

# Injection-Based Attacks

Injection is a class of vulnerability where an application is tricked into processing user-supplied data as executable code or commands. This typically arises when an application incorporates unvalidated user input directly into a code interpreter. The exploited technology often relies on interpreted languages, allowing an attacker to modify the logical flow of the application. The primary goals of injection attacks include bypassing authentication, extracting sensitive information, or achieving remote code execution. The three major types include SQL Injection, Command Injection, and Server-Side Template Injection.

## 1.1 SQL Injection

Databases are structured collections of data, modeling aspects of the real world. While many types exist (e.g., relational, NoSQL, graph), relational databases are the most common target for SQL injections. Data in relational databases is organized into tables, with rows representing individual records and columns defining the fields or attributes. SQL (Structured Query Language) is the standard language for managing and manipulating data in these databases.

**Authentication Bypass.** User authentication often relies on checking credentials against a database. A vulnerable application might construct an authentication query by directly concatenating user input into the SQL string. Consider a login attempt where the username is `Markus` and the password is `secret`. A typical, vulnerable query structure might look like this:

```
SQL

SELECT * FROM users WHERE username='Markus' AND password='secret'
```

The danger lies in the fact that the application is executing a single, constructed string that contains both the fixed query logic and the user-controlled data. If an attacker inserts characters that have special meaning in SQL (like a single quote `'`), they can break out of the intended data field and inject their own SQL syntax, which the database interpreter will execute. A common goal in SQL is authentication bypass, often achieved using the `OR 1 = 1` trick:

- (1) *Attacker Input:* The attacker enters a specially crafted string into the password field, such as: `'OR 1 = 1 --'`.
- (2) *Resulting Query:* The application constructs the final SQL statement, which becomes:

```
SQL

SELECT * FROM users WHERE username='Markus' AND password=' ' OR 1 = 1 --'
```

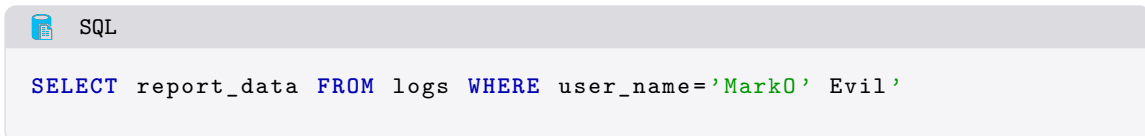
- (3) *Database Interpretation:* The first single quote closes the opening quote for the password field. The `OR 1 = 1` is injected into the query logic. Since `1 = 1` is always true, the entire `WHERE` clause condition now evaluates to true. The `--` is a SQL comment operator. It tells the database to ignore all subsequent text in the query string, which effectively

nullifies the final closing single quote and any other trailing query components, preventing a syntax error.

Because the `WHERE` condition is universally true, the query returns the record for the specified user without needing the correct password, thus successfully logging the attacker in.

**Second Order SQL Injection.** Protecting against SQL Injection often involves input validation and sanitization, such as escaping single quotes (e.g., turning `'` into `"`). However, if this validation is applied only at the initial entry point, a vulnerability known as Second-Order SQL Injection can occur. In this attack, the malicious payload is stored by the application in the database during an initial, non-exploitable transaction. When the application executes a second query at a later time, retrieving and using this stored, unsanitized value, the attack is triggered.

- (1) *First Query:* A user changes their display name to a payload like `Mark0' Evil`. The application may correctly escape the quote before storing it, resulting in `Mark0" Evil` in the database.
- (2) *Second Query:* A separate administrative function later executes a query that retrieves and uses this stored data without applying sanitization. If the stored value is `Mark0' Evil`, a query like the following will break the SQL syntax and potentially allow the attacker to inject their code:



```
SELECT report_data FROM logs WHERE user_name='Mark0' Evil'
```

**Blind SQL Injection.** Blind SQL Injection occurs when the attacker does not receive direct feedback from the database execution. Instead of seeing results, the attacker must infer the database structure and data by observing the application's behavior or response time to various injected conditions.

**Defending Against SQL Injection.** The most effective and fundamental defense against SQL Injection is the use of parametrized queries, also known as Prepared Statements. This defensive approach enforces a strict separation between the SQL query structure and the user-supplied data.

- (1) *Preparation:* The application first sends a template of the SQL query to the database, where the position for user input are replaced with placeholders.
- (2) *Execution:* The application then sends the user input to the database separately as a parameter.
- (3) *Database Handling:* The database API is designed to treat the data in the parameter exclusively as a literal string value, never as executable SQL code. This means any embedded quotes or special SQL syntax in the user input are automatically neutralized, preventing the malicious code from breaking the original query structure.

This technique prevents the attacker from modifying the query's intent, regardless of what they submit.

## 1.2 Command Injection

Command Injection is a vulnerability that allows an attacker to execute arbitrary operating system commands on the host server. This flaw arises when a web application incorporates unvalidated user inputs directly into functions that are designed to execute system-level commands or shell commands. Many web applications, particularly those written in interpreted languages like PHP, use system-related APIs or functions to interact with the underlying OS. If an attacker can inject special shell characters into the user input, they can effectively terminate the intended command and append an arbitrary command for execution.

**Example of Command Injection.** The `eval()` function in PHP takes a string and executes it as PHP code. If a web page uses an external parameter to construct the code passed to `eval()`, it introduces a high-risk vulnerability.

```
PHP
$storedsearch = $_GET['storedsearch'];
eval("$storedsearch;");
```

The attacker exploits the semicolon to separate the intended, harmless statement from their malicious command. The input is URL-encoded.

- *Input:* `/search.php?storedsearch=$search=wahh;%20system('cat%20/etc/passwd')`
- *Server Executes:* `eval("$search=wahh; system('cat /etc/passwd');");`

The database will execute the arbitrary OS command `cat /etc/passwd`, causing the contents of the critical system file to be displayed to the attacker. Attackers initially test for this flaw by submitting common shell characters like `;` or `&` to observe if the application responds with an error or executes the injected command.

**Path Traversal.** Path Traversal is a distinct but related vulnerability often found in file-handling functionalities. This attack exploits flaws in how an application constructs file paths, allowing the attacker to access files and directories outside of the application's intended root directory. This vulnerability typically occurs when the server uses user-supplied input to specify the name or path of a file to be opened or read.

- *Intended Access URL:* `http://example.com/GetFile.ashx?filename=keira.jpg`

The server's code expects to open `keira.jpg` within the secure file store directly.

- *Path Traversal Attack:* The attacker injects the directory traversal sequence `../`, which means "go up one directory level". By chaining multiple of these sequences, the attacker attempts to navigate upward in the file system hierarchy until they reach the root directory. From there, they specify the path to a sensitive system file, such as `/etc/passwd`.

If the server fails to sanitize the `filename` parameter by stripping or resolving the `../` sequences, it will execute the instruction to read the contents of the `/etc/passwd` file and display it.

**PHP Code Injection.** Code Injection is a variation of injection attacks where the attacker injects and executes arbitrary programming language code rather than OS commands. This is

highly common in PHP due to functions like `eval()`. The vulnerability occurs when a code evaluation function's content can be partially or fully controlled by the user. This often leads to maximum impact, as the injected code executes with the same privileges as the application itself.

```
PHP
eval("\$$user = '$regdate'");
```

The code above is intended to dynamically create a variable named after the content of `$user` (e.g., if `$user` is `username`), the resulting code is `eval("$username = 'date'")`. The attacker controls this variable and injects a payload that includes PHP syntax separators (`;`). The server concatenates the string and executes:

```
PHP
eval("\$x = 'y';phpinfo();// = '$regdate'");
```

The semicolon successfully terminates the intended statement and allows the insertion of the malicious code: `phpinfo()`. The double forward slash comments out the remaining, syntactically-incorrect portion of the original string, preventing a parse error. The function `phpinfo()` then executes, leaking extensive configuration and sensitive details about the PHP environment.

### 1.3 Server-Side Template Injection

Server-Side Template Injection (SSTI) is a critical vulnerability that occurs when user-supplied input is insecurely processed and rendered directly within a server-side template engine. Template engines are a vital part of modern web applications, separating application logic from presentation. They facilitate the creation of dynamic HTML pages by combining static content with dynamic data. The core components involved are:

- *Data Source*: The dynamic data (e.g., username, product list) provided by the application logic.
- *Web Template*: The file or string containing the static HTML structure and placeholders for dynamic data.
- *Template Engine*: The server-side software (e.g., Jinja2, Twig, Velocity) that combines the data source with the web template to generate the final output sent to the user's browser.

Template engines are a fundamental feature of many popular web frameworks, such as Django and Flask.

**Example of Vulnerable Code.** SSTI arises when the web application constructs a template by concatenating unsanitized user input directly into the template definition string, and then instructs the template engine to parse that newly created string. The security flaw is evident in how the `user_input` is merged directly into the template string before rendering.



Python

```
user_input = request.form['username']
# Insecure: User input is inserted directly into the template structure
template = "<html><h1>Welcome, %s !</h1></html>" % user_input
return render_template_string(template)
```

If an attacker enters a template expression, such as `{{7*7}}`, as the username, the template engine will interpret it, perform the calculation, and render "Welcome, 49!" instead of the literal input string.

SSTI can be exploited because template languages are designed to handle complex logic, including functions, object manipulation, and sometimes, direct access to the underlying programming language's objects and methods. The fundamental steps for exploitation are:

- (1) *Detection*: Identifying the vulnerability by testing basic template expressions (e.g., `${7*7}`, `{{7*7}}`, `<%= 7*7 %>`).
- (2) *Engine Identification*: Determining the specific template engine and its version, as this dictates the exact syntax and objects available for exploitation.
- (3) *Exploit Crafting*: Constructing a payload to execute commands. This is often achieved by abusing features that allow navigation through the host language's object inheritance chain (e.g., in Python's Jinja2, using special objects like `__globals__`) to access system libraries and execute operating system commands.

The final impact of a successful SSTI exploit is typically Remote Code Execution (RCE), allowing an attacker to execute arbitrary commands on the web server itself.