Detecting and Mitigating SQL Injection Attacks

By XXXXXX

March 15, 2012

The results of SQL injection attacks have headline-grabbing notoriety. Perform an internet search for "SQL injection attacks" and one of the top responses you are likely to receive will be a corresponding Wikipedia entry [1] with a list of note-worthy and successful exploits. The list found at the time of this writing spans a period between 2005 – 2011 and involves some of the largest corporations in the world. Next, visit the National Institute of Standards and Technology National Vulnerabilities Database website and perform a CVE (Common Vulnerabilities and Exposures) search for "SQL injection": as of this writing, results number over 5,400 [2].

SQL injection attacks are neither new nor difficult to understand conceptually, but are consistently exploited even today, though numerous approaches have developed over time to mitigate or eliminate them from production web applications. Several of these approaches will be presented in this review and include the unique or common aspects supporting detection and mitigation of this vulnerability.

SQL injection attacks (SQLIA) are a specific type of user-generated input attack, typically made against web-based applications through form submissions. Against an unprotected system, a malicious user or program can submit specially crafted form data in order to achieve unintended results; for example, instead of processing a typical search query using a login username, the malicious data instead marries with the underlying SQL query string to send an altogether different request to the target database. Without proper analysis and processing of user-generated input, SQLIAs have the potential of returning enormous amounts of data to an unintended audience at great expense to governments, businesses, and the general public.

While other forms of input attacks exist (XSS (cross-site scripting) and buffer overflow being two common examples [3]) SQLIAs remain as one of the most prolific and difficult-to-detect vulnerabilities facing developers today [4]. This review analyzes a number of studies that propose solutions to form-based attacks. Other forms of data delivery, such as web services, are not covered here but are still a potential vector for malicious delivery. In many cases, the solutions reviewed in this paper could accommodate various methods for data submission besides standard form-based delivery. None of the

solutions presented take a client-side approach to detection or mitigation, in part, because client-side validation is so easily defeated [5].

An issue raised by Smith, et. al. [6] is simply that database testing coverage is inadequate when it comes to SQL statement validation. Although their research aims to target SQL injection attacks specifically, much of the research they present addresses SQL testing coverage in more general terms. They propose the need to include two validation metrics to a site's test suite to ensure adequate database testing coverage: target statement coverage and target variable coverage. The authors add methods to source code that trigger checking and logging functions as a means to ensure coverage of SQL statement calls, a procedure common to other solutions reviewed in this paper [3], [8], [9]. It is only by the inclusion of a SQL statement blacklist that their testing assumes a security position. The structure and means of utilizing the blacklist are addressed tangentially and do not appear to be of specific concern to their solution, but we can assume the methods used to trigger checking and logging can also compare submitted data against the blacklist to determine if a match exists and, therefore, discovery of malicious data submitted by the user [9], [10].

Deng, et. al. [7] also realized the importance of testing web application databases, but their work is unique from others in this review since they concentrate on the state prior to and immediately after a transaction is made against the target database. They leverage the AGENDA toolset from previous work, a collection of utilities specific to testing databases. By extending the AGENDA toolset, they have devised a process where URLs are extracted from an application's source code and used to map the application; a particular path leading from a defined starting point to an end node constitutes a single test case. For those test cases requiring user input, AGENDA creates the input and commits transactions against the database where the pre- and post- states are compared for inconsistencies; however, actual output from the submitted data must be performed manually. This design feature may present a security testing issue since the toolset would not flag a query that returns, for example, the entire Accounts table in a Customer database. As with Smith [6], Deng's focus is not so much about security as it is about test coverage, but could incorporate security checking as part of its normal validation processes.

Huang [4] presents an automated testing framework with the unique component of a "Self-Learning Knowledge Base Model", part of their WAVES (Web Application Vulnerability and Error Scanner) security system. This security model is designed to learn expected form input values dynamically, using a complex algorithm of field naming, proximity searches and text adjacency attributes culminating in a deductive guess for any fields not tracked in the current list of known values. Ideally, form design will use both syntactically and semantically descriptive field value names (they use the example of "strUserName"; it is clear by convention that this field expects a string for the user name value).

WAVES uses black-box testing instead of white-box, arguing that white-box, static testing and analysis is not representative of today's dynamic web application architecture. A crawler loads a site's pages into memory and searches for form inputs, outputting XML representations of these HTML pages. WAVES then uses this XML to generate input data that is submitted to the application under test. Here, it's unclear exactly how this submission is achieved – one example used by the authors is flawed in that they demonstrate submission of a user name and password with the HTTP GET method. It is highly unlikely that any developer would use this method in practice since the GET method appends user input to the target URL as cleartext. If WAVES is dependent upon finding and executing GET requests with appended test data, then its relevancy as a security testing framework is probably questionable since information such as passwords should not be submitted in a URL query string [5].

Contrasting with Huang is another study conducted by Huang, et. al. [8] that claims black-box testing is insufficient for identifying all potential bugs and inadequate if the desire is to also provide immediate security against any discovered vulnerabilities. Their solution aims to detect and control information flow in an application, similar to the Bell-LaPadula system where sensitivity is defined by security labels ranging from less restrictive to more restrictive. Application variables are given the strictest security label as their initial value, but should be allowed explicit declassification when appropriate, typically resulting after a runtime data sanitization or validation check. Their WebSSARI (Web application Security by Static Analysis and Runtime Inspection) tool implements static analysis to

determine where validation tests should be inserted in the source code and also provides the PHP-extension libraries used during runtime for dynamic analysis and security execution.

Halfrond and Orso [9] present AMNESIA, a white-box tool specifically tailored to detect and mitigate SQLIAs in a simple and straightforward manner. Like others before them [8], [3] they use a measure of both static and runtime analysis in their solution. Key to the success of their proposed solution is identifying areas within the application source where potentially vulnerable statements occur; how this discovery should be conducted is not addressed in their study but important to call out since missing an entry point that allows a SQLIA negates the benefit of this tool (or indeed any tool discussed in this review). Once these entry points are identified, code is added to the application source that identifies from where it is calling (by passing an ID parameter) and invokes the AMNESIA tool, a library that takes a static list of SQL query statement models permissable for the now-identified caller. User-generated input is parsed into tokens and compared against the allowable models: if they match, the submission is allowed to proceed, otherwise it fails. This solution is simple in its implementation, but suffers from a common issue where intimate knowledge of a potential attack must be known in order to build the models against which validation checks are made.

Buehrer, Weide, and Sivilotti [10] observe that an identifying characteristic of SQL injection attacks is that they, by necessity, alter the expected structure of a SQL statement passed to the server for processing. They conclude, similar to Halfrond and Orso [9], that detection of such attacks can then be performed by parsing the submitted data into tokens. Employing a parse tree validator against the resulting tokens results in a pass/fail response for the data. They describe common attacks used to extract data from databases, but success is dependent upon the parse tree validator knowing about all permutations of potential attacks, again a shared weakness with [9]. Implementation is a Java-based library using a single call to SQLGuard() at a point just prior to the SQL query string submission – if the user-supplied data parses according to expectations, then the query is allowed to pass.

Despite the research reviewed in this paper and many other similar studies aimed at detecting and mitigating SQL injection attacks, this threat continues to live on in many applications' code. While much

of the research included here is dated by technology standards, it is worth mentioning that SQL syntax has likely not evolved much over this same time, nor has the mechanism by which SQL query statements are delivered to a server for processing; most user-supplied data is done through form submissions from a web page. Why, then, do we continue to see these exposures?

From the research reviewed, solutions can range from simple [9], [10] to complex [8] and still claim that, if done properly and to their specifications, SQLIAs can be nearly, if not completely, eliminated from the web application's infrastructure. There is the catch: "…if done properly and to their specifications", or if done at all. I believe, historically, security has been victim of an unfortunate belief that, somehow, it just gets done. Without an awareness of the threat opportunities and exposures prevalent in the average developer's lexicon of programming tools, the solutions presented in this review are worthless. It is not good enough to have an elegant and complex solution if it is too difficult to implement. It is not adequate enough to have a simple and easy solution if too much accountability is given to the developer for configuration and threat signature creation. Developers abide by programming design patterns, perhaps the creation of security design patterns would benefit the profession by standardizing the way individuals and teams view security in code.

Anley [11] summarizes this review best when identifying the true culprit to SQLIA vulnerabilities: input validation is complex, cumbersome, and adds little to the functionality of a piece software. From my experience, I would also add that security is an enigmatic subject to most developers and little time is spent learning how to write secure application code. Perhaps the expectation is put upon the code libraries themselves for protection against vulnerabilities. Given the frenetic pace of software development, secure code reviews and best practices may simply be ignored, forgotten, or simply never considered when laying out delivery schedules.

The literature reviewed here seeks to detect and mitigate SQL injection attacks. Claims were made for both a black-box and white-box approach to testing. Depending on the particular solution, these claims were validated by the support and processes provided, but a mixture of both appears to be the best method to ensure adequate coverage and identification of attack opportunities in the application source

code. All solutions support the concept of server-side data validation as a means to sanitize and evaluate user-supplied data.

REFERENCES

[1] Wikipedia: SQL injection [Online]. Available: http://en.wikipedia.org/wiki/SQL_injection

[2] National Vulnerability Database (NVD) Search Vulnerabilities [Online]. Available: http://web.nvd.nist.gov/view/vuln/search

[3] Zhendong Su and Gary Wassermann, "The essence of command injection attacks in web applications," in *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2006, pp. 372-382.

[4] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai, "Web Application Security Assessment by Fault Injection and Behavior Monitoring," In *Proceedings of the 12th international conference on World Wide Web*, 2003, pp. 148-159.

[5] Mark Curphey, et. al. (2002). *A Guide to Building Secure Web Applications*. (v1.1.1) [Online]. Available: ftp://sbin.org/pub/doc/books/web/OWASPGuideV1.1.1.pdf

[6] Ben Smith, Yonghee Shin, and Laurie Williams, "Proposing SQL statement coverage metrics," in *Proceedings of the fourth international workshop on Software engineering for secure systems*, 2008, pp. 49-56.

[7] Yuetang Deng, Phyllis Frankl, and Jiong Wang. (2004, September). Testing web database applications. *SIGSOFT Softw. Eng. Notes. 29 (5)*, pp. 1-10.

[8] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo, "Securing Web Application Code by Static Analysis and Runtime Protection," in *Proceedings of the 13th international conference on World Wide Web*, 2004, pp. 40-52.

[9] William G. J. Halfond and Allesandro Orso, "Preventing SQL injection attacks using AMNESIA," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 795-798.

[10] Gregory Buehrer, Bruce W. Weide, and Paolo A. G. Sivilotti, "Using parse tree validation to prevent SQL injection attacks," in *Proceedings of the 5th international workshop on Software engineering and middleware*, 2005, pp. 106-113.

[11] Chris Anley. (2002). *Advanced SQL Injection In SQL Server Applications.* [Online]. Available: http://vmg.pp.ua/books/%D0%9A%D0%BE%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D1%8B%D0%98%D1%81%D0%B5%D1%82%D0%B8/hack/engl/sql%20injection/Hacking%20-%20Advanced%20SQL%20Injection.pdf