SQL Injection Attacks

Sources:

- D. Stuttard and M. Pinto, *The Web Application Hacker's Handbook*, 2nd Ed., Wiley, 2011.
- W.G.J. Halfond, et al. *A Classification of SQL Injection Attacks and Countermeasures*, Intl. Symp. Secure Software, 2006.

SQL injection attacks (SQLIAs) are code injection attacks in which:

- User data is included in an SQL query.
- Part of the data is treated as valid SQL code.

They allow an attacker to submit SQL commands directly to a database.

It is a threat to web applications to incorporate user input into SQL queries to an underlying database.

The fundamental cause of SQL injection (SQLI) vulnerabilities is *insufficient validation of user input*.

Defensive coding practices can prevent SQLI vulnerabilities, but their application is error-prone.

There are many types of SQLI attacks and many variations of these types.

SQLIAs occur when an attacker changes the intended effect of an SQL query by inserting new SQL keywords or operators into it.

Injection Mechanisms

- *Injection through user input*: Typically form submissions via HTTP GET or POST requests
- *Injection through cookies*: A malicious client can tamper with a cookie's contents, which an application later uses to build SQL queries
- *Injection through server variables*: Such variables may contain HTTP and network headers and may be logged to a database without sanitation

  An attacker can place an SQLIA into a header
- *Second-order injection*: attackers seed malicious inputs into a system or database to trigger an SQLIA when the input is later used

Example [Halfond et al]: Second-order injection

User registers on a website using a seeded username such as “**`admin’ -- `**”.

The application escapes the single quote before storing the input in the database.

The user modifies his password.

The application (1) checks that the user knows the current password and (2) changes the password if so.

To do this, it might construct an SQL command as follows:

```
queryString=”UPDATE users SET password=’” + newPassword
+ “’ WHERE userName=’” + userName + “’ AND password=’”
+ oldPassword + “’”
```

Assume that **`newPassword`** is “newpwd” and that **`oldPassword`** is “oldpwd”.

Then the query string sent to the database is:

```
UPDATE users SET password='newpwd'
WHERE userName= 'admin'--' AND password='oldpwd'
```

Because “--” is the SQL comment operator, everything after it is ignored by the database.

Hence, the result of the query is that the database changes the password of the administrator to an attack-specified value.

2nd-order injections are especially difficult to detect and prevent because the point of injection is different from where the attack manifests itself.

A developer may escape, type-check, and filter user input and assume it is safe.

When the data is used later in a different context, it may result in an injection attack.

Attack Intent

Attacks can be classified based on the intent of the attacker:

- *Identifying injectable parameters*
- *Performing database fingerprinting*: to discover type and version of web app's DB
- *Determining database schema*: to learn table names, column names, and column data types
- *Extracting data*: most common type of SQLIA
- *Adding or modifying data*
- *Denial of service*
- *Evading detection*
- *Bypassing authentication*
- *Executing remote commands*
- *Perform privilege escalation*

Example [Halfond et al]: Servlet login code

```
String login, password, pin, query
login = getParameter("login");
password = getParameter("pass");
pin = getParameter("pin");
Connection conn.createConnection("MyDataBase");
query = "SELECT accounts FROM users WHERE login=’"
    + login + "’ AND pass=’" + password +
    "’ AND pin=" + pin;
ResultSet result = conn.executeQuery(query);
if (result!=NULL)
    displayAccounts(result);
else
    displayAuthFailed();
```

This uses input parameters **`login`**, **`pass`**, and **`pin`** to build SQL query and submit it to a database, e.g.,

```
SELECT accounts FROM users WHERE
    login=’doe’ AND pass=’secret’ AND pin=123
```

We’ll refer to the login servlet in several examples.

SQLIA Types

Tautology Attack

Intent: Bypassing authentication, identifying injectable parameters, extracting data

Description: Code is injected into one or more conditional statements so they always evaluate to *true*.

The most common usages are to bypass authentication pages and extract data.

For this, an attack exploits an injectable field used in a query's **WHERE** conditional.

Making it a tautology causes *all rows* in queried table to be returned.

The attacker must consider code that evaluates the query results.

Typically, the attack is successful when the code either

- Displays all of the returned records; or
- Performs some action if at least one record is returned

Example: Attacker submits "**`’ or 1=1 --`**" for the **`login`** input field.

The resulting query is:

```
SELECT accounts FROM users WHERE
    login=’’ or 1=1 -- AND pass=’’ AND pin=
```

In the example, the returned set is non-null.

This causes the application to decide authentication was successful and display all accounts returned by the DB.

Illegal/Logically Incorrect Query Attack

Intent: Identifying injectable parameters, database fingerprinting, extracting data

Description: Exploits overly descriptive default error page returned by application servers.

Attacker tries to inject statements that cause syntax, type conversion, or logical errors:

- *Syntax errors* can be used to identify injectable parameters.
- *Type errors* can be used to deduce the data types of columns or to extract data.
- *Logical errors* often reveal the names of the tables and columns that caused the error.

Example [Halfond et al]: Attack's goal is to cause a *type conversion error* that reveals data.

The attacker injects the following text into the input field **`pin`**:

```
“convert(int,(select top 1 name from sysobjects
where xtype=’u’)”
```

The resulting query is:

```
“SELECT accounts FROM users WHERE login=’’ AND
pass=’’AND pin=convert(int,(select top 1 name
from sysobjects where xtype=’u’))
```

The injected query attempts to extract the first user table (**`xtype=’u’`**) from the DB's metadata table **`sysobjects`** (Microsoft SQL Server).

It then tries to convert this table name into an integer.

Because this is not a legal type conversion, the database throws an error.

For Microsoft SQL Server, the error would be:

> *"Microsoft OLE DB Provider for SQL Server (0x80040E07) Error converting nvarchar value 'CreditCards' to a column of data type int."*

This reveals that the database is an SQL Server database.

It also reveals that the name of the first user-defined table is "CreditCards".

A similar strategy can be used to extract the name and type of each column in the database.

Union Query Attack

Intent: Bypassing authentication, extracting data

Description: Exploits a vulnerable parameter to change the data set returned for a given query.

This attack can trick the application into returning data from a table different from the one intended by the developer.

The attacker injects a statement of the form **`UNION SELECT <rest of injected query>`**.

The DB returns the union of the results of the original 1st query and the results of the injected 2nd query.

Example [Halfond et al]: An attacker could inject this text into the login field:

“**`' UNION SELECT cardNo from CreditCards where acctNo=10032--`**”

This produces the following query:

```
SELECT accounts FROM users WHERE login=’’ UNION
SELECT cardNo from CreditCards where
acctNo=10032 -- AND pass=’’ AND pin=
```

Assuming there is no login equal to `’’`, the original first query returns the null set.

The second query returns data from the **`CreditCards`** table – column **`cardNo`** for account **`10032`**.

The database returns the union of these two result sets.

The effect is that the value for **`cardNo`** is displayed along with the account information.

Piggy-Backed Queries Attack

Intent: Extracting data, adding or modifying data, denial of service, executing remote commands

Description: The attacker injects additional queries into the original query.

The database receives multiple queries.

The first is the original query, which is executed as normal.

The subsequent ones are injected queries.

Attackers can insert virtually any SQL command.

Vulnerability is often dependent on having a DB configuration that allows multiple statements in one string.

Example [Halfond et al]: The attacker inputs “`’; drop table users--`” into the `pass` field.

The application generates the query:

```
SELECT accounts FROM users WHERE login=’doe’ AND
pass=’’; drop table users –- ’ AND pin=123
```

After completing the 1st query, the DB would recognize the query delimiter (“;”) and execute the injected 2nd query.

The latter would drop the table `users`.

Other types of queries could insert new users into the DB or execute stored procedures.

Note that many databases don’t even require a special character to separate queries.

Stored Procedure Attack

Intent: Privilege escalation, denial of service, executing remote commands

Description: Executes stored procedure(s) present in database

Most database vendors ship DBs with a standard set of stored procedures, e.g., for OS interaction.

Example [Halfond et al]: Stored procedure for authentication

```
CREATE PROCEDURE DBO.isAuthenticated
    @userName varchar2, @pass varchar2, @pin int
AS
    EXEC("SELECT accounts FROM users
    WHERE login=’" +@userName+ "’ and pass=’"
    +@password+"’ and pin=" +@pin);
GO
```

Attacker injects “**`’;SHUTDOWN;--`**” into either the **`username`** or **`password`** fields.

This causes the stored procedure to generate this query:

```
SELECT accounts FROM users WHERE
login=’doe’ AND pass=’’; SHUTDOWN; -- AND pin=
```

The first query is executed normally, and then the second, malicious, query is executed.

The latter query shuts the database down.

Inference Attack

Intent: Identifying injectable parameters, extracting parameters, determining database schema

Description: A query is modified into an action that is executed based on true/false question about DB values

This attack is typically used when injection results in no usable feedback via database error messages.

Example [Halfond et al]: Identifying injectable parameters by *blind injection*.

Consider two possible injections into the **`login`** field:

- “**`legalUser’ and 1=0--`**”
- “**`legalUser’ and 1=1--`**”

These result in the following queries:

```
SELECT accounts FROM users WHERE login=’legalUser’ and
1=0 -- ’ AND pass=’’ AND pin=0

SELECT accounts FROM users WHERE login=’legalUser’ and
1=1 -- ’ AND pass=’’ AND pin=0
```

Consider two scenarios.

In the first scenario, we have a secure application, and the input for **`login`** is validated correctly.

In this case, both injections would return login error messages.

The attacker would infer that the `login` parameter is not vulnerable.

In the second scenario, we have an insecure application and the `login` parameter is vulnerable to injection.

The attacker submits the first injection and, because it always evaluates to false, the application returns a login error message.

The attacker does not know whether

- The application validated the input correctly and blocked the attack; or
- The attack itself caused the login error

The attacker then submits the second query, which always evaluates to true.

If there is no login error message, he knows the attack went through and that the **`login`** parameter is vulnerable.

Inference-based attacks can also be used to perform data extraction.

Example [Halfond et al]: Use of *timing-based inference attack* to extract a table name from a database.

The following is injected into the **`login`** parameter:

```
‘‘legalUser’ and ASCII(SUBSTRING((select top 1 name
from sysobjects),1,1)) > X WAITFOR 5 --’’
```

This produces the following query:

```
SELECT accounts FROM users WHERE login=’legalUser’ and
ASCII(SUBSTRING((select top 1 name from sysobjects),
1,1)) > X WAITFOR 5 -- ’ AND pass=’’ AND pin=0
```

The **`SUBSTRING`** function is used to extract the first character of the first table’s name.

If the ASCII value of the character is greater than the value of ***X***, the attacker observes an additional 5 second delay in the database response.

The attacker can then use a binary search, by varying the value of ***X***, to identify the value of the first character.

Alternate Encodings

Intent: Evading detection, in conjunction with other attacks.

Description: The injected text is modified to avoid detection by

- Defensive coding practices
    - Scanning for "bad characters" such as single quotes and "`--`".
- Automated prevention techniques

Attackers employ alternate methods of encoding attack strings, e.g., hexadecimal, ASCII, and Unicode.

Common scanning and detection techniques don't try to evaluate all specially encoded strings.

Different application layers have difference ways of handling alternate encodings.

Example [Halfond et al]: The following text is injected into the **`login`** field:

```
"legalUser'; exec(0x73687574646f776e) - - "
```

The query generated by the application is:

```
SELECT accounts FROM users WHERE login=’legalUser’;
exec(char(0x73687574646f776e)) -- AND pass=’’ AND pin=
```

The **`char()`** function takes an integer or hex encoding of one or more characters and returns a string consisting of those characters.

Its argument here is the ASCII hex encoding of “**`SHUTDOWN`**”.

Hence the query will result in execution of the **`SHUTDOWN`** command.

Prevention of SQLIAs

Researchers have proposed a wide range of techniques to address SQL injection.

Defensive Coding Practices

Recall that the root cause of SQLI vulnerabilities is insufficient input validation.

One solution is application of defensive coding practices:

- *Input type checking*: e.g., rejecting numeric input containing non-digit characters
- *Encoding of inputs*: Encoding meta-characters so database interprets them as normal characters
- *Positive pattern matching*: Input validation routines that identify all forms of legal input
- *Identification and checking of all input sources*

Although defensive coding remains the best way to prevent SQL injection, its application is problematic.

Defensive coding is prone to human error.

Pseudo-remedies like checking user input for SQL keywords and operators generate many false positives.

They can also be subverted.

Detection and Prevention Techniques

Black-Box Testing

WAVES [Huang et al] is a block-box technique that uses a Web crawler to identify injection points in an application.

It uses a specified list of patterns and attack techniques to build attacks that target such points.

It monitors the application’s response and uses machine learning to improve its attacks.

It cannot guarantee completeness.

Static Code Checkers

JDBC-Checker [Gould et al] checks the type correctness of dynamically generated SQL queries.

It cannot catch SQLIAs involving syntactically and type correct queries.

Wasserman and Su [2004] used static analysis and automated reasoning to verify that generated queries don’t contain a tautology.

Combined Static and Dynamic Analysis

AMNESIA [Halfond & Orso] combines static analysis and runtime monitoring.

It uses static analysis to build models of queries an application can legally generate at each point of database access.

At runtime, it intercepts all queries and checks them against the models.

Its success depends on the accuracy of its static analysis.

SQLGuard [Buehrer et al] and *SQL-Check* [Wasserman & Su] also check queries against a model at runtime.

The model is expressed as a grammar for legal queries.

Both approaches use a secret key to delimit user input during parsing.

Taint-Based Approaches

WebSSARI [Huang et al] detects input-validation-related errors using *information flow analysis*.

Static analysis is used to detect taint flows against preconditions for sensitive functions.

This detects points in which preconditions haven't been met and can suggest filters/sanitation functions.

Dynamic taint analysis approaches have also been proposed.

Nguyen-Toung et al [2005] and Pietraszek and Berghe [2005] use *context sensitive analysis*.

This detects and rejects queries if untrusted input has been used to create certain types of SQL tokens.

New Query Development Paradigms

SQL DOM [McClure & Krueger] and *Safe Query Objects* [Cook & Rai] use encapsulation of database queries.

Queries are created using a type-checked API instead of string concatenation.

Intrusion Detection Systems

Valeur et al [2005] use an IDS to detect SQLIAs.

Their IDS employs a machine learning algorithm trained using typical application queries.

It learns a model of typical queries and then monitors the application for queries that don’t match the model.

Proxy Filters

Secure Gateway [Scott and Sharp] is a proxy filtering system.

It enforces input validation rules on the data flowing to a Web application.

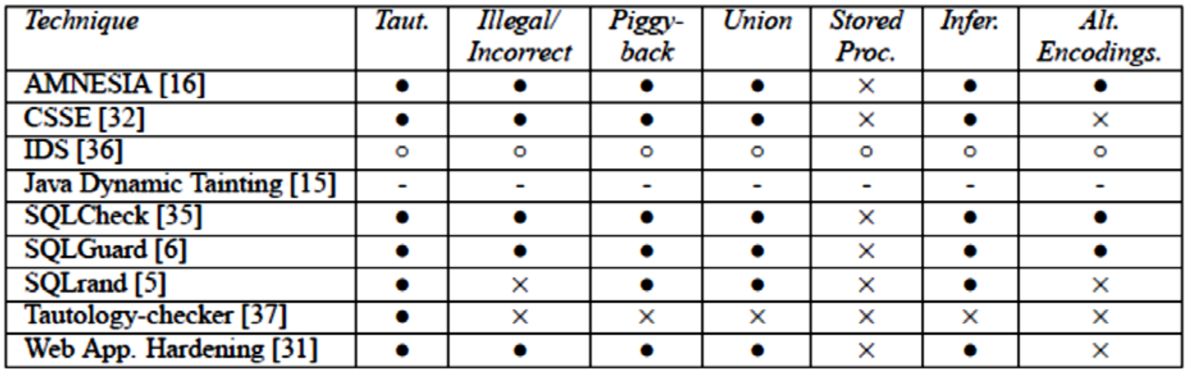
Developers use a security policy description language to provide constraints and specify transformations to be applied to application parameters.

Instruction Set Randomization

SQLrand [Boyd & Keromytis] enables developers to create queries using randomized SQL keywords.

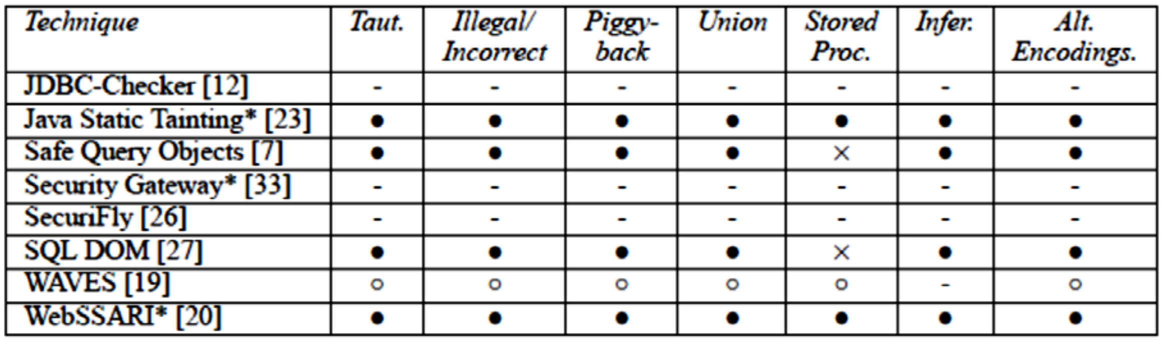
A proxy filter intercepts queries and de-randomizes them, using a *secret key*.

Comparison with Respect to Attack Types [Halfond et al]



Technique	*Taut.*	*Illegal/ Incorrect*	*Piggy-back*	*Union*	*Stored Proc.*	*Infer.*	*Alt. Encodings.*
AMNESIA [16]	●	●	●	●	×	●	●
CSSE [32]	●	●	●	●	×	●	×
IDS [36]	○	○	○	○	○	○	○
Java Dynamic Tainting [15]	-	-	-	-	-	-	-
SQLCheck [35]	●	●	●	●	×	●	●
SQLGuard [6]	●	●	●	●	×	●	●
SQLrand [5]	●	×	●	●	×	●	×
Tautology-checker [37]	●	×	×	×	×	×	×
Web App. Hardening [31]	●	●	●	●	×	●	×

Table 1: Comparison of detection-focused techniques with respect to attack types.



Technique	*Taut.*	*Illegal/ Incorrect*	*Piggy-back*	*Union*	*Stored Proc.*	*Infer.*	*Alt. Encodings.*
JDBC-Checker [12]	-	-	-	-	-	-	-
Java Static Tainting* [23]	●	●	●	●	●	●	●
Safe Query Objects [7]	●	●	●	●	×	●	●
Security Gateway* [33]	-	-	-	-	-	-	-
SecuriFly [26]	-	-	-	-	-	-	-
SQL DOM [27]	●	●	●	●	×	●	●
WAVES [19]	○	○	○	○	○	-	○
WebSSARI* [20]	●	●	●	●	●	●	●

Table 2: Comparison of prevention-focused techniques with respect to attack types.

●	Stops all attacks of type
×	Not able to stop attacks of type
○, -	Partially successful