MS-ISAC TECHNICAL WHITE PAPER
May 2017

SQL Injection

Authored by: Stephanie Reetz, SOC Analyst

INTRODUCTION

Web applications are everywhere on the Internet. Almost everything you do online is done through a web application whether you know it or not. They come in the form of web-based email, forums, bulletin boards, bill payment, recruitment systems, health benefit and payroll systems. It is important to understand that these types of websites are all database driven. Databases are an essential element of web applications because they are able to store user preferences, personal identifiable information, and other sensitive user information Web applications interact with databases to dynamically build customized content for each user. The web application communicates with the database using Structured Query Language (SQL). SQL is a programming language for managing databases that allows you to read and manipulate data in MySQL, SQL Server, Access, Oracle, DB2, and other database systems. The relationship between the web application and the database is commonly abused by attackers through SQL injection. SQL injection is a type of injection attack in which SQL commands are supplied in user-input variables, such as a web form entry field, in an attempt to trick the web application into executing the attacker's code on the database.

SQL injection was one of the primary attack vectors responsible for many of 2011's high profile compromises including Sony Pictures, HBGary, and PBS. It was also responsible for the more recent Adobe data breach in which names, email addresses, and password hashes were stolen from one of their customer databases. SQL injection is a dangerous vulnerability that is easily detected and inexpensive to fix. This method of attack has been employed by hackers for over ten years yet it is still the most common attack vector in data breaches today

OVERVIEW

SQL injection ("Improper Neutralization of Special Elements Used in an SQL Command") is at the top of the most recent CWE/SANS Top 25 Most Dangerous Software Errors list and must be taken seriously. [1] SQL injection occurs when untrusted user-supplied data is entered into a web application and that data is then used to dynamically create a SQL query to be executed by the database server. Web applications generally use server-side scripting languages, such as ASP, JSP, PHP, and CGI, to construct string queries which are passed to the database as a single SQL statement. PHP and ASP applications tend to connect to database servers using older Application Programming Interfaces (API) that are by nature more readily exploited while J2EE and ASP.NET applications are not as likely to be so easily exploited. If a web application is vulnerable to SQL injection, then an attacker has the ability to influence the SQL that is used to communicate with the database. The implications of this are considerable. Databases often contain sensitive information; therefore, an attacker could compromise confidentiality by viewing tables. An attacker may also jeopardize integrity by changing or deleting database records using SQL injection. In other words, an attacker could modify the queries to disclose, destroy, corrupt, or otherwise change the underlying data. It may even be possible to login to a web application as another user with no knowledge of the password if non-validated SQL commands are used to verify usernames and passwords. If a user's level of authorization is stored in the database it may also be changed through SQL injection allowing them more permissions then they should possess. If SQL queries are used for authentication and authorization, an attacker could alter the logic of those queries and bypass the security controls set up by the admin. Web applications may also be vulnerable to second order SQL injection. A second order SQL injection attack occurs when user-supplied data is first stored in the database, then later retrieved and used as part of a vulnerable SQL query. This type of SQL injection vulnerability is more difficult to locate and exploit. Exploitation does not end when the database is compromised, in some cases an attacker may be able to escalate their privileges on the database server allowing them to execute operating system commands.

ATTACK SCENARIO
Consider a simple SQL injection vulnerability. The following code builds a SQL query by concatenating a string entered by the user with hard coded strings:

```
String query = "SELECT * FROM items WHERE owner = '" + userName + "'
                 AND itemName = '" + ItemName.Text + "'";
```

The intent of this query is to search for all items that match an item name entered by a user. In the example above, userName is the currently authenticated user and ItemName.Text is the input supplied by the user. Suppose a normal user with the username smith enters benefit in the web form. That value is extracted from the form and appended to the query as part of the SELECT condition. The executed query will then look similar to the following:

SELECT * FROM items WHERE owner = 'smith' AND itemName = 'benefit' However, because the query is constructed dynamically by concatenating a constant base query string and a user-supplied string, the query only behaves correctly if itemName does not contain a single quote (') character. If an attacker with the username smith enters the string:

anything' OR 'a'='a

The resulting query will be:

SELECT * FROM items WHERE owner = 'smith' AND itemName = 'anything' OR 'a' = 'a'

The addition of the OR 'a'='a' condition causes the WHERE clause to always evaluate to true. The query then becomes logically equivalent to the less selective query:

SELECT * FROM items

The simplified query allows the attacker to see all entries stored in the items table, eliminating the constraint that the query only returns items owned by the authenticated user. In this case the attacker has the ability to read information he should not be able to access.

Now assume that the attacker enters the following:

anything'; drop table items—

In this case, the following query is built by the script:

SELECT * FROM items WHERE owner = 'smith' AND itemName = 'anything'; drop table items--'

The semicolon (;) denotes the end of one query and the start of another. Many database servers allow multiple SQL statements separated by semicolons to be executed together. This allows an attacker to execute arbitrary commands against databases that permit multiple statements to be executed with one call. The double hyphen (--) indicates that the rest of the current line is a comment and should be ignored. If the modified code is syntactically correct, it will be executed by the server. When the database server processes these two queries, it will first select all records in items that match the value anything belonging to the user smith. Then the database server will drop, or remove, the entire items table.

BLIND SQL INJECTION

Another form of SQL injection is referred to as blind SQL injection. Normally, if the attacker were to inject SQL code that caused the web application to create an invalid SQL query, then the attacker should receive a syntax error message from the database server. However, the specific error code from the database should not be shared with the end user of an application. It may disclose information about the design of the database that could help an attacker. In an attempt to prevent SQL injection exploitation, some developers return a generic page rather than the error messages or other information from the database. This makes exploiting a potential SQL injection vulnerability more difficult, but not impossible. An attacker will know whether or not a query was valid based on the page returned. If the application is vulnerable and the query is valid, a certain page will be returned. However, if the query is invalid, a different page might be returned. Therefore, an attacker can still get information from the database by asking a series of true and false questions through injected SQL statements. The same page should be returned regardless of if an invalid SQL query was executed.

MITIGATION

The three main defense strategies against SQL injection are parameterized queries, stored procedures, and input validation. The first option is the use of parameterized queries. They require that all SQL code is first defined and then parameters are passed to the query. They are easier to write than dynamic queries and help prevent SQL injection by differentiating between the SQL code and the user-supplied data. This prevents an attacker from changing the design of a query by injecting his own SQL code. For example, if an attacker were to enter anything' OR '1' = '1 a parameterized query would search the database for an item that matched the string anything' OR '1' = '1 instead of inserting OR '1' = '1 into the query.

The second defense strategy, comparable to the first, is the use of stored procedures which prevent SQL injection as long as they do not include any unsafe dynamic SQL generation. Again, the SQL code must be defined first and then the parameters are passed. The difference between parameterized queries and stored procedures is that the SQL code for a stored procedure is defined and stored in the database itself, then called from the application. Applications can call and execute the stored procedures using the call SQL statement. The syntax and capabilities of stored procedures varies by the type of database. The following code will create a stored procedure that will execute the original query without dangerously concatenating user input with SQL code.

```
 CREATE PROCEDURE SearchItems
@userName varchar(50),
@userItem varchar(50)
AS
BEGIN
SELECT * FROM items
```

```
WHERE owner = @userName
AND itemName = @userItem;
END
GO
```

The query structure is already defined on the server before the query is executed and the conditions of the query are passed as parameters, therefore, the user-supplied input will not be mistaken as part of the SQL query. Like with parameterized queries, if an attacker were to submit the string anything' OR '1' = '1, it will be treated as user input, not SQL code. In other words, the query will look for an item with this name instead of executing unexpected SQL code. It is important to note that stored procedures do not prevent SQL injection in and of themselves. If a SQL query is built within a stored procedure by concatenating parameter values, like in the original vulnerable query example, it runs the same risk of SQL injection. An advantage to using this approach is that all database user accounts can be restricted to only access the stored procedures and therefore will not have the authority to run dynamic queries. Without the ability to run dynamic queries, SQL injection vulnerabilities are less likely to be present.

The third approach is to escape all user-supplied input before adding it to a query. When usersupplied input is escaped, special characters are replaced with characters the database should not confuse with SQL code written by the developer. The specific functions used for escaping usersupplied input vary by server-side scripting language. For example, in PHP the addslashes() function can be used to insert backslashes before the single quote ('), double quote ("), and backslash (\) characters as well as before the NULL byte. Note that addslashes() should not be used on strings that have already been escaped with magic_quotes_gpc (enabled by default until removed in PHP 5.4). It is also highly recommended to use database specific escape functions such as mysqli_real_escape_string() for MySQL. Each database management system supports one or more character escaping schemes specific to certain kinds of queries. If all user-supplied input is escaped using the proper escaping scheme for the database, it will not be confused with SQL code written by the developer. This will eliminate some possible SQL injection vulnerabilities, but a determined attacker will be able to find a way to inject SQL code. This technique is not as robust as the others, but may be considered if rewriting dynamic queries as parameterized queries or stored procedures might break a legacy application or have a significant negative impact on performance.

Some supplemental protection against SQL injection is offered by using a whitelist for input validation. Input validation can be used to detect unauthorized input before it is processed by the application. A whitelist approach to input validation involves defining exactly what input is authorized. All other input is considered unauthorized. A very rigid validation pattern using regular expressions should be created for well-structured fields such as phone numbers, dates, social security numbers, credit card numbers, email addresses, etc. If the data the user enters does not match the pattern it should not be processed. If the input field has a fixed set of options, such as a drop down list or radio buttons, then the input from that field must exactly match one of those values. The most challenging fields to validate are free text fields, such as forum entries. However, even these fields can be validated to some degree. Unexpected and unnecessary characters should be removed and a maximum size should be defined for the input field. All input fields should be validated using a whitelist.

Another measure that can be employed to supplement a main defense strategy is the principal of least privilege for database accounts. In order to limit the damage done by a successful SQL injection attack, administrator access rights should never be assigned to application accounts. Any given user should have access to only the bare minimum set of resources required to perform business tasks. This includes user rights and resource permissions such as CPU and memory limits, network, and file system permissions. Access should only be given to the specific tables an account requires to function properly. If stored procedures are used, then application accounts should only be allowed to execute the stored procedures that they use and they should not have direct access to database tables. It is important to restrict permissions so that if an attacker is able to successfully inject malicious code, the application's database user account will not have the authority to execute the command.

The database management system itself should also have minimal privileges on the operating system. Many of these systems run with root or system level access by default and should be changed to more limited permissions. This will make it more difficult for an attacker to gain system level access through SQL injection

CONCLUSION

It is important to know how to identify and remediate SQL injection vulnerabilities because the vast majority of data breaches are due to poorly coded web applications. Any code that constructs SQL statements should be reviewed for SQL injection vulnerabilities since a database server will execute all queries that are syntactically valid. Also, keep in mind that even data that has been parameterized can be manipulated by a skillful and persistent attacker. Therefore, web applications should be built with security in mind and regularly tested for SQL injection vulnerabilities. More information about SQL injection and how to prevent it, including specific examples for a variety of scripting languages, as well as guidelines to review code and test for SQL injection vulnerabilities can be found at the Open Web Application Security Project (OWASP). See references below. [2]

The MS-ISAC is interested in your comments - an anonymous feedback survey is available at: https://www.surveymonkey.com/r/MSISACProductEvaluation.

The MS-ISAC is the focal point for cyber threat prevention, protection, response, and recovery for the nation's state, local, tribal, and territorial (SLTT) governments. More information, as well as 24x7 cybersecurity assistance for SLTT governments, is available by contacting the MS-ISAC at 866-787-4722, SOC@cisecurity.org, or https://msisac.cisecurity.org/.

REFERENCES

[1] "2011 CWE/SANS Top 25 Most Dangerous Software Errors." mitre.org. Ed. Steve Christey. The MITRE Corporation, 2011. Web. November 8, 2012.

[2] The Open Web Application Security Project (OWASP):

- "SQL Injection." owasp.org. The Open Web Application Security Project, July 8, 2012. Web. November 7, 2012.
- "SQL Injection Prevention Cheat Sheet." owasp.org. The Open Web Application Security Project, July 8, 2012. Web. November 8, 2012.
- "Blind SQL Injection." owasp.org. The Open Web Application Security Project, July 8, 2012. Web. November 10, 2012.