**Detecting SQL Injection and Cross Site Scripting Attacks with Static and Taint Analysis**

CSCE 526

Capstone Project

2d Lt Marvin Newlin

Dr. Okolica

20 Nov 18

**Abstract**

*Web applications are heavily used in every facet of life on the internet today. However, a large number of SQL Injection and Cross-Site Scripting vulnerabilities exist in web applications. These vulnerabilities pose a risk to users and companies alike with the threat of stolen or damaged data and compromised personal information. In this paper we discuss the background of SQL Injection and Cross-Site Scripting and the research that has been done on detecting these vulnerabilities with static analysis and taint analysis techniques. We also discuss the characteristics of a tool that would improve the ability to detect these types of vulnerabilities.*

**Introduction**

Web applications used everywhere in the world today. From e-commerce to banking web apps are utilized in every facet of life today. Most of the web applications that we use every day like Amazon, banking applications, and others involve interacting with a web page which interacts with a database behind the scenes. Given this high level of usage, one would think that web applications would be inherently secure. However, this is not the case. Since the mid-2000s, there has been a class of vulnerability known as command injection. The two most common forms of command injection are SQL Injection (SQLI) and its younger cousin Cross-Site Scripting (XSS) [1].

These vulnerabilities come in many different forms and have varying levels of maliciousness. However, due to the inherent security risk that these vulnerabilities present, they have been ever present on the Open Web Application Security Project (OWASP) Top 10 vulnerability list for several years, including the most recent edition in 2017 [2].

Given the extent to which these vulnerabilities exist and the problems that they can cause, SQLI and XSS have been heavily studied. One area of research on these vulnerabilities is the area of formal analysis. Formal analysis of these vulnerabilities involves usually static analysis along with some form of dynamic analysis to analyze the flow of information through the application to spot vulnerabilities. The upside to this form of analysis on web applications is that it allows for finding these vulnerabilities in code without executing the code which in a development environment can mean savings in time and money [3]. Additionally, this form of analysis can be performed on existing code to identify vulnerabilities without requiring modification to the code which also saves time and money [4].

In this paper, we explore a few of the most well-known approaches for formal analysis of SQLI and XSS in web applications and present some avenues for future work.

**Background**

The most prevalent command injection vulnerabilities generally fall into two categories: SQL injection and Cross-Site Scripting attacks [5], [6], [3]. According to the Common Weakness Enumeration (CWE) list, SQLI is an instance where software (think web application) generates a SQL query that contains user input. If the input contains certain elements of a SQL query, then a SQL injection attack can occur [7]. Outside influences should not be allowed to modify code and if they do modify it, problems will arise such as loss of confidentiality or integrity of data [2].

XSS attacks are related to SQLI attacks in that they are a vulnerability in which outside influences can affect execution of code, particularly from JavaScript. According to the CWE, XSS attacks occur generates a request for web data and that web page either contains a malicious script or allows execution of remote code [6]. There are three categories of XSS each with different characteristics.

* Reflected XSS Attack: The server reads data directly from the HTTP request and reflects it back into the HTTP response. This happens when an attacker causes a user to send malicious code to a vulnerable web program which is then reflected back to the user for an exploit [6].
* Stored (Persistent) XSS Attack: The attacker stores malicious code or data in a database, server log, or other trusted data source. This code is then executed as part of the dynamic content of a webpage after a legitimate request at a later time [6].
* DOM Based XSS: The client injects the XSS into the page. This usually occurs when a trusted script (i.e. a webpage) is sent from server to client and the user then sends data into the web page which is injected into the dynamic content. This injection of user supplied data is what allows a DOM based attack to occur [6].

Formal and particularly static analysis of SQLI and XSS involves discussing the different types of analysis that can be performed. They are: flow, context, and path. Each of these can be either sensitive or insensitive.

*Flow Sensitive*: This type of analysis involves generating a control flow graph (CFG) of the source code. The CFG is then used to find relationships between where data is defined and where it is used. Conversely, flow insensitive analysis does not utilize a CFG and instead ignores it and looks directly at the data. As such, flow sensitive analysis has more time overhead but is more precise [3].

*Context Sensitive*: This form of analysis allows for distinguishing scope of variables and can distinguish between different function calls within code. Conversely, context insensitive analysis cannot do this, so it is generally less precise than context sensitive analysis [3], [8].

*Path Sensitive*: This type of analysis analyzes the data as it flows through every possible valid path within the program. Path insensitive analysis looks at all possible path, valid and invalid. Path insensitive analysis in more time consuming and can also lead to path explosion in a program that contains a lot of branches, but is more precise in its results [3], [9].

Detecting and preventing SQLI and XSS attacks can occur on either the server side or the user side. Most static analysis approaches to dealing with XSS and SQLI vulnerabilities approach the problem from the server side because that allows them to deal with the code itself. Client side approaches do exist and we explore one below in [10]. However, the client-side approaches are harder to implement and usually require extensive modification of an existing browser.

**Literature Review**

Detecting SQLI and XSS attacks has been a heavily researched topic. With regards to formal analysis of these vulnerabilities, there are a handful of original papers on the subject with many others through the years extending on the topic.

In 2006, Wasserman and Su developed SQLCHECK, a program that could detect SQL Injection vulnerabilities in PHP code. In addition to their tool, they also were the first authors to develop formal definitions for both web applications and SQL injection [1]. Their tool was a server-side application that utilized a context-free grammar and special delimiters to evaluate string input and SQL queries for syntax that would lead to a SQL injection attack. They tested their tool on a small number of programs that had known vulnerabilities and were able to detect all of them, leading them to claim that their tool had no false positives or false negatives. However, they do acknowledge that the tool would probably produce some on programs that were in use in the world and not test programs [1].

One of the first effective methods of detecting and preventing XSS attacks was researched by Vogt et al. in “Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis” [10]. They explored the prevention of reflected XSS attacks through a combination of data tainting and static analysis. Their methodology involved using data tainting to track direct access of sensitive values (i.e. cookies) within the browser and combined it with static analysis in order to detect indirect accesses of sensitive data [10]. Overall, the tool they developed was effective in preventing reflected XSS attacks, but did generate some false positive warnings due to the method of tainting that they used [10].

Jovanovic et al. developed Pixy, the first open source tool for detecting XSS vulnerabilities in PHP code [3]. This tool accomplished this static analysis through a flow-sensitive and context-sensitive data flow analysis for PHP [11]. The data flow analysis allowed them to determine the possibility of tainted data reaching sensitive areas of code (i.e. cookies again) without that tainted data having been properly validated [11]. The results of Pixy were promising, but one issue with Pixy is the high false positive rate it produced which was around 50% [3].

In 2008, Wasserman and Su, presented a method of static analysis for detecting XSS vulnerabilities that focuses on input validation [12]. Using that static analysis model, they developed an algorithm that checked for untrusted script that could be used to generate an XSS attack [12]. Some of the limitations of their tool were that it could not detect DOM-Based XSS attacks, and additionally, suffered from a high false positive rate as well. [12]

Lam et al. discussed securing web applications with static and dynamic information flow tracking. This concept of information flow tracking they discussed was useful for preventing both SQLI and XSS attacks [13]. They developed a high-level query language called Program Query Language (PQL). This language allowed them to represent a given information flow as a an excerpt of java code [13]. This allowed them to conduct context-sensitive pointer analysis on their transformed code which afforded better identification of SQLI and XSS vulnerabilities [13].

More recently, in 2013, Zheng and Zhang developed a method for detecting Remote Code Execution (RCE) vulnerabilities (a more sinister version of stored XSS attacks). They proposed a method of path and context sensitive analysis along with a constraint solving algorithm that addressed several problems that exist in static analysis of web applications [14]. The problems they addressed were static analysis of strings, difficulty solving string and non-string constraints, and difficulty modeling RCE vulnerabilities [14]. Their approach generated good results with a lower false positive rate than most static analysis tools of 21% [14]. Additionally, in the programs they tested they had zero false negatives, detecting the 13 known vulnerabilities in their test code and discovered 8 more unknown RCE vulnerabilities [14].

In 2016, Steinhauser and Gauthier discussed detecting context sensitive XSS vulnerabilities in legacy web applications [4]. Context sensitive XSS vulnerabilities occur when a discrepancy exists between user input that has been sanitized (cleaned to supposedly render any malicious code moot) and output context (i.e. JavaScript input to HTML output) [4]. They developed a tool called JSPChecker that analyzed the code of legacy web applications for this type of vulnerability and was additionally able to handle syntax errors in HTML input while performing the checks [4]. The makeup of the tool was a series of algorithms that utilized data flow analysis to track data through the sanitization process, then statically analyzed the input, and finally generated an HTML document which was then used to check for the vulnerabilities via mismatches [4]. The results of the tool were promising in that they were successful in detecting the vulnerabilities with a low number of false positives. However, due to the makeup of the tool, it induces a performance hit when run [4].

The table below summarizes the type of approach used by each of the previously mentioned authors as well as the user side that their solution is implemented on.

Table 1: Classification of Approaches and User Side

|  |  |  |
| --- | --- | --- |
| **Author** | **Approach Type** | **User Side** |
| Vogt *et al.*[10] | Flow Sensitive | Client Side |
| Jovanovic *et al.* [11] | Flow sensitive Context Sensitive | Server Side |
| Wasserman and Su [1], [12] | N/A | Server Side |
| Lam *et al*. [13] | Flow Insensitive Context Sensitive | Server Side |
| Zheng and Zhang [14] | Path Sensitive Context Sensitive | Server Side |
| Steinhauser and Gauthier [4] | Context Sensitive | Server Side |

**Research Methodology**

Given the high propensity of static analysis to have false positives, an approach should be developed that is as precise as possible. Static analysis alone cannot identify all vulnerabilities precisely so it will need to be a hybrid analysis approach of both static and some form of dynamic string and taint analysis [14], .

One method of approach to help with this problem is to aggregate the approaches of the research discussed in the previous section. An approach that is context sensitive, path insensitive, and flow insensitive would lend itself to results that are more precise [3], [8], [9]. Although this type of approach is more time consuming, given the advance in computer technology in the last decade, the time element should not be as large of a factor as it may have been several years ago. Additionally, this type of approach is required if we want to achieve a good level of precision.

To implements this approach will require several steps to be completed. The first step for the context sensitive element, we would have to develop a method for sanitizing user provided strings before allowing them to propagate through the system, much like the approach used in JSPChecker [4]. Next, we will need to develop constraints for both strings and non-strings that will allow for mathematically determining whether a string is valid or not. While it is difficult to implement these two types of constraints together, it is a vital step in detecting and preventing these vulnerabilities [14]. Additionally, to implement this step will also require developing a Context-Free Grammar for analyzing the strings, like the approaches used in [1], [12], and [13]. Additionally, we will want to perform a taint analysis on the code to determine where the spots are that might be vulnerable to SQLI or XSS attack [8], [10], [11], [15]. From there, we would turn the Context-Free Grammar into a Finite State Machine in an approach similar to [12] and [13], that would allow us to deterministically evaluate the input to check for malicious values. From there, we would be able to produce this as a result if it shows up as malicious.

**Testing**

The most common method for testing a tool for detecting these types of vulnerabilities is to evaluate the tool on a set of programs in which the exact XSS and SQLI vulnerabilities are well known in order to evaluate for false positives and negatives [1], [8], [10], [12]. From there, we would analyze the results of the program against the results of the program we tested on to determine our false positive rate, false negative rate, and vulnerability detection rate. Additionally, we would want to analyze the runtime of our tool to get an idea of its efficiency.

**Conclusion**

The web application vulnerabilities of SQL Injection and Cross-Site Scripting are dangerous and common vulnerabilities in web applications today. One common method that is used for detecting these vulnerabilities is static analysis combined with taint analysis that allows for detection of the vulnerabilities without the requirement of executing the code. In this paper, we have explored the background of SQL Injection and XSS. We have also discussed the different sub-methods of static analysis that exist and have been utilized. We have discussed and analyzed some of the most common research on these vulnerabilities and the tools that they have developed for their detection. Finally, we have discussed what the characteristics of a new tool for detection of SQL and XSS vulnerabilities would contain and the advantages of utilizing a tool with these characteristics. In the future, analysis for detecting these vulnerabilities can be improved by faster string processing and lower false positive rates.

**References**

[1] Z. Su and G. Wassermann, “The essence of command injection attacks in web applications,” *ACM SIGPLAN Not.*, vol. 41, no. 1, pp. 372–382, 2006.

[2] Open Web Application Security Project, “OWASP Top 10-2017,” 2017.

[3] M. K. Gupta, M. C. Govil, and G. Singh, “Static analysis approaches to detect SQL injection and cross site scripting vulnerabilities in web applications: A survey,” in *International Conference on Recent Advances and Innovations in Engineering (ICRAIE-2014)*, 2014, pp. 1–5.

[4] A. Steinhauser and F. Gauthier, “JSPChecker: Static Detection of Context-Sensitive Cross-Site Scripting Flaws in Legacy Web Applications,” in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security - PLAS’16*, 2016, pp. 57–68.

[5] “CWE - CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection’) (3.1),” 2018. [Online]. Available: https://cwe.mitre.org/data/definitions/77.html. [Accessed: 26-Oct-2018].

[6] “CWE - CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting’) (3.1),” 2018. [Online]. Available: https://cwe.mitre.org/data/definitions/79.html. [Accessed: 26-Oct-2018].

[7] “CWE - CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection’) (3.1),” 2018. [Online]. Available: https://cwe.mitre.org/data/definitions/89.html. [Accessed: 26-Oct-2018].

[8] N. Jovanovic, C. Kruegel, and E. Kirda, “Static analysis for detecting taint-style vulnerabilities in web applications,” *J. Comput. Secur.*, vol. 18, no. 5, pp. 861–907, 2010.

[9] L. K. Shar, H. Beng Kuan Tan, and L. C. Briand, “Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis,” in *Proceedings - International Conference on Software Engineering*, 2013, pp. 642–651.

[10] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, “Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis,” 2007.

[11] N. Jovanovic, C. Kruegel, and E. Kirda, “Pixy: a static analysis tool for detecting Web application vulnerabilities,” in *2006 IEEE Symposium on Security and Privacy (S&P’06)*, 2006, p. 6 pp.-263.

[12] G. Wassermann and Z. Su, “Static detection of cross-site scripting vulnerabilities,” in *Proceedings of the 13th international conference on Software engineering - ICSE ’08*, 2008, p. 171.

[13] M. S. Lam, M. Martin, B. Livshits, and J. Whaley, “Securing web applications with static and dynamic information flow tracking,” in *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation - PEPM ’08*, 2008, pp. 3–12.

[14] Y. Zheng and X. Zhang, “Path sensitive static analysis of web applications for remote code execution vulnerability detection,” in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 652–661.

[15] A. Kiezun, P. J. Guo, K. Jayaraman, and M. D. Ernst, “Automatic creation of SQL injection and cross-site scripting attacks,” in *Proceedings - International Conference on Software Engineering*, 2009, pp. 199–209.