acunetix

SQL Injection Cheat Sheet

FOR DEVELOPERS

ASSUMPTIONS

In this cheat sheet, we will assume that:

- You are a developer or you know programming
- You have limited web application security knowledge
- You need to know how SQL injection attacks happen
- You need to know how to fix SQL injection issues in your code

GOALS

In this cheat sheet, you will learn:

- How do malicious hackers conduct SQL injection attacks
- How to fix your code that has SQL injection vulnerabilities
- How to avoid SQL injection vulnerabilities for the future

PART 1 What Are SQL Injection Attacks

SQL injections happen when:

- Your code uses unsanitized data from user input in SQL statements
- A malicious user includes SQL elements in the input in a tricky way
- Your code executes these SQL elements as part of legitimate SQL statements

SQL INJECTION FAQ

- **What SQL servers are affected by SQL injections?**
  All SQL servers may be affected by SQL injections: MySQL, MSSQL, Oracle, PostgreSQL, and more.
- **What programming languages are affected by SQL injections?**
  SQL injections may happen in any programming language.
- **What may be the consequences of an SQL injection?**
  An SQL injection may lead to data leaks but it may also lead to complete system compromise.
- **How common are SQL injections?**
  In 2020, SQL injections were found by Acunetix on average in 7% of web apps.
- **Do web application firewalls (WAF) protect against SQL injections?**
  No, WAFs only make it more difficult for the attacker to send SQL injection payloads.

SIMPLE SQL INJECTION EXAMPLE

YOUR CODE IN PHP:

```
<?PHP
  $userid = $_GET["userid"];
  $query  = "SELECT user FROM users WHERE userid = $userid;";
  $result = pg_query($conn, $query);
?>
```

ATTACKER REQUEST:

http://www.example.com/test.php?userid=0;DELETE FROM users WHERE 1

YOUR CODE PROCESSES THE FOLLOWING SQL QUERY:

```
$query  = "SELECT user FROM users WHERE userid = 0; DELETE FROM users WHERE 1;";
```

As a result, if the current user (current database user) has suitable permissions, the entire users table is cleared.

SQL INJECTION TYPES

TYPE 1: IN-BAND SQL INJECTION: ERROR-BASED SQL INJECTION

- The attacker sends a request designed to cause an error in the database server
- The server returns an error message to the attacker
- The attacker uses information contained in the error to escalate the attack
- This type of SQL injection is used to access sensitive information such as database type, file names, and more

EXAMPLE:

- **Payload:**
  http://testphp.vulnweb.com/listproducts.php?cat=1'
- **Result:** The web application displays the following error in the browser:

Error: You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near ''' at line 1 Warning: mysql_fetch_array() expects parameter 1 to be resource, boolean given in /hj/var/www/listproducts.php on line 74

TYPE 2: IN-BAND SQL INJECTION: UNION-BASED SQL INJECTION

- The attacker uses a `UNION` clause in the payload
- The SQL engine combines sensitive information with legitimate information that the web application should display
- The web application displays sensitive information

EXAMPLE:

- **Payload:**
  http://testphp.vulnweb.com/artists.php?artist=-1 UNION SELECT 1,version(),current_user()
- **Result:** The web application displays the system version and the name of the current user:

```
8.0.22-0ubuntu0.20.04.2
acuart@localhost
```

TYPE 3: BLIND SQL INJECTION: BOOLEAN-BASED SQL INJECTION

- The attacker sends many payloads containing expressions that evaluate to either TRUE or FALSE
- Alternating between the two, the attacker can draw conclusions about the database and its contents
- This type of SQL injection is often used to access sensitive information when the web application returns neither meaningful error messages nor the targeted data itself

EXAMPLE:

- **Payload 1:**
  http://testphp.vulnweb.com/artists.php?artist=1 AND 1=1
- **Payload 2:**
  http://testphp.vulnweb.com/artists.php?artist=1 AND 1=0
- **Result:** In both cases, the application behaves differently. The attacker now knows that the application is vulnerable to SQL injections.

TYPE 4: BLIND SQL INJECTION: TIME-BASED SQL INJECTION

- If the web application doesn't return errors and the returned information is the same for boolean-based payloads, the attacker sends a payload that includes a time delay command such as `SLEEP`, which delays the whole response
- The attacker draws conclusions from the length of response delays and repeats the process as many times as necessary with different arguments
- This type of an SQL injection is often used to check whether any other SQL injections are possible
- This type of SQL injection may also, for example, be used to guess the content of a database cell a character at a time by using different ASCII values in conjunction with a time delay

EXAMPLE:

- **Payload:**
  http://testphp.vulnweb.com/artists.php?artist=1-SLEEP(3)
- **Result:** The page loads with a delay.

TYPE 5: OUT-OF-BAND SQL INJECTION

- This type of SQL injection is possible only for some databases, for example, Microsoft SQL Server and Oracle
- The attacker includes a special database command in the payload – this command causes a request to an external resource (controlled by the attacker)
- The attacker monitors for attempts to contact the external resource, for example, DNS lookups or HTTP request logs of the external resource
- If there is a request coming once the payload is executed, this confirms that the SQL injection is possible
- The attacker accesses database information and can send it to the external resource

EXAMPLE:

- **Payload:**
  1||UTL_HTTP.request ('http://example.com/')
- **Result:** A request is made to *example.com* – you can monitor such requests if you control *example.com*.

PART 2 SQL Injection Defense

PARAMETERIZED QUERIES (PREPARED STATEMENTS)

- This technique is available in many programming languages
- Instead of forming the query by using string concatenation, the query string includes parameters
- The prepared statements library replaces these parameters with values supplied by the user, so that SQL commands and user input (parameters) are passed separately

PHP EXAMPLE

Using PHP Data Objects (PDO):

```
$dbh = new PDO('mysql:host=localhost;dbname=database', 'dbuser', 'dbpasswd');
$query = "SELECT column_name FROM table_name WHERE id = :id order by column_name desc";
$sth = $dbh->prepare($query);
$sth->bindParam(':id', $_GET["id"]);
$sth->execute();
$result = $sth->fetchColumn();
```

JAVA EXAMPLE

```
int id = Integer.parseInt(id);
String query = "SELECT column_name FROM table_name WHERE id = ? order by column_name desc";
PreparedStatement stmt = connection.prepareStatement(query);
stmt.setInt(1,id);
ResultSet results = stmt.executeQuery();
```

STORED PROCEDURES

- Use only if your programming language does not support prepared statements
- To avoid SQL injections, you must use prepared statements in stored procedures
- Available only for database engines that support stored procedures but most modern engines support them
- The query is prepared and stored in the database engine
- The application calls the stored procedure and passes variables to it

MYSQL EXAMPLE

Creating the procedure:

```
CREATE PROCEDURE example(IN suppliedId VARCHAR(8))
BEGIN
  SELECT column_name FROM table_name WHERE id = suppliedId;
END
```

Calling the procedure with id = 1:

```
CALL example("1");
```

SQL injection payload will not work:

```
CALL example("0;DELETE FROM users WHERE 1");
```

MSSQL EXAMPLE

Creating the procedure:

```
CREATE PROCEDURE dbo.example @id nvarchar(8)
AS
  SELECT column_name FROM table_name WHERE id = @id;
GO
```

Calling the procedure with id = 1:

```
EXEC database.dbo.example 1;
```

SQL injection payload will not work:

```
EXEC database.dbo.example 0;DELETE FROM USERS WHERE 1
```

PART 3 SQL Injection Detection

METHOD	TOOLS	KEY PROS	KEY CONS	RATING
Manual code review	Development environment	May improve the general quality of code	Unlikely to find SQL injections	★★☆☆☆
Manual penetration testing	Attack proxies	Able to find even very complex and rare types	Very time and resource intensive	★★★☆☆
Automatic code analysis (white box scanning)	SAST software	Can reach even code that is not used directly	Reports a lot of false positives and does not prove that a vulnerability exists	★★★☆☆
Automatic vulnerability scanning (black box scanning)	DAST software	Can run in any environment and at any development stage	Does not point to the issue in the source code	★★★★☆
Automatic vulnerability scanning with proof of exploit and grey box sensors	Acunetix	Proves that the vulnerability exists by showing data that should be restricted; points to the error in source code or bytecode (PHP, Java, .NET, Node.js)		★★★★★

Note: To improve detection, it is best to employ several methods at the same time. However, if you cannot afford it, go for the most effective method first.

APPENDIX: ADDITIONAL RESOURCES

- Acunetix: A general introduction to SQL injections
- Acunetix: A more detailed explanation of SQL injection types
- Acunetix: A more detailed explanation of blind SQL injections
- Acunetix: A detailed explanation of out-of-band SQL injections
- Acunetix: A practical example that shows how an SQL injection may lead to system compromise
- Acunetix: A practical example that shows how to analyze logs to discover an SQL injection attack
- Acunetix: A detailed article about preventing SQL injections in PHP
- Acunetix: An article about preventing SQL injections in Java
- Acunetix: An article about preventing blind SQL injections
- Pentestmonkey: Detailed SQL injection cheat sheets for penetration testers
- Bobby Tables: The most comprehensible library of SQL injection defense techniques for many programming languages

acunetix

We keep your web applications secure

Read more: acunetix.com/blog