CSCE 526

Background & Lit Review

2d Lt Marvin Newlin

Dr. Okolica

30 Oct 18

**Background**

The most prevalent command injection vulnerabilities generally fall into two categories: SQL injection (SQLI) and Cross-Site Scripting (XSS) attacks [1], [2], [3]. According to the Common Weakness Enumeration (CWE) list, SQLI is an instance where “The software constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component” [3]. The problems with this command modification/creation are obvious to those with an understanding of security principles. 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 [4].

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 [2]. 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 in the HTTP response. Reflected XSS exploits occur when an attacker causes a victim to supply dangerous content to a vulnerable web application, which is then reflected back to the victim and executed by the web browser” [2].
* Stored (Persistent) XSS Attack: “The application stores dangerous data in a database, message forum, visitor log, or other trusted data store. At a later time, the dangerous data is subsequently read back into the application and included in dynamic content” [2].
* DOM Based XSS: “In DOM-based XSS, the client performs the injection of XSS into the page; in the other types, the server performs the injection. DOM-based XSS generally involves server-controlled, trusted script that is sent to the client, such as JavaScript that performs sanity checks on a form before the user submits it. If the server-supplied script processes user-supplied data and then injects it back into the web page (such as with dynamic HTML), then DOM-based XSS is possible” [2].

SQLI and XSS attacks have been around since the mid-2000s, but even though they have been extensively studied, they still exist today [5].

**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.

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” [6]. 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 [6]. 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 [6].

Jovanovic et al. developed the first open source tool for detecting XSS vulnerabilities in PHP code [5]. This tool accomplished this static analysis through a flow-insensitive and context-sensitive data flow analysis for PHP [7]. 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 [7]. The results of Pixy were promising, but one issue with Pixy is the high false positive rate it produced which was around 50% [5].

In 2008, Wasserman and Su, presented a method of static analysis for detecting XSS vulnerabilities that focuses on input validation [8]. Using that static analysis model, they developed an algorithm that checked for untrusted script that could be used to generate an XSS attack [8]. 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. [8]

Livshits 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 [9]. 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 [9]. This allowed them to conduct context-sensitive pointer analysis on their transformed code which afforded better identification of SQLI and XSS vulnerabilities [9].

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 [10]. The problems they addressed were static analysis of strings, difficulty solving string and non-string constraints, and difficulty modeling RCE vulnerabilities [10]. Their approach generated good results with a lower false positive rate than most static analysis tools of 21% [10]. 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 [10].

In 2016, Steinhauser and Gauthier discussed detecting context sensitive XSS vulnerabilities in legacy web applications [11]. 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) [11]. 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 [11]. 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 [11]. 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 [11].

References

[1] “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].

[2] “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].

[3] “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].

[4] OWASP, “OWASP Top 10-2017,” 2017.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.