2. Top 3: Injection

An application is vulnerable to attack when:

- User-supplied data **is not validated, filtered, or sanitized** by the application.
- Dynamic queries (SQL: "SELECT * FROM users WHERE username = '" + userInput + "'") or non-parameterized calls (string sqlQuery = "SELECT * FROM users WHERE username = '" + userInput + "'";) without context-aware escaping(without field validation) are used directly in the interpreter.
- Hostile data is used within object-relational mapping (ORM)[programming technique and methodology used in software development to bridge the gap between object-oriented programming (OOP) languages and relational databases. Example: Select * from users where userid=1; is equal to python Django Code: user = User.objects.get(userid=1)] search parameters to extract additional, sensitive records.
- Hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures.

Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Source code review is the best method of detecting if applications are vulnerable to injections. Automated testing of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs is strongly encouraged. Organizations can include static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline to identify introduced injection flaws before production deployment.

How to Prevent

Preventing injection requires keeping data separate from commands and queries:

In the context of SQL and databases, "escaping" refers to the practice of adding special characters or sequences to data before including it in an SQL query or statement. The purpose of escaping is to ensure that the data being inserted into the query is treated as literal values and not as part of the SQL code. This is done to prevent SQL injection and ensure that user input or dynamic data does not compromise the security and integrity of the database.

- The preferred option is to use a safe API, which avoids using the interpreter entirely, provides a parameterized interface, or migrates to Object Relational Mapping Tools (ORMs).
  Note: Even when parameterized, stored procedures can still introduce SQL injection if PL/SQL or T-SQL concatenates queries and data or executes hostile data with EXECUTE IMMEDIATE or exec().
- Use positive server-side input validation. This is not a complete defense as many applications require special characters, such as text areas or APIs for mobile applications.
- For any residual dynamic queries, escape special characters using the specific escape syntax for that interpreter.
  Note: SQL structures such as table names, column names, and so on cannot be escaped, and thus user-supplied structure names are dangerous. This is a common issue in report-writing software.
- Use LIMIT and other SQL controls within queries to prevent mass disclosure of records in case of SQL injection.

Example Attack Scenarios

Scenario #1: An application uses untrusted data in the construction of the following vulnerable SQL call:

```
String query = "SELECT \* FROM accounts WHERE custID='" + request.getParameter("id") + "'";
```

Scenario #2: Similarly, an application's blind trust in frameworks may result in queries that are still vulnerable, (e.g., Hibernate Query Language (HQL)):

```
 Query HQLQuery = session.createQuery("FROM accounts WHERE custID='" + request.getParameter("id") + "'");
```

In both cases, the attacker modifies the 'id' parameter value in their browser to send: ' UNION SLEEP(10);--. For example:

```
 http://example.com/app/accountView?id=' UNION SELECT SLEEP(10);--
```

This changes the meaning of both queries to return all the records from the accounts table. More dangerous attacks could modify or delete data or even invoke stored procedures.

- Top 3: Hands-on