XXXXXXXXXX

SQL “Rejection”

Introduction

SQL injections are one of the most formidable web application vulnerabilities to plague data driven web applications. For those entities unfortunate enough to be victims of such an attack, the effects can mean compromised data and loss of reputation. All of which can result in substantial financial loss for the entity. The Open Web Application Security Project (OWASP) has consistently rated SQL injections as one of the top 10 web application vulnerabilities [1]. In 2013 SQL injections took the number one position on the OWASP top ten list of web application vulnerabilities [2].

Modern web applications typically generate dynamic content by soliciting input from the user along with the interaction of this input with a back end database. The users input is usually submitted to the web application through various html forms, processed by some sort of server side scripting language which then creates and sends a query string to a back end database to perform any number of SQL operations such as SELECT, INSERT, UPDATE, DELETE, etc. If the input is not properly sanitized by the application code, it can be used to create a query string that will execute database operations that were not intended by the application and its developers. Such illicit database operations could expose sensitive and confidential information such as bank accounts and credit card numbers to unauthorized changes, theft, and destruction.

A basic SQL injection requires the applications query string to be manipulated in such a way that the database executes an SQL statement with unintended consequences. For example, a web form could allow a user to login by entering their username and password. The username and password would be stored in the variables “uname” and “pwd”. These variables would be included in a query string such as “SELECT * FROM users where username = ‘uname’ and password = ‘pwd’”. If these variables are not properly sanitized by removing or escaping unwanted characters such as single and double quotes the query string can be passed a parameter that contains additional SQL commands and or operators. The following is an example in PHP of a malformed query string [3]:

```
$Uname = ‘vespa’;

$Pwd=” ‘ OR ‘ ‘=’ “;

$bad_query = “SELECT * from users where user=’uname’ and password = ‘Pwd’”;

Mysql_query($bad_query);
```

The query that executes will be: SELECT * from users where user=’vespa’ and password= ‘ ‘ or ‘ ‘= ‘ ‘

This statement will always evaluate to TRUE and thus allow login without a valid password. This same technique could be employed to append another, entirely different query by including a semi-colon “;” in the malicious string followed by the additional query.

The sample code demonstrates how easy it is to construct an SQL injection exploitation code. This coupled with the increasing frequency and sophistication of SQL injection attacks has and continues to prompt researchers to develop effective methods to combat this technological menace.

SQL injection Literature Review

Researchers have developed many interesting and novel methods and strategies for detecting and preventing SQL injection attacks. These mitigation efforts can range from simply sanitizing user input to sophisticated detection and prevention algorithms.

The Wasserman and Zhendong [4] paper is one of the first and more definitive papers on the subject of SQL injection attacks. Using both SQL statements and application code the authors provide many examples of how an SQL injection attack is crafted. Additionally they suggest the conventional mitigation strategies such validating and sanitizing user input, but they go a step further and present a sophisticated algorithm for detecting and preventing improperly formed SQL statements.

Wasserman and Zhendong developed a parser generator program called SQLCHECK that parses the query string to detect substrings that could be an attempt to modify the syntactical structure of the remaining query. The SQLCHECK program can also be used to detect injections in the form of cross site scripting, XPath injection, and shell injection attacks [4].

Central to the SQLCHECK algorithm is the parsing of the query strings WHERE clause into a parse tree of substrings. Each sub string exists as a node in the parse tree and may or may not have leaf objects. In a legitimate query the descendent leaves should consist of the entire input substring. An illegitimate query does pose this distinction. According o the authors “This distinction is common to all examples of legitimate vs. malicious queries that we have seen”. [4]

Similar to the Wasserman and Zhendong parse tree algorithm, authors Gregory Buehrer, Bruce W. Weide, Paolo A. G. Sivilotti offer another method to analyze the SQL payload prior to its execution on the database system. By parsing the SQL payload into a tree structure the author’s technique can examine and compare the user imputed query string to the query string that would be intended by the developer for that particular type of query. Any deviation from the intended SQL string could be an indication of nefarious activity upon which the application code and the database system could react accordingly. [5]

Gregory Buehrer , Bruce W. Weide , Paolo A. G. Sivilotti developed a Java class called SQLGuard that parses and evaluates an SQL string into a parse tree in order to evaluate the syntactical grammar of the submitted query. The SQLGuard function generates a query string that includes the user submitted input. From this string a key is generated by the function. This key is pre and post appended to the query string and is then verified by the verify() function. The key is then removed from the query string. The string is parsed into two parse trees consisting of a tree of unpopulated tokens/nodes for the user input and another tree of these same nodes populated with user input. The two trees are then compared for matching structures [5]. Similar to the Wasserman and Zhendong parse tree algorithm, the two trees are compared to determine if the user input are only in leaf nodes. User input located in a non-leaf node would be an indication of a deviation from the intended syntactical structure of the query. The Wasserman Zhendong application, SQLCHECK, offers a few features that SQLGUARD does not. SQLCHECK can parse and analyze query input that contains text of SQL queries such as for websites that allows users to post/submit comments and or questions about SQL related matters. The SQLGUARD creators feel this is unnecessary as most websites have a different purpose, so they do not include this capability in their application.   Additionally SQLGUARD does not perform input inspection for cross site scripting, XPath injection, and shell injection attacks. Overall the SQLCHECK program appears to be better than SQLGUARD at detecting and preventing various forms of SQL injections.

The authors William G. J. Halfond, Alessandro Orso, Panagiotis Manolios have developed an innovative way to disarm SQL injection attacks that is automated, requires less development time, and results in little or no false positives. A method called positive tainting and syntax aware evaluation is used to detect SQL injection attacks. In contrast to traditional tainting methodologies, positive tainting identifies those elements of a query that are predetermined as acceptable or trusted as opposed to traditional tainting that identifies those elements that are unacceptable or untrusted [6].

One of the key differences of positive tainting as compared to negative tainting is the effect of completeness of the data to be examined. Analysis of incomplete input by negative tainting methods can lead to false negative situations by trusting input that should not be trusted. In contrast, positive tainting responds to incomplete data with false positives thereby not trusting input that should be trusted. A false positive is preferable to a false negative in that no SQL injection will go undetected. The author's state that their application can be tuned during testing or production to remediate false positives but before tuning the application needs to be configured to know what is considered trusted data sources. These trusted data sources would include hard coded query strings along with their variables and operators but could also include external sources such as files, external query fragments, server variables, and network connections [6].

At runtime these data sources are "tainted" before the query is executed in the database. It is imperative that the data sources are accurately tainted. The authors employ two methods of tainting the data by "tracking taint markings at a low level of granularity and precisely accounting for the effects hat operate on tainted data"[6]. Analyzing taint information at the character level is more accurate at determining effect of the query string. In addition, all relevant query strings need to be identified along with their effects.

In conjunction with accurately character tainting, is the concept of using syntax aware evaluation. This method parses the query string into a series of tokens. These string tokens are then checked to determine if they are SQL key words, operators, or literals and if to determine if they are trusted data.

The solutions presented thus far have been single application module solutions. The authors Davide Balzarotti , Marco Cova , Viktoria V. Felmetsger , Giovanni Vigna take a different approach to analyzing web application vulnerability. Typical web application vulnerability assessment examines only one application module pertaining to the input that is submitted to the application. The authors offer a multi-mode approach that examines the extended state and the intended state of the application in order to more fully understand the systematic implications of particular web application vulnerability [7].

The authors contend that the current web application security solutions proposed by the research community are subject to three limitations. These solutions are classified as first class solutions and second class solutions. A first class solution simply attempts to detect and block web application requests by analyzing the HTTP request and or the HTTP response. A web application firewall (WAF) such as Mod_security is one such first class solution [8]. A second class solution attempts to identify issues with the web applications implementation before the application is rolled out. One limitation of these solutions is that they are limited to one particular web application module and therefore not as effective at detecting vulnerabilities in web applications using more than one module. Another limitation is that these solutions do not consider the various interactions among multiple, different languages, and or databases. The final and most important limitation exhibited by these solutions is their inability to consider the intended workflow and or the extended state of the web application. Addressing this limitation is at the heart of the author's web application security program [7].

The authors program attempts to remedy these intended work flow and extended state issues by using a variety of analysis techniques to identity related vulnerabilities. In particular the author's application examines the various states of a web application both on the client side and server side. The authors refer to this concept as "state entity". Another component of their application is the analysis of the various web applications modules. They term this the "module view". The module view is the representation of the web application's intra-module and inter-module operations and the resulting changes in the applications extended states[7].

Data applications are increasingly gravitating toward persistent frameworks in order to interact with relational data and although these persistent frameworks support explicit queries, they do not perform validation of static object types or syntax at runtime. The authors William R. Cook, Siddhartha Rai's solution to this problem is called Safe Query Objects. Their solution employs various techniques such as object relational mapping and reflective metaprogramming in order to convert queries to statically typed objects. These objects/classes are then executed like a typical database query [9].

A safe query object is "just an object containing a Boolean method that can be used to filter a collection of candidate objects" [9]. SQL queries are established through object oriented classes and functions that are converted into code that calls standard DMBS interfaces.

Conclusion

Although SQL injection attacks are ever increasing in numbers and sophistication it appears that researchers are rising to the challenge of finding innovative and effective methods of securing web applications. Researchers have developed a variety of techniques for detecting SQL injections ranging from parse tree algorithms to more holistic approaches such as object relation mapping. Of the research papers reviewed it would appear that the safe objects program developed by William R. Cook, Siddhartha Rai would be the most effective at detecting and preventing SQL injections. The parse tree methodologies would perhaps be quite adequate for less complex websites. Of the parse tree methodologies the positive tainting approach presented by the others William G. J. Halfond , Alessandro Orso , Panagiotis Manolios seems to be superior to the negative tainting parse tree as it does not produce false negatives like that of negative parse tree algorithms.

Regardless of which approach is the best, web application security must be approached with a layered security model. This would include traditional security practices such as the implementation of a properly configured firewall, a hardened operating system, encryption technology, sanitization of user input, regular OS patching, etc. As part of a layered security approach, the methods researched by the authors would do very well to complement and improve the overall security posture of a web application.

References

1. "OWASP Top Ten Project". OWASP. Retrieved 2014-03-29
2. "OWASP Top Ten Project". OWASP. Retrieved 2014-03-29

3. "Mysql_real_escape_string". PHP.net. Retrieved 2014-03-30
4. Zhendong Su , Gary Wassermann, The essence of command injection attacks in web applications, *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p.372-382, January 11-13, 2006, Charleston, South Carolina, USA
5. Gregory Buehrer , Bruce W. Weide , Paolo A. G. Sivilotti, Using parse tree validation to prevent SQL injection attacks, *Proceedings of the 5th international workshop on Software engineering and middleware*, September 05-06, 2005, Lisbon, Portugal.
6. William G. J. Halfond , Alessandro Orso , Panagiotis Manolios, Using positive tainting and syntax-aware evaluation to counter SQL injection attacks, *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, November 05-11, 2006, Portland, Oregon, USA.
7. Davide Balzarotti , Marco Cova , Viktoria V. Felmetsger , Giovanni Vigna, Multi-module vulnerability analysis of web-based applications, Proceedings of the 14th ACM conference on Computer and communications security, November 02-October 31, 2007, Alexandria, Virginia, USA
8. HOWTOFORGE. Retrieved 2014-04-01
9. William R. Cook , Siddhartha Rai, Safe query objects: statically typed objects as remotely executable queries, *Proceedings of the 27th international conference on Software engineering*, May 15-21, 2005, St. Louis, MO, USA.