Similar to fuzzing, tainting is likewise interested in the runtime information of the web application, but the two are different in that the former uses the execution path as feedback information, and the latter solely concentrates on the input and output relative to the execution path. Before the web application is launched, all source inputs from HTTP request packets will be marked with specific tags identifying them as untrusted user inputs. The tainted data will be spread through variables and functions in web application. Finally, in target dangerous sink functions that may trigger the vulnerability, check the arguments for runtime parameter and report the vulnerability if any parameter contains tainted data. Compared to exploring paths through random inputs in fuzzing, tainting can accurately report the inputs corresponding to runtime-triggered vulnerabilities.

2.3. Runtime Application Self-Protection

The Runtime Application Self-Protection (RASP) is a novel runtime protection technique for the application layer. Special daemon code is injected into the web application and integrated with it to protect the web application from adversary attacks in runtime environments. Traditional WAF filters malicious requests mainly by analysing the features in HTTP traffic and intercepting HTTP requests carrying attack features [6]. Although WAF can effectively filter out the majority of malicious requests, it is unable to comprehend the context environment in which the web application is running, which will inevitably lead to a certain number of false positives [7]. Compared to other traditional security protection techniques, real-time is the most important feature of RASP, and it can analyse application behaviour in a runtime context, detect attacks, and report possible vulnerabilities.

In recent work, E. Rajesh et al. [8] proposed an automated RASP system for preventing web parameter attacks using mobile agents on the client side, by extracting keywords from web application parameters on the client side and using gene alignment to compare the identity between two parameter sequences, it can only work in mobile applications. Tainting was successfully applied to the RASP of PHP web applications through the work of I. Papagiannis et al. [9], where variables of string type are converted to arrays that record meta information and determine whether tainted data exists by the information in the array to defence attacks from adversary, however, this approach seriously slows down the execution efficiency and can only achieve partial tainting.

3. System Design

3.1. Overview

We design a new RASP system based on tainting and fuzzing in our work. Fig. 1 shows the overview of the system architecture. Through the instrument, special code snippets that record runtime information are inserted into the original web application source code and hook some functions to spread the tainted data. The web application with RASP enabled runs in the web container and is prepared to process HTTP requests when the instrumented web application has been deployed to the web server.

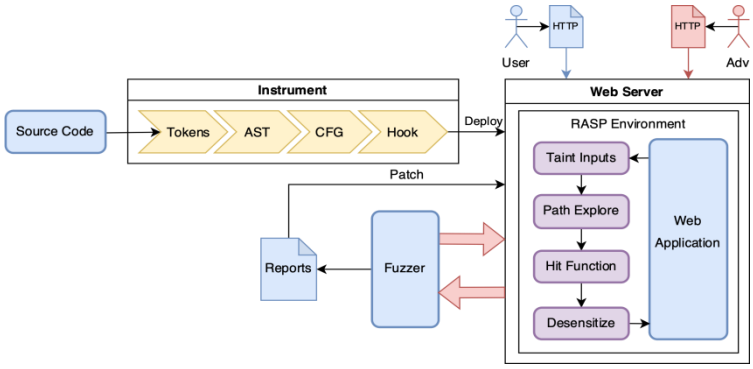

Fig. 1: The overview of fuzzing-RASP architecture.

The RASP system uses tainting to provide complete protection for the web application. Every HTTP input will be marked as tainted and handed over to the web application for execution. When the web application is running, additional instrumented code snippets record details about the execution stream path,

including the basic blocks and edges that the stream passed, while the hooked functions spread the tainted data until the data flow path eventually encounters the dangerous function that could cause a vulnerability and checks whether the provided data is tainted.

The tainted data will be desensitized and continue to be handled by the web application after being recorded by the hooked dangerous function to ensure that the website runs normally, but the tainted data and its path information are saved locally, as raw seeds for fuzzing the web application in runtime. An external coverage-guided fuzzer gathers saved vulnerabilities, user inputs, and other path information. The fuzzer then tries to find more paths against the same dangerous functions and generates a detailed vulnerability report with suggested patch code to fix the detected vulnerability.

3.2. Implementation

First, the source code of the web application will be instrumented to hook input and potentially dangerous functions that might result in vulnerabilities and other functions that could pass data. The original web application source code can be tokenized using a lexical scanner, and a token stream is generated as input for the syntax parser. To obtain the integrated Abstract Syntax Tree (AST), the syntax parser checks the syntax against the token stream provided by the lexical scanner and generates an AST from the token stream. For example, fig. 2 shows the workflow of generating AST from a simple PHP source code, compared to the original source code, the tree-structured AST omits many syntax details, but the control flow implicit in the AST still resembles the control structure of the original code.

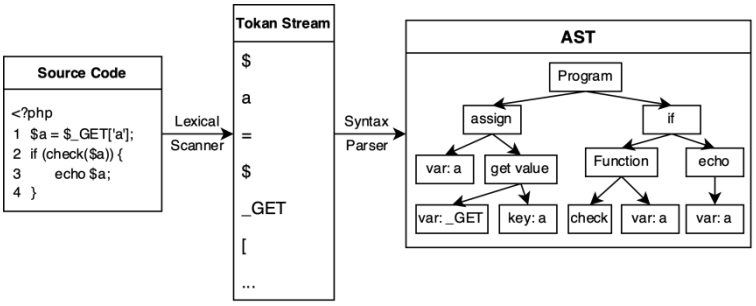

Fig. 2: The example of generating AST from a simple PHP source code.

Although the AST can be used directly to analyze the execution flow path, it still contains a lot of redundant syntactic information that increases the complexity of analysis, and a Control Flow Graph (CFG) generated based on the AST is a perfect solution for this challenge. Each basic block is placed into the CFG as a graph node, and an edge in the CFG is represented by a triple-tuple *<from, to, condition>*, including the source basic block, the destination basic block and the execution jump condition, showing in fig. 3, the PHP source code in fig. 2 contains four basic blocks with four possible edges. To spread the tainted data and check dangerous functions, the function *check* and *echo* will be hooked by analyzing generated CFG. At the same time, the basic block is inserted as the smallest instrument unit into the code snippets that record the edge information to collect execution path information for coverage-guided fuzzing.

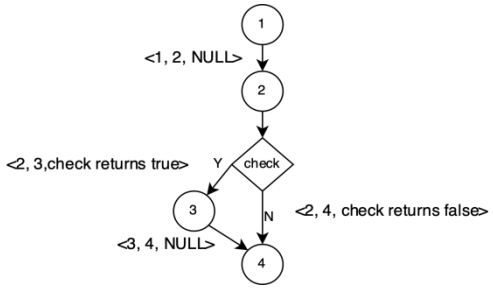

Fig. 3: The example of control flow graph with triple-tuples based on fig. 2.

The RASP-enabled web application that contains instrumented code snippets with the basic blocks can be deployed directly in the web server and used as a regular application. Any input from the user will be tainted before the web application starts, and the web application will continue to process the tainted data. During the web application processing, these code snippets will record the execution paths of the data flow and eventually generate a Data Flow Graph (DFG). When the tainted data arrives at a dangerous function,

the DFG and the corresponding input can be saved as the seed for fuzzing, while the tainted data will be checked and desensitized to avoid adversary attacks.

The input from untrusted users and generated DFG from tainting is used for path exploration by coverage-guided fuzzing. The goal of fuzzing is to try to explore more paths that may arrive at the same dangerous function from the same user input. Relying on the generated DFG, the mutation strategy of fuzzing will be guided to improve the coverage as much as possible. Finally, a detailed fuzzing report will be generated to the website administrator after the fuzzing finished, along with a proposed patch code to repair these unsafe data paths after there are no more paths that can be explored during fuzzing.

3.3. Evaluation

Based on the architecture we designed before, we have implemented a simple demo program against XSS vulnerabilities to verify the feasibility of the design in a PHP application. The instrument is implemented through the extension ability provided by PHP itself, and the function hook is based on the PHP extension *XDebug*. The coverage-guided fuzzer in the RASP system is developed by using Python.

We test and evaluate the demo program, and choose *DVWA*, *WordPress* and *Joomla* as test subjects. *DVWA* is a vulnerability range which collects many commonly known Web vulnerabilities. *WordPress* is a large-scale use blog system for quickly building a personal blog. *Joomla* is also a large-scale easy-to-use CMS. We select a set of benchmarks to test the impact of our RASP system on efficiency for each website. Table 1 shows the network throughput between the non-protection and RASP protected website in four hours.

Table 1: Network throughput between the original website and RASP protected website.

Application Name	Request Rate (requests/s)				Efficiency Ratio
	Original Website		RASP Website		
	Average	Maximum	Average	Maximum	
DVWA	69.79	151.77	51.62	146.72	73.96%
WordPress	7.03	133.74	4.57	128.15	65.00%
Joomla	6.27	127.78	4.19	121.37	66.83%

For the small-scale web application like *DVWA*, RASP caused a smaller impact, with an average website throughput drop of 26.04%. For some large-scale web applications like *Joomla* or *WordPress*, the average throughput dropped by nearly 35%. Websites that inevitably turn on RASP protection pay an extra price for security, but the average impact of our RASP design with tainting on web applications is around 30%.

To verify the defensive effect of the RASP system, we collect six common typical XSS payloads and use *DVWA* as the test subject because it contains many XSS vulnerabilities with three different level protection, which all three levels can be exploited or bypassed by some special payloads. We build an obfuscation tool to use some usual XSS payload bypass technique for randomly mutating these basic payloads to see how far we can go. For each level, the basic and mutated payloads, along with nearly 50% randomly generated normal data are mixed in random order and keep sending to the XSS vulnerability test page in DVWA which enabled our RASP protection in four hours. Table 2 shows the defense result for each level, FP (False Positive) is the normal data but is reported as a payload, FN (False Negative) is missed payloads, and the Path is extra data flow path that RASP-fuzzer found.

Table 2: The defense result for XSS in RASP protected DVWA.

DVWA Level	Total Requests	Payloads	Normal Data	FP	FN	Path
low	974,482	501,858	472,624	5	439	8
medium	926,101	458,527	467,574	18	54	5
high	1,003,539	490,733	512,806	9	15	4

Our results showed that the most payloads can be defended perfectly by RASP. We noticed that in low level test, 439 payloads were missed because some data spread functions that we did not hook for keeping a high execution efficiency. Besides, the RASP-fuzzer provides a lot extra data flow paths that could cause the same XSS vulnerability by the path exploration, and the location of vulnerability is accurately reported by the fuzzer.

4. Limitations

Although the RASP system based on tainting can provide real-time protection for web applications, RASP is more difficult to implement compared to traditional WAF. Inevitably, RASP reduces the efficiency of web applications when we hook too many functions to spread tainted data. In addition, for many server-side programming languages with flexible features, such as Python or PHP, it is difficult to construct an accurate CFG from the AST. A better solution is to generate a semantically equivalent but simpler Intermediate Representation (IR) based on syntactic analysis statically and generate an accurate CFG based on the IR. Find a good mutation strategy for fuzzing is still a challenge. Traditional random mutation cannot quickly explore the paths associated with HTTP inputs, and more research is required for the mutation strategy of HTTP request packets.

5. Conclusion

Fuzzing and tainting are the better technologies used for web application runtime vulnerability detection and attack protection compared to traditional web scanners or static analysis tools. In this work, we design a new RASP architecture based on tainting and fuzzing. The input from users that transmitted by HTTP is marked as tainted data, and the functions that spread tainted data can be hooked by instrument. While the web application processing tainted data, the parameter content of the dangerous function will be captured and checked to determine whether it may cause unintended vulnerabilities, and finally the tainted data can be desensitized to achieve runtime self-protection of the web application.

Based on RASP that we combine the coverage-guided fuzzing to explore more paths that could cause the same vulnerability, with the CFG and DFG guided mutation strategy, the fuzzing can quickly explore new paths. After exploring all the paths associated with the dangerous functions, a detailed vulnerability report is provided with the suggested patch code to fix the unsafe data flow paths that found by fuzzer. We have implemented a simple RASP prototype system against XSS based on the solution we designed for PHP application and performed a proof of concept. After the testing and evaluation, our work can protect web applications from XSS attacks perfectly with minimal efficiency impact.

6. References

[1] Sausalito C. Cybercrime to Cost the World $10.5 Trillion Annually By 2025. *Cybercrime Magazine*. 2020.

[2] A. Fioraldi, D. Maier, H. Eißfeldt and M. Heuse. AFL++: combining incremental steps of fuzzing research. *In Proceedings of the 14th USENIX Conference on Offensive Technologies (WOOT'20)*. 2020.

[3] A. Doupé, L. Cavedon, C. Kruegel and G. Vigna. Enemy of the state: a state-aware black-box web vulnerability scanner. *In Proceedings of the 21st USENIX conference on Security symposium (Security'12)*. 2012.

[4] O. V. Rooij, M. A. Charalambous, D. Kaizer, M. Papaevripides and E. Athanasopoulos. WebFuzz: grey-box fuzzing for web applications. *In Computer Security – ESORICS 2021: 26th European Symposium on Research in Computer Security*. 2021, 152-172.

[5] F. Gauthier, B. Hassanshahi, B. Selwyn-Smith, T. N. Mai, M. Schlüter and M. Williams. BackREST: a model-based feedback-driven greybox fuzzer for web applications. *arXiv, <https://arxiv.org/abs/2108.08455>*. 2021.

[6] K. R. Khamdamovich, I. Aziz. Web application firewall method for detecting network attacks. *International Conference on Information Science and Communications Technologies: Applications, Trends and Opportunities*. 2021.

[7] T. S. Riera, J. R. B. Higuera, J. B. Higuera, J. A. S. Montalvo and J. J. M. Herráiz. Systematic approach for web protection runtime tools' effectiveness analysis. *Computer Modeling in Engineering and Sciences*. 2022, 133 (3): 579-599.

[8] E. Rajesh, R. Raju and R. Ezumalai. Mitigation of Web Based attacks using Mobile Agents in client side. *International Journal of Computer Theory and Engineering*. 2010, 2 (5): 746-750.

[9] I. Papagiannis, M. Migliavacca and P. Pietzuch. PHP aspis: using partial taint tracking to protect against injection attacks. *In Proceedings of the 2nd USENIX conference on Web application development (WebApps'11)*. 2011.