Anna Schmult

Professor Justin Price

CSC 3080-100

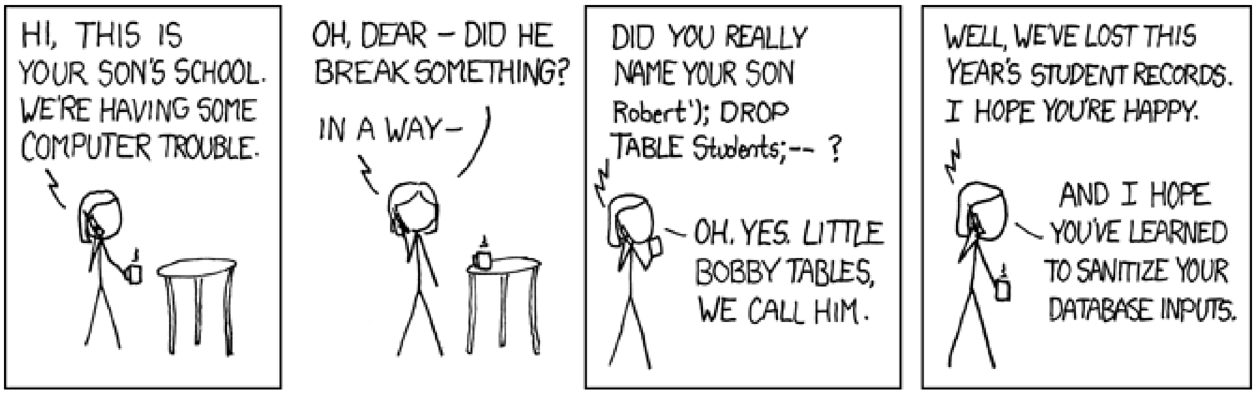
02 May 2022

Little Bobby Tables: SQL Injection and its Mitigation

Introduction

SQL injection poses a serious threat to websites and web apps worldwide. In 2017, the Open Web Application Security Project (OWASP) rated SQL injection as number one in their Top 10 ranking, where they calculate the ten vulnerabilities that pose the greatest threat to websites. SQL injection poses a high risk for a number of reasons: SQL is easy to exploit, very common across applications of all types, and its exploits have the potential for significant damage, including the theft or deletion of data. Luckily, SQL injection is also reasonably straightforward to mitigate.

SQL injection occurs due to a failure to sanitize user inputs. Sanitization of an input ensures that anything entered or provided by a user is treated as a string, integer, etc., and is not executed as a script. In normal execution of a user-specified SQL script, a user provides some form of input (most commonly through a text box on a website), and that input is used as a portion of an intentional query. SQL injection occurs when that user enters a malicious string, which, when included as part of the intentional query, causes one or more other unintentional queries to be run, or alters the structure of the intentional query. SQL injection is most often used to gain access to data which a malicious actor is not authorized to see, but can also be used to modify, add, or delete data. This has the potential for serious consequences, as demonstrated below:


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

This paper will aim to discuss the various threats and vulnerabilities posed by SQL injection, with the goal of helping those involved in web or application development, as well as those more broadly involved in information security, to gain an understanding of how SQL injection is carried out, as well as how and why to mitigate SQL injection. It will focus on the technical details of SQL injection, as opposed to the historical details of its use.

Background

In order to understand SQL injection, one must first have a basic understanding of SQL, or Structured Query Language. This paper will assume the reader has a working understanding of SQL, but a brief review will be provided. A SQL `SELECT` query (to view data) is structed in the following format:

```
SELECT [column names] (or a * to return all columns)
FROM [table name]
WHERE [conditions];
```

`UPDATE` queries (to modify existing records) and `DELETE` queries are structured in largely the same manner, where the specified operation will be performed on all records which match the conditions specified in the `WHERE` clause. A drop query may be used to delete a table or schema, given its name. More advanced queries may have additional types of clauses, which will be addressed in this paper where they are relevant. Each query, no matter how complex or how many clauses, ends in a semicolon.

The `WHERE` clause may have multiple conditions, which will be related using boolean logic, such as `AND` or `OR` statements, with parenthesis being used for complex logic. Each `WHERE` condition is structured as `[column name] [operator] [value]`. Operators can be simple, like =, >, etc., and any operator may be negated with `NOT`. The value provided, if it is a string, must be wrapped in either single or double quotes. An integer or float may be in quotes but doesn't have to be. Conditions are not separated by commas. The value of a `WHERE` condition may be generated by a subquery, which is a logically separate query.

In addition to the structure of queries, there are several other facts about SQL which are important to know when dealing with SQL injection:

- A `WHERE` clause will return a boolean true/false value for each record to decide if it should be included in the result. Practically, this means that adding "`OR 1 = 1`" as a condition in the `WHERE` clause will cause all records to be returned.
- SQL queries can be written over multiple lines or in a single line. A multi-line versus single-line SQL statement will have the same output, as line breaks are ignored in evaluation. In this paper, queries will be written in the multi-line format to help readability. In most practical programming, queries use single line formatting.
- A single-line comment is started with "`--`" (a double dash) and runs to the end of the line, and a multi-line comment is started with "`/*`" and ended with "`*/`". Curly braces may also be used for a multi-line comment. This paper may use single line comments even when using multi-line query format.

- Data from different tables may be returned in a single query using a query with a JOIN or UNION clause. A JOIN type query will combine data based on one or more conditions in the records (like an ID), effectively adding columns from table A to table B. A UNION type query will concatenate the resulting records of multiple SELECT statements together. The different SELECT statements must have the same number of resulting columns with matching data types, but don't have to be related by any criteria.

In order to put together a SQL query that is useful in the real world, a developer will generally slot user input into a pre-structured query, or they may allow the user input to dictate which pre-written query structure will be used, such as selecting if an ORDER BY clause will be used.

As an example, consider a website written in PHP for an online shop, which includes a search box. When the user clicks the submit button, the contents of the box are put into a PHP variable named $search_term. (Red text indicates user input, which will be continued throughout the paper.) Then, a SQL string will be put together as:

```
SELECT *
FROM products
WHERE product_name LIKE ‘%$search_term%’
```

The LIKE operation performs string matching, the single quotes enclose the string value for the WHERE condition to check against, and the % specifies that there can be any sub-string in that position. In this case, the query is looking for products where the product name has the search term located somewhere in it. This is how a malicious actor is able to implement SQL injection- their input gets slotted into an intended query, generally as a value in part of a clause.

History & Notable Attacks

SQL injection has a long history, with the first recorded publication about it in 1998 by Jeff Forristal, in which he described it as piggybacking an unauthorized command onto an authorized one. At the time, when he reported the flaw to Microsoft, as it impacted their SQL Server, they did not see it as an issue. Unfortunately, this was largely in line with the lackadaisical attitudes to security at the time, with practices like sending passwords over the internet in plaintext being common.

The first massive breach due to SQL injection came in 2007, when, according to court documents, Russian hackers used SQL injection to gain initial entry into 7-Eleven's servers. 7-Eleven ran its own transaction servers for about 2,000 ATMs in its stores, and the hackers were eventually able to steal information from anyone who had used one of those ATMs during a 2-week period in September 2007, including pin numbers. This information was used to create fake debit cards, which were used to withdraw large amounts of cash, much of which was laundered to Russia. The exact details of the technical implementation of the hack, as well as exactly how much money was stolen, are unclear.

Two of the most amusing instances of attempted SQL injection are a Polish business legally named "`Dariusz Jakubowski x'; DROP TABLE users; SELECT '1`" in an attempt to combat spam bots, and a Swedish voter in 2010 who included "`DROP TABLE VALJ`" as part of their handwritten write-in ballot. In the same election, another handwritten ballot attempted to invoke and run JavaScript. Both attempted to exploit failed user input sanitization, and both were unsuccessful.

Methods of SQL Injection

SQL injection is commonly seen in major cyberattacks as an initial attack vector. SQL injection is used to steal authorization information, which then allows the attackers to gain access to more of the database or more of the victim companies digital infrastructure. In other instances, it is the entire attack, with data being directly stolen or modified via SQL injection. Once attackers have stolen this data, it can be sold and/or used for identity theft. However, some SQL injection attacks are not financially motivated, and may aim to gain data for espionage, or are simply to make a political or social statement.

When used in the wild, SQL injection comes in several flavors, which classify the attack largely based on how the attacker receives information back from the query they send, such as whether they get results directly or indirectly. Major categories and sub-categories are below, although they are not necessarily inclusive of every possible type of SQL injection attack.

Classic or In-Band SQL Injection: The attacker submits a query and receives direct feedback to that query through the same channel of communication. This is the easiest, fastest, dirtiest, and most common form of SQL injection.

Modified construction: A broad category often involving inserting or commenting out clauses or parts of clauses, leading to a query constructed differently than the developer intended, sending the attacker more and/or different data than intended.

Union-based: The attacker piggybacks additional queries behind the query intended to be run by the developer. This can result in more/different data than intended being returned. Alternatively, the piggybacked query may be an alteration, where data is deleted, modified, or added, which returns no feedback.

Error-based: The attacker intentionally creates a SQL query with incorrect syntax and gets an error message in return. If the error message does not make it all the way from the server to the attacker, but rather the client only gets a generic message letting them know something has gone wrong, it is a form of blind SQL injection. This helps reveal the structure of the database or the code which puts together the query. This is generally done through modified construction.

Blind or Inferential SQL Injection: the attacker does not receive the direct results of the query run by the attacker, and they must instead infer the direct results based on some other action the attacker causes to happen. This is generally used to learn about the structure of the database (such as table or column names, size, etc.), or to determine the presence of some particular piece of data. It may also be used to discover particular data values, but the process will be slow. The attacks can be carried out through modified construction and/or union implementations.

Boolean-based: The malicious part of the query generally boils down to a True/False value, and the rest of the query causes a different action to be taken based on that value, such as sorting the results of the intended query in ascending versus descending order.

Time-based: The malicious part of the query generally boils down to a True/False value, and the rest of the query causes the server to wait a given amount of time based on that value before returning the results of the intended query.

Blind SQL injection generally uses a malicious query portion which returns a True/False value, although it can boil down to a single value of another type, which is used to find a similar value. For instance, one could count the number of network administrators, and then return a product with a price starting with that number. However, this is a much less common implementation.

Out-of-Band SQL Injection: the attacker causes some sort of response that delivers feedback to them via a different communication channel, generally an HTTP or DNS request, which goes to an attacker-controlled server. It is generally implemented in a blind fashion, where the feedback is not the direct result of the query. Out-of-band SQL injection relies on the server being attacked having HTTP or DNS requests enabled, making it uncommon.

Attack Implementations and Strategies

Condition to Return All Records (Modified Construction Classic SQL Injection)

Adding a condition to a `WHERE` clause which will cause all records to be returned is a very common SQL injection technique, which is almost always used in conjunction with other techniques. When used alone, it cannot be used to access unauthorized data, but it is a good scenario to understand the details of general implementation. Returning to the shop example from the Background section, say a user is searching on price, and puts 5 in the search box. The SQL query put together in this instance will be:

```
SELECT *
FROM products
WHERE product_price = ‘5’;
```

As intended, they will see all products with a price of 5.

Now, let’s say the user wants to see all products. There’s not an easy way to do so, so they decide to do so with SQL injection. First, they put OR 1 = 1 in the search box, which gives the query:

```
SELECT *
FROM products
WHERE product_price = ‘OR 1 = 1’;
```

However, in this case, the query will be searching for a product with a price of the string literal “OR 1 = 1”, so no records will be returned. Depending on database implementation, an error may be thrown. Instead, the user will have to put 0’ OR ‘1’ = ‘1 into the search box, resulting in:

```
SELECT *
FROM products
WHERE product_price = ‘0’ OR ‘1’ = ‘1’;
```

This query will return all products, because for every row, the `WHERE` clause will return True.

This simple example is a good demonstration of the importance of quotes. The quotation marks (or other syntax) must be exactly correct, which requires a good knowledge of SQL, and often some trial and error. In this case, the user would not be able to know without testing if the code assembling the SQL query put quotes around the price, because price is an integer, and integers can be put in quotes but do not have to be.

Another implementation of the ‘Or 1 = 1’ type of condition would be in allowing an unauthorized login. Say there is a login page with a username field and a password field, which generates the query:

```
SELECT *
FROM users
WHERE username = ‘$username’ AND password = ‘$password’;
```

Putting, say, aschmult into the username field and ’ OR ‘1’ = ‘1 into the password field would result in:

```
SELECT *
FROM users
WHERE username = ‘aschmult’ AND password = ‘’ OR ‘1’ = ‘1’;
```

Now, assuming that a valid username is entered, the query will always return the user's information, so the login will always be successful.

Commenting Out Clauses (Modified Construction Classic SQL Injection)

Commenting out clauses is another bread and butter technique used for SQL injection. It is generally used to eliminate those pesky `WHERE` clauses that keep users from seeing data they are not authorized to see. In the store example, say there are enterprise products which are not publicly visible, because the prices charged are considered sensitive information. In this case, the query put together by the price search would be:

```
SELECT *
FROM products
WHERE product_price = '$price' AND enterprise = '0';
```

The curious user repeats the input of 0' OR '1' = '1 from the previous example, giving the query:

```
SELECT *
FROM products
WHERE product_price = '0' OR '1' = '1' AND enterprise = '0';
```

However, SQL servers evaluate AND logical operators before OR operators, so this will still not override the enterprise condition. Instead, we must comment it out. To achieve this, the user enters 0' OR '1' = '1';-- which will lead to the query:

```
SELECT *
FROM products
WHERE product_price = '0' OR '1' = '1';--' AND
     enterprise = '0';
```

The blue text is interpreted by the server as a comment, and thus ignored, so the enterprise condition has been circumvented. Note that before starting the comment, the user ends the query with a semicolon. (In this case, the double dash comment is a single line comment, which will comment out to the end of the line, but the query would, in a real world implementation, be a single line query, and is only written over multiple lines in this paper to improve readability.)

However, the ability to run this query is entirely dependent on how the developer structured the original query. If in the `WHERE` clause the enterprise condition was put before the price condition, there would be no way for the SQL injection to comment out the enterprise clause. This is another instance where the malicious actor will need to use trial and error to determine if a given query is feasible to run.

Union Based Attack (Classic SQL Injection)

Simple modified construction may be able to display a larger number of records than intended, but it often is unable to properly handle data from different tables than expected, because the developer only wrote handling with the intended target table in mind. However, if results from a secondary table can be formatted similarly to the results from the intended table, the results can be concatenated together using a `UNION` clause.

A union based attack requires knowledge of the structure of both the intended and unintended tables (the unintended table being the malicious actor's attack target). Let's say we have the following database structure:

Products		Users	
Product Name	String	First Name	String
Category	String	Last Name	String
Price	Integer	Phone number	String
Purchase Cost	Integer		

The main thing to overcome here is the different number of columns from the two tables. The columns from the different SELECT statements should have similar data types, but SQL is able to handle integer and string data together. However, it can't deal with different numbers of columns from the two SELECT statements. There are two possible solutions- you could only select certain columns from the wider table, or you could "bulk" the narrower table by adding a constant value. Because you only have access to one of the SELECT statements (the one about the unintended table), you will need to choose your approach based on which table is wider. Again, this is all information that would probably need to be discovered through trial and error.

Assuming that the products table is the intended table, we will use the input in red and blue:

```
SELECT *
FROM products
WHERE price = ‘5’
UNION
SELECT first_name, last_name, phone_number, ‘Blank’ as blank
FROM users;--’;
```

This query will be sent to the database, which will send to the client backend:

product_name	**category**	**price**	**purchase_cost**
Shoes	Clothes	15	10
Shirt	Accessories	5	3
Anna	Schmult	610-519-4500	Blank
Steve	Donchez	610-519-4500	Blank

Although SQL will be able to deal with the different data types in the third and fourth column, the handling written by the developer may not deal with it so well. For instance, it is likely that the developer might use a format function to extend the prices to 2 decimal places. In the opposite scenario, a string-specific function (like to standardize capitalization) could be used. These would cause errors specific to the language the website backend was written in (as opposed to SQL errors). As a result, it might be necessary to run the query twice, the first time getting the first and last name plus too padding rows, each padded with an integer, and the second time getting the phone number and three padded rows, the first with a string and second and third with an integer.

Multiple Independent Queries (Classic SQL Injection)

An alternative way to run multiple queries is to simply put them together back to back, separated by a semicolon. This is generally counted as a form of union SQL injection, despite not using a `UNION` clause. However, one downside, when compared to a true union attack, is that the user has no way to harness the results of the extra query. The developer only wrote handling for a single query, so if the second query returns results, they won't go anywhere, or may result in an error. In a true union attack, the results from the two `SELECT` statements are concatenated into one result table by the SQL server, so from the perspective of the website backend, there only ever was a single query and a single set of data returned. As a result, multiple independent queries are generally used to run alteration queries, which don't return anything anyway.

Let's say that our malicious actor is trying to cause problems, by deleting the records of all the products the shop has available. In the product search, the actor will enter 5' DROP TABLE products;-- which will lead to the query:

```
SELECT *
FROM products
WHERE price = '5';
DROP TABLE products;--';
```

From the perspective of the SQL server, this is 2 separate queries that come in one immediately after the other. It will result in the product table (both the data in it and its structure) being deleted. Other types of alteration queries add new records, modify existing records, delete one or more specific records, truncate a table (delete the data but not the structure), or even drop an entire database.

In this case, a semicolon and comment are needed at the end, to deal with the closing single quote around the value in the price condition of the WHERE clause. This would also be needed if the enterprise condition was added after the price condition. If the value in the price condition wasn't wrapped in single quotes, the semicolon and closing quote would not be needed, which is another example of a place trial and error may be needed.

<u>Blind Error Generation (Boolean-Based Blind SQL Injection)</u>

Intentionally generating an error can be a very good way for an attacker to learn about the structure of a database or the code that generates queries and handles responses. It is important to note here that errors may be generated in three places- the code that puts together the query (client backend), the SQL server itself, and the handling of the query results (also the client backend). All three may be useful, but the exact location of the intended error depends on the specific attack.

In all cases, the error message will either be dumped to the screen or will be stopped before it reaches the screen. If the error message (like the stack trace) doesn't reach the client frontend, they will instead receive some sort of generic message saying there's been an error. In this case, it is a form of blind error generation, because the malicious actor does not see the direct results of their query. Intentional error generation is generally the way that blind SQL injection gets feedback. If the malicious portion of the query sent to the server is True, the server will send back results from the intended query. If the portion is False, no results will be returned, and the handler will throw an error.

An easy way to throw an error is a condition in the `WHERE` clause stating `‘1’ = ‘2’`. If the condition is triggered, no results will be returned. Assuming that the result handling is expecting a result no matter what, the application server will throw an error.

Say that the malicious actor wants to know what version of MySQL the server they are targeting is running, perhaps to figure out what syntax they will need to use for a later portion of the attack. This can help them figure out what sorts of vulnerabilities it may have. The query to determine version is `SELECT @@version`, which will return the entire version, such as `5.7.36-39-log`. However, MySQL conditional logic can only deal with one number at a time, so to get only the major version, we use the functions:

```
SUBSTRING(string, starting character, length)
INSTR(main string, substring to search for)
```

Substring will return the substring specified, and in string will return the character position of the first occurrence of the substring in the main string. Both are indexed at 1. The query to find just the major version is:

```
SELECT SUBSTRING(@@version, 1, INSTR(@@version, '.') - 1)
```

In this case, the query would return `5`.

Next, we have to package this up into a query which will trigger an error. In our shop example, we can do:

```
SELECT *
FROM products
WHERE price = ‘5’ OR substring(@@version, 1,
     INSTR(@@version, '.') - 1) = 5;--’;
```

This query will return a product list if the SQL server version is 5, but if it isn’t 5, there will be no products returned, which will trigger an error. Obviously, this shop example isn’t entirely realistic, because a search box will just about always have some sort of handling for there being no records returned, but the principle stands.

Standard Error Generation (Classic SQL Injection)

Knowing information about the SQL instance being targeting can be incredibly helpful to an attacker for crafting the most efficient attack possible. In addition to learning about the server, the attacker may also use errors to learn about the structure of the SQL database, particular to learn the names of entities. Knowing names of schemas, tables, and columns is crucial to extract data from those structures later.

It’s important to note that at this point, SQL syntax starts to vary significantly between versions – for instance, in researching this paper I ran into lots of issues with MSSQL syntax versus MySQL syntax. I have tried to use entirely syntax that is valid for a MySQL server, but some exact syntax may vary between versions.

We want to start by finding the name of the schema. To get all the user-defined schemas, we can use the query:

```
SELECT DISTINCT TABLE_SCHEMA
FROM information_schema.TABLES
WHERE TABLE_TYPE = 'BASE TABLE';
```

If we need to see them one at a time, we can add a `LIMIT = 1` clause, and after the first schema name we find, we can add a `WHERE` condition:

```
AND TABLE_SCHEMA NOT IN (‘[previous name 1]’,
        ‘[previous name 2]’)
```

By running this repeatedly, adding the most recently discovered schema name to the list each time, we will get a list of all the user-defined schemas.

Now that we have our schema names, we can use the same technique to find the unique values of `TABLE_NAME`, also from the `TABLES` table of `information_schema`, using a `WHERE` clause to limit the `TABLE_SCHEMA` to one of the schemas we just found, and we can iterate through all the schemas, getting us a list of all tables for each schema.

Finally, we need our columns names. We can obtain these from the `COLUMN_NAME` field of the `COLUMNS` table, with a `WHERE` condition limiting the `TABLE_NAME` field to one table at a time, again iterating through to find all the column names for each table for each schema.

We now have the code needed to determine the names of the entities on the server, but the malicious actor still needs to exfiltrate those names out. I found in my research that the most popular way of doing so is to attempt to convert the names, one at a time, to an integer. This will throw an error, which will have an error message including the value that was unable to be converted, which in this case will be the entity name being exfiltrated. However, this process seems only to work on MSSQL, as it seems that MySQL instead returns a null if it cannot make the conversion. There are no doubt other ways to throw an error, but research to that level of detail is beyond the scope of this paper.

It is important to keep in mind that the same information could be exfiltrated other ways, such as with a union attack. In the same manner, an error-based blind SQL attack could also be used to steal data, but it would not be highly efficient. Once an attacker has discovered the names of the entities in the server, and is able to write queries specific to them, it is generally more efficient to use another means of attack, assuming that one is available and exploitable.

Time-Based SQL Injection (Blind SQL Injection)

Time-based blind SQL injection is largely the same as boolean-based blind SQL injection, except that the information is exfiltrated by making the server either wait for 10-15 seconds before returning the results of the intended query, or returning the results immediately. On a MySQL (version 5 or above) server, this can be done with the `SLEEP(number of seconds)` function, implemented as:

```
WHERE price = 5 – IF([version] = ‘5’, SLEEP(15), 0)
```

In this example, [version] must be substituted with a string to find only the major version of SQL being run by the server, as demonstrated above. This will trigger the 15-second delay if the SQL version is 5, and either way, the if statement will always return either 0 or nothing, because the sleep function doesn’t return anything. The malicious actor can create a query using this exploit, then wait for the intended result to appear, seeing how long that takes.

Out-of-Band SQL Injection

Out-of-band SQL injection, as well as more generally getting a SQL server to send an HTTP or DNS request, is beyond the scope of this paper. It is also not very common, as most SQL servers have said features disabled. However, the attacks use the same tools, exfiltrating the information either in small pieces or all at once, by triggering a request to a server controlled by the malicious actor.

Second Order SQL Injection

Second order SQL injection, which is also quite uncommon, occurs when a malicious actor forms a string which is valid as a SQL query, and gets that string inserted into a table as data. The hope is that, at some point during later processing, the string will be run as a query.

Other forms of SQL Injection

Neither the examples of implementations listed in this section, nor the general types of SQL injection in the previous section, are complete lists which encompass every possible form of SQL injection attack. However, they encompass most of the building blocks which are used to implement a given attack. Malicious actors are constantly using these building blocks and combining them in different, novel ways, attempting to construct a specific attack which harnesses the exploits available, extracts the type of information they are seeking, and, ideally, will go undetected and unstopped.

SQL Injection Mitigation

Mitigation of SQL injection is largely the responsibility of application developers, as opposed to the developers of SQL itself. SQL injection, from the perspective of a SQL server, isn't a thing- the server is merely doing as it is told. In addition, a SQL developer has no way of knowing what sorts of queries another developer, or a user of one of their products, might want to run for a legitimate reason. As such, the SQL developers provide resources to client application developers, which they must use as they see fit and necessary to mitigate SQL injection attacks.

One of the most difficult parts of mitigating the possibility of a SQL injection attack is the fact that most mitigation strategies are applied at the individual user input instance. The mitigation measure generally occurs somewhere between when the user types their input into the box, and when the query is sent to the server, which means that it must be applied to every single instance of open-ended user feedback. If it is forgotten on even a single input field, and an attacker becomes aware of this fact, the entire server is vulnerable.

The mitigation measures will be presented in chronological order, starting with the user entering input, then that input being processed, the SQL query being assembled, the query being sent to the sever, and finally the result being handled when it is returned (or when no result is returned).

Input Validation

Input validation ensures that a user can only submit a form when the inputs they have put in each field match what is expected. At its simplest, it checks for the expected variable type, enforces minimum and maximum character counts, prevents non-standard characters, etc. More complex validation may use regex to check for expected patterns (such as in phone numbers or email addresses). The most complex validation may trigger queries to check that input isn't a duplicate of information already present, or that an email address or phone number actually exists.

Validating user input is a good first step, and is very important, but can't protect against SQL injection alone. Validation prevents all sorts of other issues, such as ensuring that someone can't put a string into an integer field, and it can *help* (keyword: *help*) to encourage valid, true input from users. That being said, validation isn't everything. At a certain point, validation starts assuming what users must want to input, which may or may not be true, particularly in edge cases. Validation could be created to look for certain characters or patterns that would be used in SQL injection, but this doesn't ensure that every case of SQL injection will be caught (particularly newer implementations). It also may have false positives, blocking cases where a user genuinely wants to use a character like a single quote or semicolon.

Escaping

Escaping prevents certain characters, such as a single quote or semicolon, from being interpreted as anything but a string literal, but placing a backslash in front of the character. Escaping is technically a safe way to protect against SQL injection, although it requires intentional handling, in that when escaped data is returned in a query result, it will still have the backslashes. For instance, it might read “`I didn\’t like these shoes`”, which must be handled to ignore the backslashes. If sloppily implemented (as it was on a website I once worked on), it can lead to backslashes being repeatedly added, resulting in long and obnoxious strings such as “`didn\\\\\\\\\\\\\\\\\\’t`”.

Escaping suffers from the issue of being able to be forgotten about, which can be devastating even if only forgotten once. This can be improved by creating a transparent layer between the client application and the SQL server, so that the necessary characters are escaped before the query reaches the SQL server.

For the reasons listed, escaping is not considered best practice, but it is, if done correctly every time, technically perfectly safe.

Next-Gen Firewalls

Next generation or Layer 7 firewalls are often able to check for SQL injection, largely by observing the queries being sent to the SQL server by the application server, which they correlate with POST requests made by the end user client to the application server. They look for known patterns common in SQL injection attacks, including a high concentration of queries using `UNION` clauses. Firewalls are often not able to detect SQL injection on the first instance that a malicious agent attempts it. However, malicious actors are never able to execute a SQL injection attack with only one query. There are always many queries, especially as they execute and trial and error. The firewall will pick up on a pattern of repeated suspicious traffic, coming from the same source, and may block that source. Hopefully, this happens while the malicious actor is in the trial-and-error phase, before any damage has been done.

A firewall alone is not enough to protect against SQL injection. That being said, the type of firm with a next-gen firewall is probably not the type of firm who is entirely relying on said firewall to mitigate SQL injection. A firewall can be a good measure to take to protect in the case that one or two fields, perhaps old and forgotten about fields, are not properly mitigating against SQL injection.

Parameterization

Parameterization is generally considered the industry-standard method of mitigating SQL injection, although it is generally used in conjunction with other measures for added protection. Parameterization is a framework, developed by SQL developers, which application developers may choose to use, which indicates to a SQL server specifically what portion of a query is meant to be what. This means that, if an attacker puts 0' OR '1' = '1 into an input box, the application server will tell the SQL server that that entire input is the value, stopping the server from breaking it up into a value and then a `WHERE` condition. An example of parameterization in PHP is:

```
$statement = $database->prepare("INSERT INTO REGISTRY
     (name, value) VALUES (:name_var, :value_var)");
$statement->bindParam(':name_var', $name_input);
$statement->bindParam(':value_var', $value_input);
```

Preparing the statement sets up a structure where the different parts of the query (query construction versus values) are clearly defined, and binding values to the prepared statement adds those values to the query.

Parameterization suffers from the same issue as most other forms of SQL injection mitigation, which is that it must be applied to each any every SQL query before it's sent to the SQL server. It cannot be applied between the application server and SQL server, as it has to be baked into the process of assembling the SQL query. However, in a serious application development environment, it is fairly simple to set up a rule to check to make sure that all SQL queries are parameterized, because a parameterized query can be easily identified by the `prepare` and `bind` functions (or the equivalent functions in other languages).

Database Setup & Permissions

The setup of a database can be used as a factor of protection, although is not nearly sufficient protection on its own. The use of encryption or hashing of sensitive data can reduce the utility of said data to malicious actors if they are able to steal it. Backups can serve to mitigate the damage of data being deleted or modified. Permissions can also help minimize attack damage, by making sure each user account has onlythe permissions it needs to use, and no more. For instance, the user account used by the application server to fulfill client requests could be prevented from querying the information schema table, which could help stop other SQL injection attacks which have slipped through the cracks elsewhere. However, security through obscurity, such as naming one's schemas, tables, and columns weirdly, is no security at all.

Error Handling

Errors, whether from the SQL server or the handling of the SQL response, can be a wealth of information if the error message makes it all the way to the user's screen. These messages can reveal information with the potential to be weaponized by a malicious agent, not just for a SQL attack, but for all sorts of other attacks, and can also confuse or frighten legitimate users. As such, errors should always be caught, the fact that an error occurred should be logged, and the user should be given a generic message that lets them know something has gone wrong. A full fledged error message should *never* make it to an end user. In addition to dealing with errors being thrown, handling for query results should always be prepared for no results to be returned, even if this is logically unreasonable.

Common Sense

A lot of information security is common sense. Common sense security includes regularly updating systems, patching vulnerabilities as soon as possible, enabling MFA, implementing strict access controls, and fixing problems as soon as they become apparent. Ultimately, common sense drives policies, which drive technical procedures.

Conclusion

SQL injection continues to be a very serious issue to the modern developer, but luckily, today it is largely an issue that can be mitigated. Obviously, there will continue to be new SQL injection vulnerabilities discovered, largely relating to convoluted ways to get a malicious string past mitigation measures. However, for the most part, the enduring threats from SQL injection are ultimately due to sloppy coding and poorly designed or poorly followed security practices.

Ultimately, this is in line with what information security professionals and malicious actors tend to already know- humans are the weakest part of any cybersecurity system. Fortunately, well-researched, well-designed, and well-enforced security practices, along with a company culture which promotes secure and safe design paradigms, are good countermeasures to human weaknesses. Furthermore, many tools exist in the world today which enforce the implementation of these practices, as well as verify the integrity of developed applications. By leaning into these practices and using this tooling, developers can avoid the risks of SQL injection in their applications, resulting in a safe and secure environment for users and their data.

Works Cited

A03 Injection - OWASP Top 10:2021. https://owasp.org/Top10/A03_2021-Injection/. Accessed 20 Apr. 2022.

Did Little Bobby Tables Migrate to Sweden? https://alicebobandmallory.com/articles/2010/09/23/did-little-bobby-tables-migrate-to-sweden. Accessed 19 Apr. 2022.

"Difference between JOIN and UNION in SQL." *GeeksforGeeks*, 7 Apr. 2020, https://www.geeksforgeeks.org/difference-between-join-and-union-in-sql/.

Garcia, Murilo. "Answer to 'Get Table Names Using SELECT Statement in MySQL.'" *Stack Overflow*, 18 Nov. 2014, https://stackoverflow.com/a/27001800.

goswamiijaya. "Exploiting Error Based SQL Injections & Bypassing Restrictions." *Medium*, 19 Jan. 2021, https://infosecwriteups.com/exploiting-error-based-sql-injections-bypassing-restrictions-ed099623cd94.

How to Tell What SQL Server Versions You Are Running. https://www.mssqltips.com/sqlservertip/1140/how-to-tell-what-sql-server-version-you-are-running/. Accessed 20 Apr. 2022.

"How Was SQL Injection Discovered? | ESecurity Planet." *ESecurityPlanet*, 25 Nov. 2013, https://www.esecurityplanet.com/networks/how-was-sql-injection-discovered/.

MySQL INSTR() Function. https://www.w3schools.com/sql/func_mysql_instr.asp. Accessed 20 Apr. 2022.

MySQL :: MySQL 8.0 Reference Manual :: 26.2 INFORMATION_SCHEMA Table Reference. https://dev.mysql.com/doc/refman/8.0/en/information-schema-table-reference.html. Accessed 20 Apr. 2022.

MySQL :: MySQL 8.0 Reference Manual :: 26.3.8 The INFORMATION_SCHEMA COLUMNS Table. https://dev.mysql.com/doc/refman/8.0/en/information-schema-columns-table.html. Accessed 20 Apr. 2022.

MySQL :: MySQL 8.0 Reference Manual :: 26.3.38 The INFORMATION_SCHEMA TABLES Table. https://dev.mysql.com/doc/refman/8.0/en/information-schema-tables-table.html. Accessed 20 Apr. 2022.

OWASP Foundation | Open Source Foundation for Application Security. https://owasp.org/. Accessed 20 Apr. 2022.

OWASP Top Ten Web Application Security Risks | OWASP. https://owasp.org/www-project-top-ten/. Accessed 20 Apr. 2022.

Poulsen, Kevin. "7-Eleven Hack From Russia Led to ATM Looting in New York." *Wired*. *www.wired.com*, https://www.wired.com/2009/12/seven-eleven/. Accessed 19 Apr. 2022.

Query Parameterization - OWASP Cheat Sheet Series. https://cheatsheetseries.owasp.org/cheatsheets/Query_Parameterization_Cheat_Sheet.html. Accessed 20 Apr. 2022.

SQL Comments. https://www.w3schools.com/sql/sql_comments.asp. Accessed 19 Apr. 2022.

SQL Injection. https://www.w3schools.com/sql/sql_injection.asp. Accessed 15 Apr. 2022.

"Types of SQL Injection?" *Acunetix*, https://www.acunetix.com/websitesecurity/sql-injection2/. Accessed 19 Apr. 2022.

"What Is SQL Injection | SQLI Attack Example & Prevention Methods | Imperva." *Learning Center*, https://www.imperva.com/learn/application-security/sql-injection-sqli/. Accessed 15 Apr. 2022.

"What Is SQL Injection - Examples & Prevention." *Malwarebytes*, https://www.malwarebytes.com/sql-injection. Accessed 19 Apr. 2022.