Static Detection of Access Control Vulnerabilities in Vue Applications

Kefeng Pan and Qian Wang
School of Software & Microelectronics, Peking University, Beijing, China.
Email: pankefeng@pku.edu.cn

Abstract. In this article, we reviewed the research progress of mainstream methods for detecting access control vulnerabilities, and designed a method for detecting access control vulnerabilities in Vue applications based on previous research. After that, we tested our detection method on the vulnerability test set designed by ourselves, and basically achieved the research goal of this article.

1. Introduction
Web applications play an important role in modern society because they play a huge role in e-commerce, social networking, and finance. With their openness and wide deployment, a large number of criminals attempt to obtain user data and resources from them. Therefore, the security of Web applications has become an extremely important issue.

Access control technology is to prevent unauthorized access to any resources, thereby ensuring the security of each user data in the application and the stability of the system. Web applications store a large amount of user privacy data. In order to ensure the security of privacy data, developers will adopt strict access control in applications. In WEB applications, every time a user accesses privileged resources, a new HTTP request needs to be parsed to identify the previously logged-in user, which leads to stricter requirements for access control policies in WEB applications.

There are several reasons for the access control vulnerabilities. First of all, the access control functions of WEB applications are written by developers, but during the development process, they often pay more attention to the development of other key functions, so missed access control detections may occur. Secondly, Web applications often have multiple authorization roles, and the logic of access control policies is very complex. Limited by the coding ability of developers and lack of knowledge about access control design, incomplete or wrong access control often occurs. In addition, a Web application usually connects directly to the database as a super user, and it has the highest authority. Any small error in the authorization logic may cause serious data damage.

Since the 1990s, researchers have proposed many methods and tools for access control vulnerability detection. We have investigated relevant documents since 2011, which are mainly static detection methods.

The contributions in this paper are:

- Summarizes four common detection techniques in the field of access control vulnerability detection.
- Based on previous studies, we have optimized the relevant algorithms and selected a more suitable parser and solver to adapt to Vue.
2. Related Work
There are two common detection methods for access control vulnerability detection, namely dynamic detection and static precision detection. Dynamic detection methods are often difficult to find hidden pages and to obtain the expected access permissions of each user. In addition, since user input is usually restricted, the pages covered by dynamic detection are often shallow and incomplete. Sun et al. [7] first proposed a static detection method for access control vulnerabilities at the UIXUSE conference in 2011. The static detection method has better coverage than the dynamic detection method. After that, static detection has become the mainstream direction of the academic research on access control vulnerability detection.

At present, the biggest difficulty that researchers face in static detection methods is to extract accurate access control strategies. On the one hand, there are many types of web applications, and their access control strategies are often different from each other. On the other hand, developers often do not write an access control strategy specifically for web applications, which requires researchers to extract the strategy from the source code. Analyzing the source code involves extremely cumbersome manual work, and also faces the situation that developers have errors in their implementation. In response to these problems, researchers have proposed some solutions. We can summarize them into the following four types: Force browsing, Context-based Detection Technique, Model-based Detection Technique, Graph-Based Detection Technique.

In this section, we specifically introduce the technical routes of these four static detection methods and briefly analyze their characteristics.

2.1. Force Browsing
Force browsing means the act of directly accessing a page, that is, the visitor does not access the page through a link displayed on the page, but through some brute force techniques. For an attacker, the URLs of certain pages in the application are highly predictable, such as files named by date. Researchers believe that when the HTML page obtained through force browsing is the same as the HTML page obtained through normal access, and there is no redirection, exception or error during the page rendering process, the force browsing is successful.

Force browsing technology is often used to simulate the behavior of low-level users accessing resources of high-level users. This technique was first proposed by Sun et al [7].

2.2. Context-based Detection Technique
Context-based Detection Technique first needs to extract security-sensitive operations in the program. Researchers generally use context-free grammars and other means to accomplish this. Researchers believe that the same security-sensitive operation should have the same context and set of redirects. By analyzing the consistency of the two, you can determine whether the program has access control vulnerabilities [9].

This technology does not require access control policies for web applications. In web applications, access control policies are usually hidden in the code and are difficult to extract. This technology is cleverly avoiding this problem.

2.3. Model-based Detection Technique
Developers often use some frameworks to implement access control functions, and this technology is for access control in such web applications. Researchers first need to analyze an access control implementation framework, and then define a model to describe this framework. In the model, access control attributes are included. Finally, use the model to analyze whether the web application correctly implements the access control strategy to determine whether there are vulnerabilities [10].

The biggest difficulty of this technology is to describe the relationship between access control attributes and other attributes in the model. Different frameworks often have different definitions. And the characteristics of various development languages are different. This technology can only be implemented for a specific framework in a development language, and cannot detect access control vulnerabilities in other types of web applications.
2.4. Graph-Based Detection Technique

There are many ways to implement this detection technology. Some researchers use control flow graphs to describe web applications, some use site maps to describe them, and some use permission verification graphs. The nodes in the graph often represent a page or a resource in a web application. Researchers often use algorithm constraints, flow analysis and other methods to detect the reachability between nodes. If an unprivileged user can access a privileged node by some means, then the node has access control vulnerabilities [7] [8] [11].

This technology has excellent scalability, and suitable methods can be selected according to the characteristics of the web application during node detection. Therefore, most researchers have adopted this technique.

3. Detection Method

We use context-free grammar to describe the site map of each character, in the form of $CFG = (V_T, V_N, P, S)$. In this article, we mark each page as a symbol, where $V_T$ is a set of terminal symbols. The inactive node is defined as the terminator of a role's site map. $V_N$ is the non-terminal set. We use the active node as the non-terminal. $P$ is the production set. We use the production to calculate the reachability of the node. $S$ represents the start symbol. We use the entry node of each character in the Vue application as the value of the start symbol.

Assume that there are two different roles $a, b$ in the Vue application, and the permissions of $a$ are higher than $b$. We obtain the privileged nodes by comparing the site maps of role $a$ and role $b$. We believe that when a user belonging to role $b$ accesses a privileged node through a navigation path allowed by the system, we believe that this node is vulnerable to an access control vulnerability. Therefore, our detection algorithm is divided into the following two steps:

1. According to the input specifications and regular expressions, construct two context-free grammars with different permission roles.
2. The privileged node is obtained by comparing the site maps of the two roles. Privileged nodes are detected through forced access, and the vulnerabilities are analyzed by comparing the context-free grammars and redirection sets obtained.

The Sitemap builder is the most important part of our detection method. It has two components: link extractor and context-free grammar constructor. We construct a context-free grammar for each page and use a link extraction algorithm to extract links from it, so that we can find the outgoing edge of each node.

A Vue page contains three pieces of content. HTML code is written in the <template> module, CSS code is written in the <style> module, and javascript code is written in the <script> module. In order to generate each character site map, we need to analyze the HTML code block and javascript code block in the Vue page.

The <script> module contains the code of access control detection. These codes detect whether the current access role is the authorized role of the page, and determine whether the current role has the permission to jump to the next page. The <template> module contains the link to the next node that the current page can reach. We need to extract these links to find the edge of the current node.

3.1. Context-free Grammar Constructor

Figure 1 shows the process to construct context-free grammars. First convert the js source code to AST through Acorn, and then convert it to IR through Jimple. Collect algorithm constraints by traversing IR, and then solve the constraints by Z3 solver to update the context-free grammar. Finally build a site map of the role.
Traversing the IR in a top-down manner, constantly updating the system's algorithm constraints during the analysis, especially the value of conditional control statements, determines the actual execution path of the program. The conditional expression in the if statement generally uses an expression that is equal to or not equal to this return Boolean type to determine whether the user is an authorized user. Therefore, our system aims to collect such path constraints when traversing the IR.

For solving the algorithm constraints that have been established, we refer to the SMT (Satisfiability Modulo Theories) solver Z3. We take the role's access control information as input, including the role type, role status information, etc., and solve the algorithm constraints through the solver. If there is a solution, we think that the role can access the node.

We use user information and nodes as tokens. For users who can access the current node after solving, we can generate a new generation rule. According to this rule, in this context-free grammar, a grammatical result of a path that an authorized user is allowed to access can be obtained.

In this way, we continuously update the context-free grammar of each role, and eventually the CFG will be able to generate the grammatical results of all access paths of the role. At this point, the construction of a character's context-free grammar is complete.

3.2. Link Extractor

The algorithm for link extraction is as follows:

\[
\text{EXTRACTLLINKS}(\text{template}, \text{dfa})
\]

1. \( \text{AST} \leftarrow \text{templateParser}(\text{template}) \)
2. \( \text{IR} \leftarrow \text{constructIR}(\text{AST}) \)
3. \( \text{Vnode} \leftarrow \text{constructVnode}(\text{IR}) \)
4. \( \text{HTML} \leftarrow \text{htmlParser}(\text{Vnode}) \)
5. \( \text{LINKS} \leftarrow \text{walkHTML}(\text{HTML}, \text{dfa}) \)
6. \( \text{return LINKS} \)

Unlike traditional HTML, links in Vue pages sometimes do not appear in tags, but use new tag attributes to bind data, as in the following example:

\[
< a \text{ v-bind:href='link' } > < / a >
\]

The v-bind attribute here indicates that the href attribute of the a tag is determined by the link variable in the data. Only using the above method can only obtain the variable name of link, but not the actual link address.

In Vue, the template string in <template> is dynamically converted into a real DOM node through
parsing and rendering. We analyzed its parsing principle and formulated a plan to parse the template string. This plan contains the following three steps:
1. Parse the `<template>` template string to AST.
2. Convert the AST into a virtual DOM node (Vnode). The virtual DOM node is actually a string of json data, which describes the information of the real DOM node, such as DOM tags, attributes, content objects, etc., all become the data in the object.
3. Vnode is transformed into a real DOM node.

3.3. Vulnerability Detection
The algorithm of vulnerability detection is as follows:

\[
\text{DetectVul(Privileged, Spec}_a\text{, Spec}_b) =
\begin{align*}
&\text{for each } n \text{ in Privileged} \\
&\text{do } <cfg_a, R_a> \leftarrow \text{GETCFG}(n, \text{Spec}_a) \\
&<cfg_b, R_b> \leftarrow \text{GETCFG}(n, \text{Spec}_b) \\
&\text{if } \text{SIZEOF}(cfg_a) = \text{SIZEOF}(cfg_b) \text{ and } R_a = R_b \\
&\text{then } \text{Vuls} \leftarrow \text{Vuls} \cup \{n\} \\
&\text{return } \text{Vuls}
\end{align*}
\]

When the construction of the site map for each character is completed, our analysis program will compare the two reachable node sets to infer privileged nodes. Since the HTML output presented to different roles is usually different, the set of privileged nodes is not empty in most cases. After obtaining the privileged node set, our analyzer again uses the same context-free grammar constructor to approximate the result of forced browsing. Finally, it compares the derived redirection set with two new context-free grammars to determine whether the forced browsing attempt was successful.

4. Experimental Data Analysis
In order to evaluate the detection effect and execution efficiency of our tools, we refer to several access control vulnerabilities in CVE and open source applications. We design two types of vulnerability prototypes and patch versions of these two types of vulnerability prototypes.

AccVulPrototype1 is used to compare access control violations between administrator roles and ordinary user roles. AccVulPrototype2 is used to compare access control violations between ordinary user roles and guest roles. There are two main reasons for prototyping these two types of vulnerabilities. First of all, administrator roles are usually associated with ordinary user roles. Administrator roles can generally access more important resources than ordinary users, and will have the authority to manage ordinary user roles. It is usually difficult for attackers to obtain administrator accounts legally, and it is easier to obtain accounts with ordinary user roles. Secondly, the guest role generally does not have a specific account. The guest role has fewer permissions than the ordinary user role. Both can access public resources, but the guest role does not have the permission to access the private resources of the ordinary user role.

AccVulPrototype1 refers to CVE-2006-5909[14]. The software involved in the vulnerability is SCARF, and the access control detection is lacked in the "require_admin()" function in the "generaloptions.php" file. It does not verify whether the user is an administrator role, resulting in ordinary roles that can edit the application configuration. In our vulnerability prototype, we set up these two roles. The administrator user can set the background color of the application in the config.vue page. The ordinary user role does not have this permission. On the basis of this access control, we have deleted the code segment in the config.vue page to verify user roles, so that ordinary roles can override the background color.

AccVulPrototype2 refers to a vulnerability in the miniBloggie blog application [11]. In this application, there are only two roles: normal user role and guest role. Therefore, the application uses the session detection method like $SESSION['user'] to authenticate the user. This detection is implemented using a verifyuser() function. In the del.php file of the application, this verification is missing, resulting in that even the guest role can delete blogs. Our vulnerability prototype uses a checkuser() function to verify the user role. We have designed a table data. The common user role has the permission to add, update, and delete data. In the method of deleting data, we do not use the
checkuser function to simulate this.

Table 1. Vulnerability test results.

| Application          | Privileged node | Reported vulnerability | Confirmed vulnerability |
|----------------------|-----------------|------------------------|-------------------------|
| AccVulPrototype1     | 3               | 1                      | 1                       |
| AccVulPrototype2     | 2               | 1                      | 1                       |
| debugPrototype1      | 3               | 0                      | 0                       |
| debugPrototype2      | 2               | 0                      | 0                       |

Table 1 shows the analysis results of 4 Vue applications. As can be seen from the data in the table, our tool detected one vulnerability in AccVulPrototype1, AccVulPrototype2, which indicates that our tool detected all vulnerabilities and did not occur Missed reports and false positives, and no false positives occurred in the non-vulnerable version of the application. It can be seen that our detection method has a good accuracy.

5. Conclusion
This article is based on previous work. Based on the characteristics of Vue application, this paper improves the relevant algorithms and selects a more suitable parser and solver to implement our tool. Compared to existing tools, this paper improves the context-free grammar constructor and link extraction algorithm to adapt to VUE. In applications developed in the PHP language, the HTML code is displayed in the code segment, and the Vue application needs to parse the template string to get the real HTML code. This article first parses the template string into an AST, and then converts the AST into a DOM node to obtain HTML code. And in the process of conversion, the value of the variable is extracted from the javascript code, and it is even more accurate to extract the link.

Our work has basically achieved the research goals and achieved some results, but at the same time also found some new problems and areas that need to be improved from the work of this article. This article does not deal with the object-oriented features of the ES6 standard, which prevents us from parsing related Vue pages. We hope to add support for more features of the ES6 standard including object-oriented features in future work, making the analysis more accurate.

6. Acknowledgments
This work is supported by the National Key Research and Development Program of China (No. 2017YFB1400805). Sun Jinan is the corresponding author.

7. References
[1] D. Melski and T. Reps. Interconvertibility of Set Constraints and Context-Free Language Reachability. In Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation-on, 1997.
[2] W. G. J. Halfond and A. Orso. Automated identification of parameter mismatches in Web applications. In Proceedings of the Symposium on Foundations of software engineering, 2008.
[3] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 337–340, 2008.
[4] Vallee-Rai R, Hendren L J. Jimple: Simplifying Java Bytecode for Analysis and Transformations. Sable Technical Report, School of Computer Science, McGill University, Montreal, Canada, 1998.
[5] Vallee-Rai R, Co P, Gagnon E, et al. Soot-A Java bytecode optimization framework. In Proceedings of the 20th Annual CASCON Conference. New York, USA: ACM, 2010.214–224.
[6] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: Effective Taint Analysis of Web Applications. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 87–97, 2009.

[7] Sun F, Liang X, Su Z. Static Detection of Access Control Vulnerabilities in Web Applications[J]. Shanxi Architecture, 2011, 2:6392-6396.

[8] Gauthier F, Merlo E. Fast Detection of Access Control Vulnerabilities in PHP Applications[C]// Reverse Engineering (WCRE), 2012 19th Working Conference on. IEEE Computer Society, 2012.

[9] Monshizadeh M, Naldurg P, Venkatakrishnan V N. MACE: Detecting Privilege Escalation Vulnerabilities in Web Applications[C]// Acm Sigsac Conference on Computer & Communications Security. 2014.

[10] Ivan Bocic, Tevfik Bultan. Finding access control bugs in web applications with CanCheck. ASE 2016: 155-166

[11] XIA Zhijian, PENG Guojun, HU Hongfu. Detection of access control vulnerabilities in Web applications based on privilege verification graph. Computer Engineering and Applications, 2018, 54(12): 63-68.

[12] Touseef P, Alam K A, Jamil A, et al. Analysis of Automated Web Application Security Vulnerabilities Testing[C]//Proceedings of the 3rd International Conference on Future Networks and Distributed Systems. 2019: 1-8.

[13] Zheng L, Yuan H, Peng X, et al. Research and implementation of Web application system vulnerability location technology[C]//The International Conference on Cyber Security Intelligence and Analytics. Springer, Cham, 2019: 937-944.