A1: Injection

- Allowing Untrusted data to be sent as a part of a command or query.
- Once attacker can make command, they can control your website, apps and data.

Almost any source of data can be an injection vector, environment variables, parameters, external and internal web services, and all types of users. Injection flaws occur when an attacker can send hostile data to an interpreter.

Injection flaws are very prevalent, particularly in legacy code. Injection vulnerabilities are often found in SQL, LDAP, XPath, or NoSQL queries, OS commands, XML parsers, SMTP headers, expression languages, and ORM queries.

Injection flaws are easy to discover when examining code. Scanners and fuzzers can help attackers find injection flaws.

Injection can result in data loss, corruption, or disclosure to unauthorized parties, loss of accountability, or denial of access. Injection can sometimes lead to complete host takeover. (Impact)

The business impact depends on the needs of the application and data.

Is the Application Vulnerable?

An application is vulnerable to attack when:
- User-supplied data is not validated, filtered, or sanitized by the application.
- Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter.
- Hostile data is used within object-relational mapping ?? (ORM) search parameters to extract additional, sensitive records.
- Hostile data is directly used or concatenated, such that the SQL or command contains both structure and hostile 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, closely followed by thorough automated testing of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. Organizations can include static source (SAST) and dynamic application test (DAST) tools into the CI/CD pipeline to identify newly introduced injection flaws prior to production deployment.

How to Prevent

Preventing injection requires keeping data separate from commands and queries.

- The preferred option is to use a safe API, which avoids the use of the interpreter entirely or provides a parameterized interface, or migrate to use 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 or "whitelist" 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 structure 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.