Countering SQL Injection Attacks with a Database Driver[1,2]

Dimitris Mitropoulos, Diomidis Spinellis

{dimitro,dds}@aueb.gr

Abstract

SQL injection attacks involve the construction of application input data that will result in the execution of malicious SQL statements. Many web applications today, are prone to SQL injection attacks. This paper proposes a novel methodology of preventing this kind of attacks by placing a secure database driver between the application and its underlying relational database management system. To detect an attack, the driver creates query blueprints that are then used to distinguish between injected and legitimate queries. The driver depends neither on the application nor the RDBMS and can be easily retrofitted to any system. Finally we have developed a tool, SDriver, that implements our technique and used it on several web applications with positive results.

Keywords: SQL injection attack, SQLIA, web security, JDBC driver, firewall

1. Introduction

Traditionally, most programmers have been trained in terms of writing code that implements the required functionality without considering the security aspects in many ways [Joshi (2005)]. It is very common, for a programmer, to make false assumptions about user input [Wassermann and Su (2004)]. Classic examples include: assuming only numeric characters will be entered as input, or that the input will never exceed a certain size etc.

---

[1] In Theodore S. Papatheodorou, Dimitris N. Christodoulakis, and Nikitas N. Karanikolas, editors, *Current Trends in Informatics: 11th Panhellenic Conference on Informatics, PCI 2007*, volume B, pages 105–115, Athens, May 2007. New Technologies Publications.

[2] This is a machine-readable rendering of a working paper draft that led to a publication. The publication should always be cited in preference to this draft using the reference in the previous footnote. This material is presented to ensure timely dissemination of scholarly and technical work.

SQL injection attacks comprise a subset of a wide set of attacks known as code injection attacks [Keromytis and Prevelakis (2003)] [Barrantes et al. (2003)]. Code injection is a technique to introduce code into a computer program or system by taking advantage of the unchecked assumptions the system makes about its inputs [Younan et al. (2005)].

Many web applications have interfaces where a user can input data to interact with the application's underlying database. This input becomes part of an SQL statement, which is then executed on the RDBMS. Code injection attacks that exploit the vulnerabilities of these interfaces are called "SQL injection attacks" (SQLIA) [CERT (2002)] [Joshi (2005)] [Litchfield (2005)] [Su and Wassermann (2006)] [Howard and LeBlanc (2003)] [Viega and McGraw (2001)]. There are many forms of SQL injection attacks. The most common are [McDonald (2005)]:

- Taking advantage of incorrectly filtered escape characters
- Taking advantage of incorrect type handling

With this kind of attacks, a malicious user can view sensitive information, destroy or modify protected data, or even crash the entire system [Anley (2002)] [Cerrudo (2004)]. The following example takes advantage of incorrectly filtered escape characters. In a login page, besides the user name and password input fields, there is usually a separate field where users can input their e-mail address in case they forget their password. The statement that is probably executed has the following form:

*SELECT * FROM passwords WHERE email = 'theemailIgave@example.com';*

If an attacker, inputs the string: *anything' OR 'x'='x*, he will actually view every item in the table. In a similar way, the attacker could modify the database's contents or schema.

An "incorrect type handling" attack occurs when a user supplied field is not strongly typed or is not checked for type constraints. For example, many websites allow users, to access their older press releases. A URL for accessing the site's fifth press release could look like this [Spett (2004)]:

http://www.website.com/pressRelease.jsp?pressReleaseID=5

And the statement that is probably executed:

SELECT description, date, body FROM pressReleases WHERE pressReleaseID = 5

If an attacker wished to find out if the application is vulnerable to SQL injection, he could change the URL into something like:

http://www.website.com/pressRelease.jsp?pressReleaseID=5 AND 1=1

If the page displayed is the same page as before, it is clear that the field pressReleaseID is not strongly typed and the end user can manipulate the statement as he chooses.

According to vulnerability databases like CVE[3] (Common Vulnerabilities and Exposures) and security providers such as Secunia[4] and Armorize Technologies[5] SQLIA incidents have increased in the last years.

In this paper we propose a novel technique of preventing SQLIAs. Our technique incorporates a driver that stands between the web front-end and the back-end database. The key property of this driver is that every query can be processed and identified using certain query characteristics. By analyzing these characteristics during a learning phase, we can build a model of the legitimate queries. Then at runtime our driver checks all queries for compliance with the learned model.

2. Design

2.1 Architecture

The architecture of typical tiered web applications consists of at least an application running on a web server and a back-end database [Wassermann and Su (2004)]. Between these two tiers, there is always a database connectivity driver that supports protocols like ODBC (Open Database Connectivity), or JDBC (Java Database Connectivity). The main function of such a driver is to provide a portability layer by retrieving SQL statements from the application and forwarding them to the database.

The driver that we propose is also a connectivity driver. It stands between the application and the database interface driver we mentioned above (see Figure 1). Our driver is transparent; its only operation is to prevent SQLIAs; it depends neither on the application, nor on the connectivity driver.

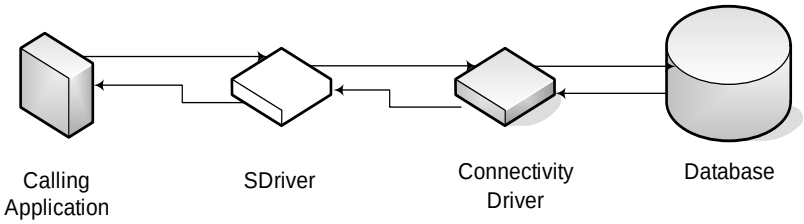

Figure 1. *The architecture of our proposed driver (SDriver)*

[3] http://cve.mitre.org/
[4] http://secunia.com
[5] http://www.armorize.com

To work as a connectivity driver, our driver implements the interfaces of the connectivity protocol. However, most of the driver's methods simply forward the request to the underlying connectivity driver. In this respect our driver acts as a *shim* or *proxy* for the underlying driver. A few methods capture and process requests in order to prevent SQLIA.

2.2 Preventing SQLIAs

In order to secure the application from SQLIAs the driver must go through a learning phase. During this phase all the SQL queries of the application must be executed so that the driver can identify them in a way we will show in the next section. Then the driver's operation can shift into production mode, where the driver takes into account all the legal queries and can thereby prevent SQLIAs.

2.2.1 Learning Mode

Every SQL query of an application can be identified combining three of its characteristics.

- Its call stack trace. This involves the stack of all methods from the method of the application where the query is executed down to the target method of the connectivity driver.
- Its SQL keywords.
- The tables that the query uses in order to retrieve its results.

A mathematical representation of the above could be the following: If K is the set of the stack traces; L is the set of the SQL keywords and M the set of the application tables, the set of the query IDs called S will be defined as follows:

$$S = \{\omega: \omega = k\ (l^*m^*) +, k \in K, l \in L, m \in M\}$$

It is easy for someone to see that a query cannot be identified by using one of the above characteristics alone. To combine these characteristics, when a query is being transferred to the database our driver carries out two actions. At first, it strips down the query, removing all numbers and strings. So if the following statement is being executed:

*SELECT * FROM table1, table2 WHERE field1 = 'foo' AND field2 = 3*

The driver removes 'foo' and 3 and saves the stripped down query. Then it goes all the way down the call stack, saving the trace, until it reaches the statement's origins. By associating a complete stack trace with the root of each query, our tool can correlate queries with their call sites. The trace and the stripped down query are concatenated and the driver applies a hash function on them. The result is the query's

blueprint. All the blueprints are saved during the learning mode so that the driver can check during the production mode if a query is legal or not.

2.2.1 Production Mode

The driver's functionality during the production mode does not differ from the one in the learning mode. The steps are the same until the driver gets the query's blueprint. At that point, if the driver identifies it as a legal one then the query passes through. If it does not then the application is probably under attack. In such a case the driver can halt the application with an exception, it can log an error message, or it can forward an alarm to a larger intrusion detection system.

Take for example an attack that takes advantage of the escape characters. The additional keywords that the malicious user injects will definitely lead to an unknown blueprint. In this case the driver becomes aware of the attack and prevents it.

3. Implementation

We have implemented our solution in Java. The driver is called SDriver and acts as a JDBC driver wrapped around other drivers that implement a database's JDBC protocol (see Figure 2).

3.1 Architecture

JDBC drivers known as "native-protocol drivers[6]" (or type 4 JDBC drivers) convert JDBC calls directly into the vendor-specific database protocol. At the client side, a separate driver is needed for each database. SDriver does not depend on the application or the native driver as we have already mentioned. As we also mentioned, it must be placed between the application and the underlying RDBMS driver. To accomplish that, the application's code must be modified only in one position. The modification will take place where the application establishes a connection with a driver. For the application to be secured, the SDriver must establish a connection with the driver that the application is meant to use. To achieve that, we pass the driver's name through the URL of the earlier connection (see Figure 1). For example, if the application is meant to connect to the Microsoft SQL Server 2000 the source code would look like this:

Class.forName ("com.microsoft.jdbc.sqlserver.SQLServerDriver");

Connection conn=DriverManager.getConnection ("jdbc: microsoft: sqlserver: //localhost:1433; databasename = MyDB"," username", "password");

After calling the SDriver, the modified code would be:

[6] http://java.sun.com/products/jdbc/driverdesc.html

Class.forName ("com.SDriver");

Connection conn = DriverManager.getConnection ("jdbc: com.microsoft.jdbc.sqlserver.SQLServerDriver: microsoft: sqlserver: //localhost:1433; databasename = MyDB", "username", "password");

SDriver is not a classic native-protocol RDBMS driver. The implementation of most of the driver's methods simply involves calling the corresponding methods of the underlying driver. That's how SDriver becomes transparent, flexible, and underlying driver-independent.

3.2 Preventing SQLIAs

There are certain methods that a native-protocol driver implements that can be described as critical. They are the ones that execute queries. Methods like: *executeQuery*, *executeUpdate* and others. To secure against SQLIAs, SDriver interferes in these methods examining the query string that is about to be executed.

Right before the execution, a method called *manageQuery* retrieves the query string, and performs the following steps: At first, *manageQuery* opens a text file to find out whether the driver operates in learning or production mode (this allows operators to switch from learning to production mode without halting the application). Regardless of the mode, the next step is to call the method *strippedDownQuery*. This method removes all strings, numbers and comments from the query using regular expressions. The stripped query is sent as a parameter to a method called *getStackID*. *getStackID* traverses the call stack and returns the complete stack trace. After that the complete stack trace is concatenated with the stripped query, and the two are passed as input to an MD5 hash algorithm. The outcome is the query's blueprint. Finally, the method *queryFilter* is called. *queryFilter* takes as a parameter the blueprint of the query and a Boolean variable indicating whether the driver operates in learning or production mode. During learning mode the blueprints of encountered queries are stored in a database called *ssql*. During production mode the blueprints of encountered queries are retrieved from the database. A failed retrieve operation is a sign of a probable SQLIA. In our implementation the driver will block the query and will not allow its execution.

As we mentioned above, query blueprints are stored in a database called *ssql*. In order to interact with ssql, SDriver must set up a JDBC connection. This connection is created in the Connection interface of SDriver and it is stored in a private variable during the whole session.

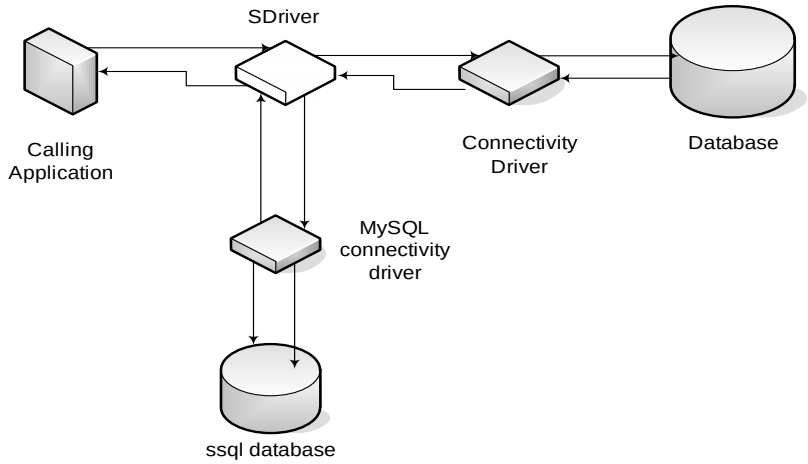

Figure 2. *SDriver's actual architecture*

4. Evaluation

4.1 Accuracy

We evaluated SDriver on simple real world applications written in the JSP framework. We managed to attack these applications using several SQL injections. Specifically we managed to access supposedly protected data, delete tables, and modify elements. After installing SDriver the applications proved to be resistant to our SQLIAs but with a significant overhead.

Thereupon, we searched for real-world web applications that had a record of being vulnerable to SQLIAs. According to the vulnerability database CVE, and the security providers US-CERT[7] (United States-Computer Emergency Readiness Team), Secunia and Armorize Technologies, one of the most known vulnerable applications was Daffodil CRM 1.5. Daffodil CRM is a commercial open source CRM Solution that helps enterprise businesses to manage customer relationships. In Daffodil 1.5, remote attackers could execute arbitrary SQL commands via unspecified parameters in a login action. In particular, this flaw is due to input validation errors in the "userlogin.jsp" script that does not properly validate the "userLoginBox" and "passwordBox" parameters before being used in SQL statements, which may be exploited by attackers to conduct SQL injection attacks and gain unauthorized access to a the application (specifically by taking advantage of escape characters). We found Daffodil's code and tested it with our tool. SDriver proved to be robust and Daffodil was protected. Injected queries were prevented with no results being displayed when an attack was taking place. We encountered the same positive results when we used

[7] http://www.us-cert.gov

SDriver with another noted application, known as WebGoat. WebGoat is a deliberately insecure web application maintained by OWASP[8] and it was designed to teach web application security lessons.

Overall, SDriver was designed to prevent all known kinds of SQLIAs and so far it has succeeded. Its major disadvantage though is the overhead it causes on the processing time.

4.2 Computational Overhead

The driver's architecture allowed us to test SDriver's performance on two RDBMSs: SQL Server 2000 and MySQL. We first tested the baseline overhead of the SDriver by executing a simple JDBC method -getAutoCommit()- that is passed through directly to the underlying database driver without further processing. The results (see table 1), point out that the cost of interposing SDriver is not unreasonable, with an impact of 6% at most.

Table 1. *Proxy Driver Cost*

Application Database	Execution time (ms)		
	Original	SDriver	Overhead (%)
SQL Server	78	79	1
MySQL	16	17	6

Subsequently, we tested the overhead of the SQLIA detection code by executing a moderately complex SQL statement, with and without SDriver. The performance overhead for the two RDBMSs was similar (see Table 2). SDriver currently incurs a significant overhead. However, the current version makes extended use of regular expressions for stripping the SQL statements. We expect the driver's performance to improve significantly, once we optimize this code.

Table 2. *Query Cost*

Application Database	Execution time (ms)			Overhead (%)	
	Original	Learning	Production	Learning	Production
SQL Server	813	3016	3078	271	279
MySQL	797	2796	2703	251	239

5. Related work

[8] http://www.owasp.org/index.php/Main_Page

Until 2004, when the incidents started to increase, little attention had been paid to SQL injection attacks. One of the first mechanisms proposed in the literature was SQLrand [Boyd and Keromytis (2004)]. SQLrand protects from SQL injection by randomizing the SQL statement, creating instances of the language that are unpredictable to the attacker. It is implemented as database server proxy but requires source code modification.

SQLBlock [SQLBlock.com (2005)] offers protection by variable normalization of SQL statements with a low performance on big applications. SQLGuard [Buehrer, et al. (2005)] and SQLCHECK [Su and Wassermann (2006)] are mechanisms that use parse tree validation. SQLGuard is not flexible enough, because the source code of the application must be modified in many positions just like SQLrand. SQLCHECK on the other hand has a complex set up.

One of the latest tools that use an approach similar to ours is AMNESIA [Halfond and Orso (2005, 2006)]. However, the tool apparently suffers from many false positives and negatives. AMNESIA at first identifies the critical spots on the application's code (the ones that execute SQL statements) and then builds query models.

Finally, there are several publications that propose methodologies on preventing SQLIA incidents [Younan, et al. (2005)], but don't put forward a specific mechanism.

6. Conclusions

SDriver is a mechanism and a prototype application that prevents SQLIAs against web applications. If an SQL injection happens, the structure of the query, and therefore its blueprint will be altered, and SDriver will be able to detect it. By associating a complete stack trace with the root of each query we increase the specificity of the stored query blueprints and avoid false negative results. The increased specificity of the blueprints also allows us to discard a large amount of the query's elements, thereby also reducing false positive results. A disadvantage of our approach is that when the application is altered, the new source code structure invalidates existing query blueprints. This necessitates a new learning phase. However, with the increased use of automated testing frameworks, this learning phase can probably be included as part of the application's testing.

Future work on our approach involves the quantitative evaluation of its performance in terms of accuracy and computational efficiency. We also plan to provide a diagnostic front-end that will allow operators to interact with the driver.

Acknowledgement This work was partially funded by the European Community's Sixth Framework Programme under the contract IST-2005-033331 "Software Quality Observatory for Open Source Software (SQO-OSS)".

References

Anley, C. (2002). *Advanced SQL Injection in SQL Server Applications*. Next Generation Security Software Ltd. White Paper.

Barrantes, E. G., Ackley, D. H., Palmer, T. S., Stefanovic, D., and Zovi, D. D. (2003). Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security* (Washington D.C., USA, October 27–30, 2003). CCS '03. ACM Press, New York, NY, pp. 281–289.

Boyd, S., Keromytis, A. (2004). SQLrand: Preventing SQL injection attacks. In Jakobsson, M., Yung, M., Zhou, J., eds.: *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*. Volume 3089 of Lecture Notes in Computer Science. Springer-Verlag. pp. 292–304.

Buehrer, G., Weide, B. W., and Sivilotti, P. A. (2005). Using parse tree validation to prevent SQL injection attacks. In *Proceedings of the 5th international Workshop on Software Engineering and Middleware* (Lisbon, Portugal, September 05–06, 2005). SEM '05. ACM Press, New York, NY. pp. 106–113.

CERT (2002). *AdCycle does not adequately validate user input thereby allowing for SQL injection.* CERT Vulnerability Note VU#282403. Online http://www.kb.cert.org/vuls/id/282403, accessed January 7th, 2007.

Cerrudo, C. (2004). *Manipulating SQL Server Using SQL Injection*, Application Security Inc.

Halfond, W. G. and Orso, A. (2005). AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks. In *Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering* (Long Beach, CA, USA, November 07–11, 2005). ASE '05. ACM Press, New York, NY, pp. 174–183.

Halfond, W. G. and Orso, A. (2006). Preventing SQL injection attacks using AMNESIA. In *Proceeding of the 28th International Conference on Software Engineering ICSE '06.* (Shanghai, China, May 20 - 28, 2006). ACM Press, New York, NY, pp. 795–798.

Howard, M. and LeBlanc, D. (2003). *Writing Secure Code*. Microsoft Press, Redmond, WA, second edition.

Joshi, S. (2005). *SQL Injection Attack and Defence*, Online http://www.heise-security.co.uk/articles/75593. Accessed January 7th, 2007.

Kc, G. S., Keromytis, A. D., and Prevelakis, V. (2003). Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security CCS '03.* (Washington D.C., USA, October 27 - 30, 2003). ACM Press, New York, NY, pp. 272–280.

Litchfield, D. (2005). SQL injection and Data Mining through inference, http://www.databasesecurity.com/dlitchfield/aboutme.htm

McDonald, S. (2005). SQL injection: Modes of attack, defence, and why it matters. Online

http://www.governmentsecurity.org/articles/SQLInjectionModesofAttackDefenseandWhyItMatters.php. Accessed, January 7th, 2007.

Ng, S. M.S. (2005). *SQL Injection Protection by Variable Normalization of SQL Statement*. Online http://www.sqlblock.com/sqlblock.pdf. Accessed, January 7th, 2007.

Spett, K. (2004). *Blind SQL Injection*, Online http://www.spidynamics.com/whitepapers/Blind_SQLInjection.pdf. Accessed, January 7th, 2007.

Su, Z. and Wassermann, G. (2006). The essence of command injection attacks in web applications. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL '06* (Charleston, South Carolina, USA, January 11 - 13, 2006). ACM Press, New York, NY, pp. 372–382.

Viega, J. and McGraw, G. (2001). *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison Wesley Professional.

Wassermann, G. and Su, Z. (2004). An Analysis Framework for Security in Web Applications. In *Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2004)*.

Younan, Y., Joosen, W. and Piessens, F. (2005). A methodology for designing countermeasures against current and future code injection attacks. In *Proceedings of the Third IEEE International Information Assurance Workshop 2005 (IWIA2005)*.