Analysis of SQL injection prevention using a filtering proxy server

By David Rowe
Project Supervisor: Barry Irwin
Department of Computer Science
Rhodes University
September 2005

Abstract - This paper details an analysis of SQL injection prevention. The paper contains two parts. The first highlights methods that should be adopted in order to reduce the risk of an SQL injection attack. The second details the creation of a filtering proxy server used to prevent a SQL injection attack.

KEYWORDS
SQL injection, fault injection, SQL poisoning, prevention, preventing.

1. INTRODUCTION

SQL injection is a method by which users take advantage of dynamic SQL through which parameters of a web-based application are chained together to create the query to the backend database [Finnigan, 2002] [Anley, 2002]. There are many measures that can be taken to prevent SQL injection including making sure that the users have the minimum database privileges possible, using input validation programming techniques, suppressing error messages returned to the client, checking error logs and filtering malicious SQL statements [Finnigan, 2003]. In research and commercial products, there is evidence proving that SQL injection can be prevented using means not so closely related to the database and web application. These methods of approach have been developed to produce a more generic solution to a problem that requires a lot of tweaking and attention to detail at the root of the problem - the application code and database deployment [Ristic, 2005] [Seclutions, 2003]. Auditing all of the source code and protecting dynamic input is not trivial, neither is reducing the permissions of all applications users in the database itself [Finnigan, 2003]. Developing a filter seems to be the best solution to preventing SQL injection.

2. EXAMPLE OF SQL INJECTION

A typical SQL statement is shown in code box 1.

```
select id, forename, surname from authors where forename = 'Joe' and surname = 'Bloggs'
```

Code Box 1: A typical SQL statement

An important point to note is that the string literals are delimited by single quotes. The user may be able to inject some SQL if the user provides the input shown in text box 2.

```
Forename: Jo’e
Surname: Bloggs
```

Text Box 1: User input

The query string formed from the input shown in text box 1 is shown in code box 2.

```
select id, forename, surname from authors where forename = 'Jo’e' and surname = 'Bloggs'
```

Code Box 2: Resultant query

In many web languages, a critical vulnerability is the way in which the query string is created. An example is shown in code box 3.

```
var SQL = "select * from users where username = " + username + " and password = " + password + "";
```

Code Box 3: Code showing a SQL injection vulnerability

If the user specifies the input shown in text box 2, the 'users' table will be deleted, denying access to the application for all users [Anley, 2002].

```
Username: '; drop table users--
```

Text Box 2: User input to delete the users table

An attack against a database using SQL Injection could be motivated by two primary objectives:

- To steal data from a database from which the data should not normally be available, or to obtain system configuration data that would allow an attack profile to be built. One example of the latter would be obtaining all of the database password hashes so that passwords can be brute-forced.
- To gain access to an organisation’s host computers via the machine hosting the database [Finnigan, 2002].

3. PREVENTION METHODS

SQL injection is a relatively simple technique and on the surface protecting against it should be fairly simple; however, auditing all of the source code and protecting dynamic input is not trivial, neither is reducing the permissions of all applications users in the database itself.

It is difficult to detect SQL injection with an audit of the SQL commands executed. A better
method is to audit the errors generated when the hacker is trying to gain access to the database. These error messages can be as useful to the hacker as they are to the database administrator building up database queries and stored procedures [Finnigan, 2003].

In the last few years, SQL injection attacks have been on the rise [Maor and Shulman, 2003]. Maor and Shulman outline research that has proved that suppressing error messages - going back to the “security by obscurity” approach [Finnigan, 2003] - cannot provide a real solution to application level risk but can add a measurement of protection. Security by obscurity tries to reduce the unnecessary information from being sent back to the client. Error messages can be used to determine information such as the database type and table structure [Cerrudo, 2004] [Anley, 2002] [Litchfield, 2001]. Applications have still proven to be vulnerable despite all efforts to limit information returned to the client. There are a few applications that have been developed by companies in an effort to provide a solution to this problem. Some have been outlined below:

- SecureSphere [Imperva Inc., 2005] uses advanced anomaly detection, event correlation, and a broad set of signature dictionaries to protect web applications and databases.
- ModSecurity is an open source intrusion detection engine for web applications, which may provide helpful tips on how to detect SQL injection. [Ristic, 2005] has developed ModSecurity for Java which is a Servlet 2.3 filter that stands between a browser and the application, monitors requests and responses as they are passing by, and intervenes when appropriate in order to prevent attacks.
- AirLock combines secure reverse proxy with intrusion prevention, content filtering, user authentication enforcement, and application-level load balancing and failover [Seclutions, 2003].

There is data that shows that injection flaws has been sixth in the top ten vulnerabilities for the past two years and that 62% of web applications are vulnerable to SQL injection attacks [OWASP, 2004] [WebCohort Inc. 2004].

[Microsoft, 2003b] offers the following tips for preventing SQL injection:

1. Validate all user input before transmitting it to the web or database server. Authentication on the client side is vulnerable to SQL injection.
2. Permit only minimally privileged accounts to send user input to the server.
3. Run SQL Server itself with the least necessary privileges.

[Kc, Keromytis, and Prevelakis, 2003] provide evidence that there has been a lot of development and research in the area of how to detect and test sites for SQL injection.

The presentation by [Hotchkies, 2004] at a Black Hat USA 2004 convention outlines automated blind SQL injection techniques. He mentions that string comparison is suitable for error based SQL injection but not blind SQL injection. He also mentions that there are three kinds of SQL injection:-

- Redirecting and reshaping a query involves inserting SQL commands into the query being sent to the database. The commands allow a direct attack on the database.
- Error message based SQL injection makes use of the database error messages returned to the client. The messages provide clues as to the database type and structure as well as the query structure.
- Blind SQL injection which involves a lot of guesswork and thus requires a larger investment in time. The attacker tries many combinations of attack and makes the next attack attempt based on their interpretation of the resulting html page output.

[Microsoft, 2003a] provides a good background into the problem of SQL injection. It puts the whole problem into context. The site provides explanations of the components of SQL injection strings and the syntax choices. The examples include SQL injection attacks, creating a secure data access component using Java’s regular expressions.

[Beyond Security Ltd., 2002] provides concise examples of SQL injection and database error messages as well as methods on how to prevent SQL injection.

The white paper by [Anley, 2002] covers research into SQL injection as it applies to Microsoft Internet Information Server/Active Server Pages/MS SQL Server platform. It addresses some of the data validation and database lockdown issues that are related to SQL injection into applications. The paper provides examples of SQL injection attacks and gives some insight into .asp login code and query error messages used to exploit databases.

[Finnigan, 2003] goes through worked examples of SQL injection attacks in his white paper on Detecting SQL Injection in Oracle. It focuses on detecting SQL injection by auditing the error message log files. It attempts to highlight the fact that during a hacking attempt, the error messages leave a trail that can help expose the vulnerabilities of the database being attacked.
[Spett, 2002] of SPI Dynamics presents a paper with describing SQL injection in general. It goes through some common SQL injection techniques and proposes a solution to the problem. The paper provides a list of database tables that are useful to SQL injection in MS SQL Server, MS Access and Oracle. It also provides examples of SQL injection using select, insert, union, stored procedures. The examples work with a web service that returns information to the user. This paper deals primarily with the structure of the SQL injection commands and guides to overcoming possible errors returned by the database.

It should be noted that SQL injection can still occur if there is no feedback to the client. So, one could create a new valid user in a database without receiving errors and then log on.

[Grossman, 2004], CEO of White Hat Security, Inc., in his presentation at the Black Hat Windows Security 2004 convention, outlines the challenges of scanning web application code for vulnerabilities. He points out that the scanner is restricted to looking for classes of vulnerabilities such as SQL injection or cross site scripting. The reason for this being that the benefit of known security issues is lost because the remote scanner does not have access to the source code.

There is no way to provide everyone with the minimum privileges necessary. Thus the paper explores some simple techniques in extracting the logging and trace data that could be used for monitoring. Finnigan, 2003 is an extension of a two-part paper on investigating the possibilities for an Oracle database administrator to detect SQL injection. This paper provides many scripts on SQL injection and extracting logs [Finnigan, 2003].

4. SYSTEM OVERVIEW AND PROJECT GOALS

- Analyse the structure of SQL query commands
- Build a parser that will check allowable patterns of SQL statements
- Construct a list of common SQL injection commands
- Create a proxy server that will alert the database administrator of possible SQL injection commands.
- Prevent a SQL injection attack to a database using this proxy server.
- Prove that SQL injection can be prevented using the filter developed to work on the proxy server.
- Provide sufficient logging to allow the user to isolate security holes.
- Produce a list of best practices for Database Administrators and Software Developers with respect to preventing SQL injection.

5. DESIGN

This project aims to eliminate the possibility of SQL injection by the use of a proxy server, which will be placed in between the two communicating devices. This will allow for the filtering of possible SQL injection attempts.

![Information flow diagram](image)

Figure 1: Information flow diagram

The information flow diagram in figure 1 shows the flow of information between a TDSProxy server within the domain of this project and the other entities and abstractions with which it communicates. The diagram helps to discover the scope of the system and identify the system boundaries. The system under investigation (TDSProxy) is represented as a single process interacting with various data and resource flow entities via an interface. As can be seen from the diagram, the web application provides the query to TDSProxy which in turn provides safe queries to the database and attack reports to the Database Administrator. The response from the database is routed back to the web application through TDSProxy. Should the need arise, log files in the database application provide information for auditing purposes at a later stage.

The design and implementation steps made use of the Rational Unified Process (RUP) with the aid of Unified Mark-up Language (UML). This iterative process started off with a simple application and developed into a more complex system in subsequent iterations. The reason for using this methodology was to overcome problem areas in segments. Once the basic concept was conceived and implemented, more advanced features were added to flesh out the software used for this proof of concept project.

The web application is where the queries are formed from the input parameters. These queries are sent to the database through TDSProxy. The bulk of the system operations take place at the TDSProxy. When the TDSProxy has filtered the query, the clean query is sent to the database server. Figure 2 is a
simple diagram graphically showing how the incoming requests are filtered and only clean queries are passed on to the database for processing. For security reasons, the proxy server will sit on the same machine as the database.

The diagram in figure 2 shows all the components in the high level view of the system. The web interface is the tool used by the client to send requests to the database. The web application is pointing to TDS proxy server so that all requests and responses must go through TDSProxy.

The client’s web application request triggers the formation of the SQL statement which uses the input parameters of the web form to create the correct SQL statement. This SQL statement is then sent to TDSProxy. When the SQL statement is received, it is first filtered. Only clean SQL statements are then sent to the database. The database processes the request and sends its response through TDSProxy. TDSProxy in turn sends the response to the web application for processing to produce the correct view for the client.

The flowchart in figure 3 focuses on the internally driven processes as opposed to external events. The action states in the diagram represent the decisions and behaviour of the processing. Figure 3 captures the actions performed at system start-up and run time.

TDSProxy loads a configuration file at start-up. This file contains, filter settings and options as well as the settings required for the passing of data to the correct destination. Once the system has started, it is able to start receiving data from the client. When data is received from the client, the payload is analysed. If the payload contains a SQL query, the query is logged and then filtered. If the filter process finds that there is a potential attack, the attack is logged. After logging the attack, the attack information is sent via UDP to the DBA. A false query is sent to the database and the response returned to the client. If the filter process did not pick up an attack, the query is sent to the database and the database response is returned to the client. If the payload does not contain a query, the data is simply passed on to the database.

The operations and methods of the system transform the query from one state to another depending on what route the information is flowing. These changes are shown in figure 4.

The raw string becomes part of the query string through processing at the client interface. This
happens when the input parameters are selected from the client interface and inserted into the hard coded query. The query may be formed at the client side or the parameters may be passed to the web application server.

Once the SQL query has been formed, it is sent to TDSProxy where it is analysed for SQL injection. The query is logged and then filtered for SQL injection.

If the query contains SQL injection, the attack is logged, the dangerous SQL is discarded, the DBA is notified via a UDP alert and a false query is sent to the database. The database response is then relayed to the client.

If the filtered query does not contain SQL injection, the query becomes a database query and is sent to the database. The database response is then relayed to the client interface through TDSProxy.

6. CURRENT STATUS

The implementation was done iteratively, starting off with an application that piped text (the TCP payload) through a proxy server. This was tested using a powerful networking tool called NetCat [Vulnwatch, 2005].

The proxy server was then improved to connect to the database using a connection string. There was a problem at this stage of the development. The default setup configuration of MS SQL Server 2000 allows Windows authentication only. This needed to be changed to windows and SQL server authentication in order to overcome the login error. It was thought that there was something wrong with the code when in actual fact, it was a database setting. With the correct username, password and rights, the database was manipulated by entering the SQL text on a NetCat client instance.

The next step involved sending hard coded SQL queries to the database at start-up of the application. This confirmed that the username, password and privileges were correct.

An attempt at using a Microsoft Access 2003 data access page as the client was unsuccessful and produced many login errors. When setting up the data access page, Access 2003 only accepted the use of an actual machine name and not its IP address. For testing purposes, a direct connection to the database was set up and querying the database through the data access page was possible. The next step was to be able to route the login through TDSProxy so that the database would ‘think’ it was talking to an Access data access page. However, when trying to connect to the database from the data access page through the proxy server, there was a problem with the connection string. The database kept returning an error message saying that the connection was refused because it was not associated with a trusted database. This problem was overcome by hard coding "trusted_server = true" into the data access page’s connection string. The login errors continued.

The database kept sending back reset packets. There was no apparent reason for it not being able to log on after the data was being routed. The packet data was altered so that the source and destination ports and IP addresses made TDSProxy seem totally transparent. The first three login packets were forged from a successful login without TDSProxy. However, this made no difference and an alternative client tool was sought. The possibility of port or IP number mismatching was eliminated by continuing the development on the same machine.

The querying client made use of OSQL, a tool that comes with MS SQL Server 2000. This tool, along with packet sniffers Ethereal [Ethereal, 2005] and Packetyser [Network Chemistry, 2005] allowed for the development of the SQL extracting method. This was done by analysis of the TDS protocol and lead to the extraction of the query in the query packet sent to the database after the login challenge [Bruns, Wheeler, Schaal, Ziglio et al., 2005].

With TDSProxy now able to capture the query sent in the TDS query packet, a vulnerable ASP application was developed [Anley, 2002]. The ASP page was hosted on a remote machine and connection to the database came through the proxy server. This application allows the user to enter SQL injection text into the input parameters and manipulate the database.

The next step involved creating the filter which made use of powerful regular expressions. The filter uses SQL injection signatures which are made up of a black list, white list, gray list and pattern matching list. The filter is able to report whether the SQL query text matches any of the given signatures.

At all stages of development, there is extensive logging of the queries captured. This helps with the debugging. SQL injection attacks are logged along with the signature that caught the attack. With the aid of the log files generated by TDSProxy and the database log files, the DBA can ascertain which database is being attacked. The DBA can also discover which web server or web page the attacker is using. The value of this is that the security holes can be patched and the database protected from further attacks.

Alerts are sent via UDP to the database administrator with the SQL injection query, the name of the machine hosting the web application and a timestamp. This will allow the DBA to block further
injection attacks from a particular user by checking the database log file which should contain the IP address of the person who sent the query at that time. The filter method made use of black, white, gray and pattern matching signatures. When filtering was turned off, the average processing time of TDSProxy was reduced from 0.256845 milliseconds to 0.002469 milliseconds. This was for a set of 5000 queries of varied length and structure. The total signature set is 190.

Timing the latency of TDSProxy was done by subtracting the time that the database spends processing the query from the roundtrip time for a client query and response. The roundtrip time was calculated as the query enters and leaves the proxy sever on the client side only. The database processing time was calculated by timing the query and response time on the database side.

The time taken to process queries seems to be negligible given the default MS SQL Server 2000 login timeout time is 4 seconds and the default query timeout time is 0 seconds.

7. FUTURE WORK
The order of filtering may have a performance impact too. This will be investigated by changing the order that filter uses the signatures.

Timing individual query execution time from a webpage will provide useful information on the impact of the TDSProxy on web interface usage.

The project could be extended to handle other databases such as MySQL, Oracle and Postgres as well as other operating systems. A further extension of the project could involve an investigation into the performance impact of the proxy server on data transfer [Beynon, Sussman and Saltz, 1999].

8. CONCLUSION
SQL injection is a relatively simple technique and on the surface protecting against it should be fairly simple. Auditing all of the source code and protecting dynamic input is not trivial, neither is reducing the permissions of all application users in the database itself. Given the research done in the area of using other methods of prevention (Kc, Keromytis, and Prevelakis, 2003) and the fact that there is a finite set of words in the SQL, it is possible to develop a filter to prevent SQL injection.

Checking through log files, making sure that code is perfectly secure and relying on the least privileges principle does not seem sufficient. Detecting SQL is not as useful as preventing it. It is difficult to detect attacks and again, an audit of log is required. The use of packet sniffers does not allow for the prevention of damage as the packets collected do not allow for the removal of malicious SQL query statements. Is it viable to develop auditing software that will require a large amount of resources for computation and storage?

TDSProxy is a feasible solution to preventing SQL injection. The filtering process seems to provide a negligible overhead. This part of the project does however require further investigation.

9. REFERENCES


