Application Layer Intrusion Detection for SQL Injection

Frank S. Rietta
10630 Greenock Way
Duluth, Georgia 30097
frank@rietta.com

ABSTRACT

SQL injection attacks potentially affect all applications, especially web applications, that utilize a database backend. While these attacks are generally against the applications and not the database directly, there are some techniques that can be deployed to mitigate the risk at the database server. Database intrusion detection systems are often based on signatures of known exploits and honey tokens, traps set in the database. This paper examines the threat from SQL injection attacks, the reasons traditional database access control is not sufficient to stop them, and some of the techniques used to detect them. Moreover, it proposes a model for an anomalous SQL detector which observes the database traffic from the perspective of the database server itself. The proposed anomaly model can be used in conjunction with the existing methods to give the database server a way to mitigate the SQL injection risk that is a major application security problem.

Keywords

SQL injection, database security, anomaly detection

1. INTRODUCTION

1.1 Motivation

Web applications are becoming increasingly commonplace and accessible. Often the developers of these programs are focused on getting a working application under time pressure and may not implement the best security practices. Many applications are developed with loosely-typed scripting languages and make use of a single database user with full permissions, a so-called “god” user. The lack of strong types and invalidated database access control make for an increased risk of SQL injection vulnerabilities unless the programmer maintains the habit of strictly validating all input. This paper is a consolidated overview of the problem and some methods to mitigate the risks. In Section 1.2 we describe why SQL Injection is an important security threat, Section 1.3 explains Intrusion Detection techniques; Section 2 describes the Security and Control problems in databases; Sections 3 and 4 discuss Methods of Security in databases; Section 5 is devoted to common SQL injection attack techniques; and Section 6 will introduce our anomaly detection model for SQL injection.

1.2 SQL Injection

SQL injection is a technique often used to exploit database systems through vulnerable web applications [7]. The technique allows the attacker to not only steal the entire contents of relational databases but also, in many cases, to make arbitrary changes to both the database schema and the contents. Relational database server products have no mechanism to deal with SQL injection as the problem is rooted not in the database server itself but in vulnerable applications with excessive privileges granted to users. In most cases, a victim of an SQL injection attack does not even know that information is compromised until long after the attack has passed. Perhaps he may receive an angry e-mail from a customer who found his credit card number stolen or from the attacker himself seeking some form of blackmail. In many instances, victims are unaware that their confidential data has been stolen or compromised. While the details of SQL injection attacks vary among implementations of relational database systems (RDBMS), both commercial and open source RDBMSs are potentially susceptible to attack.

Most SQL injection attacks are executed through an application that takes user-supplied input for query parameters. The attacker supplies a carefully crafted string to form a new query with results very different from what the application developer intended. For example, consider a script on a web site that takes a search parameter like Zipcode to return selected results from a database. A very simple attack may be possible by simply providing something, like “1 OR 1=1” in the text field, which causes the SQL server to return all records from a particular table. An attacker can often gain access to anything available with the script’s privileges, which is often full access to one or more databases.

While SQL injection attacks could be executed against any application, web applications are the most commonly vulnerable. The attacker can easily explore a site for vulnerabilities without being caught or having to work through sophisticated network intrusion techniques as most prospective targets leave their web site applications wide open. Firewalls and traditional network intrusion detection systems are useless against SQL injection since it is an application exploit that in most cases is indistinguishable from expected use. Some signature-based detection systems have been developed for web servers to protect vulnerable scripts from malicious input. However, these signature-based systems are inherently susceptible to evasion methods that take advantage of the expressiveness of the SQL language or alternate character encodings. Remarkably, writing scripts that are not vulnerable to SQL injection is as simple as passing all user-provided text through a string escaping function prior to use as a parameter in an SQL statement. As past experience has shown, vulnerable scripts are everywhere.

SQL injection affects every database on every platform. Attacks can be used to gain information disclosure, to bypass authentication mechanisms, to modify the database, and, in some cases, to execute arbitrary code on the database server itself! This paper will examine ways to build an intrusion detection system specifically designed to be situated at the database server level to detect SQL injection attacks.

1.3 Intrusion Detection

Intrusion detection systems (IDS) are similar to burglar alarms that reside on the network to look for suspicious activity and alert the system and network administrators that there is a break-in. In practice, intrusion detection systems achieve varying degrees of success, and in many cases a clever intruder can evade detection. Those who decide to employ such systems must take into account the amount of noise in the system [3], the false positive and false negative rates.

In general, all intrusion detection systems are based either on signatures or anomaly models. A signature-based system must know explicitly about an exploit or an attack before it can be detected [5]. In order to be effective, the signature database must be constantly updated, sometimes by a subscription service, and in general can only offer protection against the most basic attacks. Unlike the signature-based system, which looks for known bad input, an anomaly-based system builds a profile of what is expected to be seen and alerts on any input which is outside of the profile. Generally, an anomaly-based IDS is a bi-modal system with a training mode and detection mode. While these systems can sometimes be trained as the system evolves, there are statistical methods for evading the system.

A networked IDS may have one or more observation points, usually called sensors, on the network. Generally, a sensor may be placed at the firewall or gateway into the network and will often be physically connected to the monitoring port of a primary network switch. This network layer sensor generally may have rules for particular protocols and will use packet inspection to look for suspicious activity. However, IDS sensors are not restricted to the network layer, and in some cases it will be useful to place a sensor at the application layer. This higher level sensor is located either within a particular server, such as a database server, or on a proxy in front of the server. This application layer sensor should have specific domain knowledge for the application protocol and does not have to deal with multiple levels of encapsulation and encoding that would be present at the network layer.

An ideal solution is to create an IDS sensor to be situated at the database server that will detect SQL injection attacks. As will be discussed, this sensor would be specifically designed to inspect SQL statements. Generic network intrusion detection systems are encumbered by encryption, multiple levels of encoding, and other encapsulation. By placing this IDS sensor on a database server proxy, it has access to the entire context of the SQL stream.

2. PROBLEMS WITH DATABASE ACCESS CONTROL AT LARGE

One important topic of consideration is why the standard database management system access control mechanisms are ineffective at stopping most SQL injection attacks. While some database management systems provide role-based authentication, most use basic user-based authentication mechanisms. Many simple database-driven applications do not have a sense of a traditional user and instead use a single database login which is used by all instances of the application. For many web site applications, this means that a single user has all privileges to a particular database so any injection against that application cannot be contained by the database's internal access control mechanisms. While more advanced applications could benefit from role-based access control, they often instead have a single database user and implement their own authentication methods. In both of these cases, the database has no way to mitigate access by an exploit to the application.

It would be better if application developers took advantage of the database server's user authentication mechanisms instead of rolling their own solution, but that is often not the case. When the database server and application make use of a non-integrated view of the users, there is a major opportunity for security exploits.

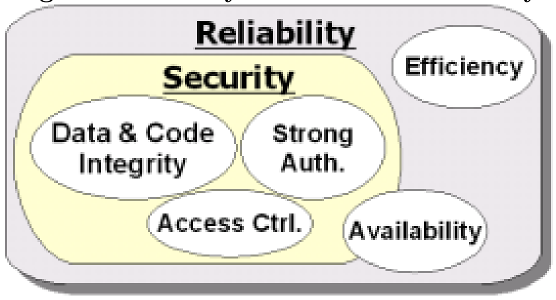
The use of a “god” user by most web applications presents a significant dilemma for database security for these applications. While the best long term solution is to fix the applications to take advantage of database authentication mechanisms in order to mitigate potential security threats, in many organizations it may be possible to deploy some additional functions at the database server level. For example, a program could monitor SQL traffic from an application for a period of time and then construct a set of least privileges for each query generated. The database proxy server could then maintain a pool of query-based access control privileges and ensure that no query is executed with more access privileges than required to execute.

3. RELIABILITY, SECURITY AND TRUST IN DATABASE SYSTEMS

According to Viega and McGraw, “Security boils down to enforcing a policy that describes rules for accessing resources” [9, page 14]. It is tempting to look at security as the same thing as reliability, and it is partially so. One way to look at reliability is as a measurement of how robust a system is with respect to software failures due to bugs. The definition of a bug can be looked at as a security policy [9].

There are a number of aspects, see figure 1, which come into play in the design and implementation of a database and its surrounding systems and all take part in the overall reliability of the system. The aspects of security, including data integrity, strong authentication, code integrity, and access control are all also part of reliability. However, effi-

Figure 1: Security is a subset of reliability.



ciency is not part of security. Availability is a special case that overlaps the two depending on the system.

Much of the work in database management systems has focused on robustness in terms of performance and basic security such as authentication, access control, and flow control. Many of these areas overlap and are all part of a reliable database system. However, if one just looks at reliability, it is sometimes easy to overlook some of the finer details of security. While access control may be in place, the database system will intrinsically trust the authenticated user. The biggest problem related to database security is trust management. Questions as to how much trust can be placed on an authenticated user is key. Best practices must be defined in such a way as to be automatically enforced.

4. DATABASE SECURITY METHODS

Database and information security often goes beyond the basics of access control mechanisms. Generally, all methods for intrusion detection fall either into the signature-based or into the anomaly-based categories. The main difference between these approaches is that a signature-based system looks for known things to block while an anomaly-based system learns what normal behavior looks like over time and detects things that do not fit the normal profile. There are commercial and home-grown database intrusion detection systems used today that basically employ three methods: signatures, honey tokens, and anomaly models.

4.1 Signatures

The classic signature-based detection system maintains a list of known attacks and is usually deployed as part of the web application instead of being placed in front of the database server. A signature may be as simple as a regular expression describing a particular string pattern, such as UNION SELECT, used in a known attack. The problem with signatures is that knowledge about the exact attack must be known ahead of time, and it is relatively easy for the attacker to adjust the input to evade the signature. Signature evasion techniques include using different encodings, fragmenting input across packets, changing to a different but equivalent expression, or changing the location and usage of white space [6]. Creative use of white space characters that do not effect that meaning of the SQL, but can fool a simple IDS signature is a typical evasion technique.

One instructive example of signature evasion would be if a signature is in place to stop input such as "OR 1=1" it may be possible to bypass this with equivalent input such as "OR 'One' = N'One'." Both of these comparisons are valid SQL which will return true in all cases, the later being a string comparison with the N data type cast being a valid part of the SQL language.

Fragmentation techniques do not generally thwart more advanced network IDS sensors or application level sensors. While signature protection is widely deployed today and will mitigate most basic attacks, it is not enough to protect against SQL injection [6]. The defender's dilemma is that SQL is a big language with a lot of possible variations. It is simply impossible to build a signature list against all possible bad inputs and their effects on all database schema.

4.2 Honey Tokens

A honey token is some type of digital entity, be it a credit card number, a spreadsheet, a presentation, a database entry, or even a bogus login. Honey tokens come in many shapes and sizes. However, they all share the same concept: a digital or information system resource whose value lies in the unauthorized use of that resource [8]. Honey tokens can be easily deployed to help protect a large variety of database systems and are particularly useful to catch internal information and privacy violations by employees. Both credit card and social security numbers have algorithms which can be used to check a particular number to see if it is valid. Invalid numbers can be attributed to certain records and an alarm sounded if those values ever leave the database server. Honey token based IDS sensors are flexible, easy to deploy, and particularly useful against insider attack.

4.3 Anomaly Models

Anomaly detection models vary in both technique and application depending on the domain being addressed. An ideal anomaly model which detects all attacks with zero false positives is the holy grail for an intrusion detection system, but in practice this is not achievable. However, there are some good models for specific areas such as SPAM detection and network worm detection. In general, a normal profile is created through a training mode or by incremental learning looking either at non-exploit network traffic or for so-called HAM messages in the case of e-mail. When in detection mode, some distance algorithms are used to compare current data against the normal model and any distance which crosses some threshold is considered to be an anomaly.

5. FORMS OF SQL INJECTION ATTACKS

An SQL injection attack is, in nearly all instances, an application exploit which takes advantage of the database management system. While the attack affects all applications that deal with databases, the most common victims are web applications because they are wide open to access. The attack goes straight through any firewall protection, which provides little or no security, and in most cases totally bypasses any access control that the DBMS would have traditionally provided.

While some attacks are easier to implement than others, the ease of any particular attack does not imply that the danger resulting from the attack is lessened. An analogous example can be brought in from the study of faults: a trivial fault in a system may have disastrous consequences while a

Figure 2: Forms of SQL Injection Attacks

Attack Type	Results
Unauthorized data access	Allows the attacker to trick the application in order to obtain from the database information that is not supposed to be returned or is not allowed to be seen by this user.
Authentication bypass	Allows the attacker to access the database-driven application and observe data from the database without presenting proper credentials.
Database modification	Allows the attacker to insert, modify, or destroy data content without authorization.
Escape from a database	Allows the attacker to compromise the host running the database or even attack other systems.

[2, page 378]

large fault may go unnoticed.

The end goal of any injection attack is to execute against the database SQL statements and/or queries so that the attack against the database is successful. Figure 2 outlines the four types of SQL injection attacks: unauthorized data access, authentication bypass, database modification, and escape from a database.

One of the challenges to detecting SQL injection attacks at the database server is that an effective SQL injection is both semantically and syntactically valid. This means that some anomaly detection rules, such as those used to catch SPAM e-mail messages, are ineffective for SQL, which is a structured programming language and not a natural language with loose grammar. A detection system is likely to have either a high rate of false positives or false negatives, hindering its effectiveness. The structured nature of the SQL language allows some techniques to be used which would not work in SPAM detection systems.

5.1 Parameterization Attack

Most applications prepare an SQL statement for submission to the database by taking a preexisting SQL template and inserting user-supplied text such as user names and search terms. If the application does not perform adequate validity checking on the user-supplied input and if the SQL template contains certain exploitable locations, then an SQL injection attack is possible. When an attack relies on placing strategically crafted input into an SQL template, the attack is classified as a parameterization attack.

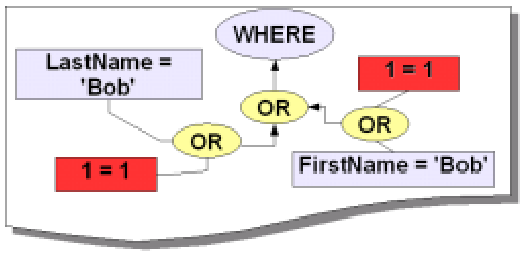
Some queries are easier to exploit than others. Those that expect a numerical parameter or that place a parameter at the end of the query, a so-called dangling parameter, are easier to transform to an alternate valid query.

5.1.1 Dangling Parameters

A dangling parameter is a first-order replacement in an SQL query. There is generally nothing following the final parameter, which leaves room for the syntax to be modified.

One example query rewrite happens when an application

Figure 3: An example parse tree for a second-order string replacement. The expression is always true.



inserts a string into a search term at the end of a query. Even though the template expects a name to be provided, a clever string is provided to totally rewrite the query:

```
SELECT * FROM Directory WHERE LastName LIKE '${NAME}';
```

can be transformed to:

```
SELECT * FROM Directory WHERE LastName LIKE 'frank'
   OR 1=1
   UNION SELECT user, password
   FROM mysql.user WHERE 'q'='q' OR "
```

This modified statement will fetch a full listing of the MySQL user table, including passwords, if the privileges are sufficient. While this query will likely fail, most attackers will try it as there are applications with full permissions.

5.1.2 Second-Order Replacement

Consider the example of a second-order string replacement in figure 3. The name provided by the user is applied to both the FirstName and LastName constraints:

```
SELECT * FROM Listing WHERE
   LastName='${NAME}' OR LastName='${NAME}'
```

is transformed, by setting ${NAME} = "Bob' OR 1=1", to:

```
SELECT * FROM Listing WHERE
   LastName='Bob' OR 1=1 OR LastName='Bob' OR 1=1
```

which will return every record in the Listing table since the constraint always evaluates to true. However, in other cases a second-order replacement is harder to exploit than a first-order replacement. This presents challenges for those crafting clever input.

5.1.3 Unquoted Numerical Parameters

The most dangerous location for exploit is a SQL statement with an unchecked value being placed in an unquoted parameter template. Consider the following query intended to update a number in a particular financial record:

```
UPDATE   Financial_Records
  SET    Salary = ${NEW_SALARY}
  WHERE  Name = '${NAME}'
```

The problem with this SQL template is that there are no quotes around the ${NEW_SALARY} parameter. SQL does not require quotes around numerical values but, unless the application enforces the restriction on the input data, there is potential for this parameter to be exploited. Without quotes present in the template, any string literal other than a number will be part of the SQL statement and will not be treated as a literal by the DBMS, resulting in a direct injection [7]. This does not mean that a value other than a number will be placed into Salary by the SET statement, but that the absence of quotes allows the query itself to be changed. Even when the application framework automatically escapes string literals, such as magic quotes do in PHP, a direct injection is possible when quotes are not present.

5.2 Batch Query Engines

Some SQL servers allow multiple queries to be bundled up into a single transaction, and some application frameworks support batch queries where each query is separated by a deliminator character, such as a semicolon.

Consider again the first dangling parameter example:

```
SELECT * FROM Directory WHERE LastName LIKE '${NAME}';
```

When a batch query engine is in use, it is possible to transform the string into more than one query. For example, it is possible to transform the above template into:

```
SELECT * FROM Directory WHERE Last_Name LIKE '';
UPDATE Directory SET Phone='555-1212'
  WHERE First_Name='Frank';
```

Depending on the batch engine being used, the first query will fail, but the subsequent SQL statements are used to change a piece of data in the relational database. The first query runs and returns a null search result but the second query executes and changes a field within the database.

6. A PROPOSAL FOR AN ANOMALY DETECTION MODEL

One class of anomaly detection models, such as the one used in the anomalous payload-based network intrusion detection system [10], breaks the data being analyzed into a number of buckets. The relative frequencies of these buckets are then used to quantify the observation and compare it against historically observed normal data. While the relative frequencies of certain bytes, or groups of bytes, are useful when analyzing network traffic, it is important to consider the structure of the queries in the observed SQL traffic.

An anomaly model for SQL must take into account specific characteristics of the database query language. See figure 4 for an overview of some fundamental characteristics. The nature of SQL allows the model to focus on a finite number of collectible data points, group them according to type, compute the relative frequencies of each group, and then compare them against the historically observed normal traffic. Since each application will likely generate a unique normal profile, the IDS must keep track of each application's SQL traffic separately.

6.1 Unexpected Constructs

Particular applications tend to generate very consistent SQL traffic with only a subset of the database features in use. Therefore, it is useful to give a weighted score to a query in which particular SQL features, that are not in the normal profile, are used. Constructs of interest include:

Figure 4: Fundamental SQL characteristics which can be observed by an application layer IDS sensor.

SQL	*Functionality*	*Example*
SELECT	Extract data from the database.	SELECT * FROM user_table;
UNION	Combine the results of several SELECT queries together, removing duplicate records.	SELECT first, last FROM customers WHERE city = 'NYC' UNION SELECT first, last FROM prospects WHERE city='NYC';
INSERT	Put new data into the database table, add a new row to the table.	INSERT INTO item_features VALUES (130012,4);
UPDATE	Change the records in the database.	UPDATE items SET description = 'New Honeypot' WHERE item_id = 15002;
DELETE	Delete specific records from a table.	DELETE FROM alerts WHERE devicetypeid = 13 AND alarmid NOT IN (1,2,5);
CREATE	Create new data structures (such as tables) within the database.	CREATE TABLE high AS SELECT * FROM events WHERE name = 2;
DROP	Remove the table from the database.	DROP TABLE user_table;
ALTER	Modify the database table by adding columns.	ALTER TABLE user_table ADD address varchar(30);
WHERE	Defines the fields to be processed by SELECT, INSERT, DELETE, and other commands.	SELECT * FROM user_table WHERE username = 'anton';
LIKE	Facility used to do approximate matching within the WHERE clause; the '%' indicates a wild card.	SELECT * FROM user_table WHERE username LIKE 'anton%';
AND, OR, NOT	Binary logic comparison operators used, for example, within WHERE clauses.	SELECT * FROM user_table username = 'anton' AND password = 'correcto';
VALUES	Used to specify the inserted or changed values for the INSERT and UPDATE commands.	INSERT INTO user_table (username, password) VALUES ('anton', 'correcto');

[2, page 375]

- Sub-queries
- Literals as left-hand-values or right-hand-values
- Previously unused SQL keywords
- Unquoted values when quoted values are expected
- Unexpected character set or encoding

6.2 General Statistics

Other particulars of interest to the statical model include:

- Query length and encoding
- Keyword usage
- Right-hand/Left-hand bias for constraint qualifiers
- Data types used for particular parameters

6.3 Query Groups, Anomaly Estimation, and Deviations

SQL statements must be grouped with similar statements to which each can be compared with the anomaly model. A simple grouping could be done on the type of query, SELECT, INSERT, UPDATE, etc. The grouping should take into account the length of the query, the type of data, and which tables and columns are being accessed.

For each SQL statement being observed, the algorithm will use the collected data-points to compute a distance, such as a Mahalanobis distance [10], from the normal profile for a particular query group. Each data point distance will be multiplied by the corresponding danger weight, through a table lookup, and a summation calculated for all of the weighted points. This sum should be compared to a reasonable threshold, which may also be learned from the model.

A query that deviates from the normal profile above the threshold would be flagged and be handled accordingly. The anomalous queries could be treated to more computationally intense screening, stopped all together, or allowed to pass through to the database server.

7. RELATED WORK

7.1 Static Analysis of Application SQL

A technique that uses a model-based approach to detect illegal queries before they are executed on the database. In its static part, the technique uses program analysis to automatically build a model of the legitimate queries that could be generated by the application. In its dynamic part, runtime monitoring is used to inspect the dynamically-generated queries and check them against the statically-built model [4].

7.2 SQL Randomization

A protection mechanism that applies the concept of instruction-set randomization to SQL, creating instances of the language that are unpredictable to the attacker [1].

8. CONCLUSIONS & FUTURE RESEARCH

8.1 Contribution of this Paper

SQL injection potentially affects every database on every platform. Attacks can be used to gain information disclosure, to bypass authentication mechanisms, to modify the database, and to execute arbitrary code, in certain instances, on the database server itself. While several techniques are available to mitigate the risk of SQL injection attacks, we propose that an additional measure of protection be added. An application layer intrusion detection system should take the form of a proxy server and employ an anomaly detection model based on specific characteristics of SQL and the transaction history for a particular user and application. Being designed with domain knowledge specific to RDBMSs, this new IDS should be more accurate than a generic network anomaly detection system.

8.2 Proposed Future Research

More research on this topic should focus on building an experimental application layer IDS sensor to collect the data necessary to see how this model performs in practice and to find ways it can be improved. It may be possible to improve the accuracy of the sensor by moving it into the DBMS in order to examine the internal representation after the query optimizer has processed each SQL statement.

9. ACKNOWLEDGMENTS

Special thanks to Shamkant B. Navathe, professor at Georgia Tech, who provided valuable advice throughout the writing process. This undergraduate research project would have not come so far without his help. Thanks also to Harrison Caudill, Jonathan Cullifer, and Eric McCorkle for feedback.

10. REFERENCES

[1] S. Boyd and A. Keromytis. Sqlrand: Preventing sql injection attacks, 2004.

[2] A. C. Cyrus Peikari. *Security Warrior*. O'Reilly Media, Inc., Sabastopol, CA, 2004.

[3] D. D. W. L. Guofei Gu, Prahlad Fogla and B. Skoric. Measuring intrusion detection capability: An information-theoretic approach. In *Proceedings of ACM Symposium on InformAction, Computer and Communications Security (ASIACCS'06)*.

[4] W. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of the IEEE and ACM International Conference on Automated Software Engineering (ASE 2005)*, pages 174–183, Long Beach, CA, USA, Nov 2005.

[5] G. Hoglund and G. McGraw. *Exploiting Software: How to Break Code*. Addison-Wesley Professional, Boston, MA, 2004.

[6] O. Maor and A. Shulman. Sql injection signatures evasion: An overview of why sql injection signature protection is just not enough. `http://www.imperva.com/application_defense_center/white_papers/sql_injection_signatures_evasion.html`, 2004.

[7] K. Spett. Sql injection: Are your web applications vulnerable? `http://www.spidynamics.com/whitepapers/WhitepaperSQLInjection.pdf`, 2002.

[8] L. Spitzner. Honeytokens: The other honeypot. `http://www.securityfocus.com/infocus/1713`, July 2003.

[9] J. Viega and G. McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley Professional, Boston, MA, 2001.

[10] K. Wang and S. J. Stolfo. Anomalous payload-based network intrusion detection. In *RAID*, pages 203–222, 2004.