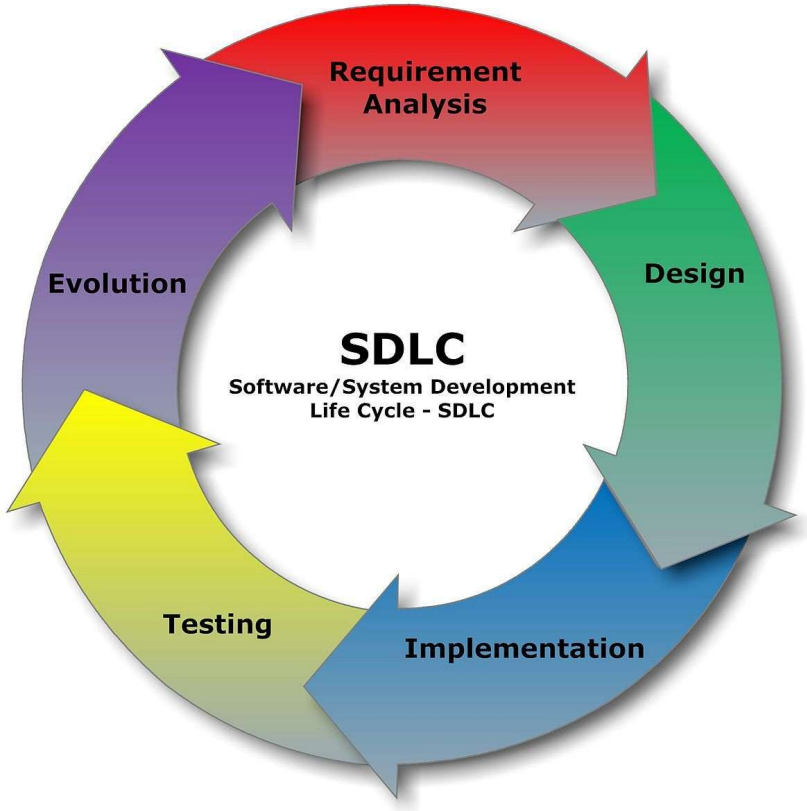
Co-funded by
the European Union

MERIT

Database Security

Injection Attacks and Input Validation

Software Development

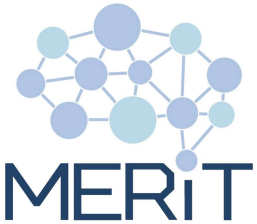


The Software Development Life Cycle (SDLC) is a structured process used for planning, creating, testing, and deploying software applications. It provides a framework that ensures high-quality software is delivered efficiently and systematically. The main phases typically include:

- ✔ Requirements gathering – Understanding what the users need.
- ✔ Design – Planning the system architecture and user interface.
- ✔ Implementation – Writing the actual code.
- ✔ Testing – Checking for bugs and verifying functionality.
- ✔ Deployment – Releasing the software to users.
- ✔ Maintenance – Updating and fixing the software after release.

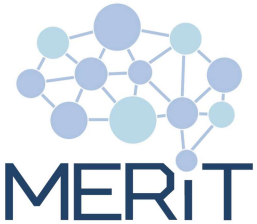
The SDLC helps teams manage complexity, reduce risks, and deliver products that meet user expectations.

Secure Software Development



- Consider cybersecurity throughout the software development life cycle
  - ✔ Requirements
  - ✔ Design
  - ✔ Implementation
  - ✔ Testing
  - ✔ Deployment

Relationship between SSDLC and Databases



- SSDLC ensures databases are secured throughout software development.
- Defines security requirements early (encryption, access control).
- Implements secure architecture and schemas to prevent database threats.
- Uses secure coding (prepared statements, input validation) to avoid injections.
- Conducts security tests (audits, pentesting) before deployment.
- Monitors databases continuously, applying patches regularly.

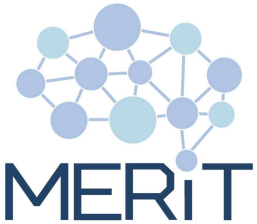
Objective:
Protect databases effectively and minimize security risks from development to operation.

Requirements

- Identify sensitive data and resources
- Define security requirements for them
  - ✔ Confidentiality
  - ✔ Integrity
  - ✔ Availability
- Consider threats and abuse cases that violate these requirements

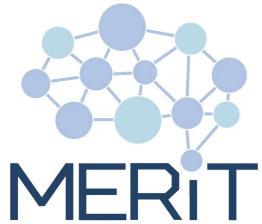
Requirements

Design



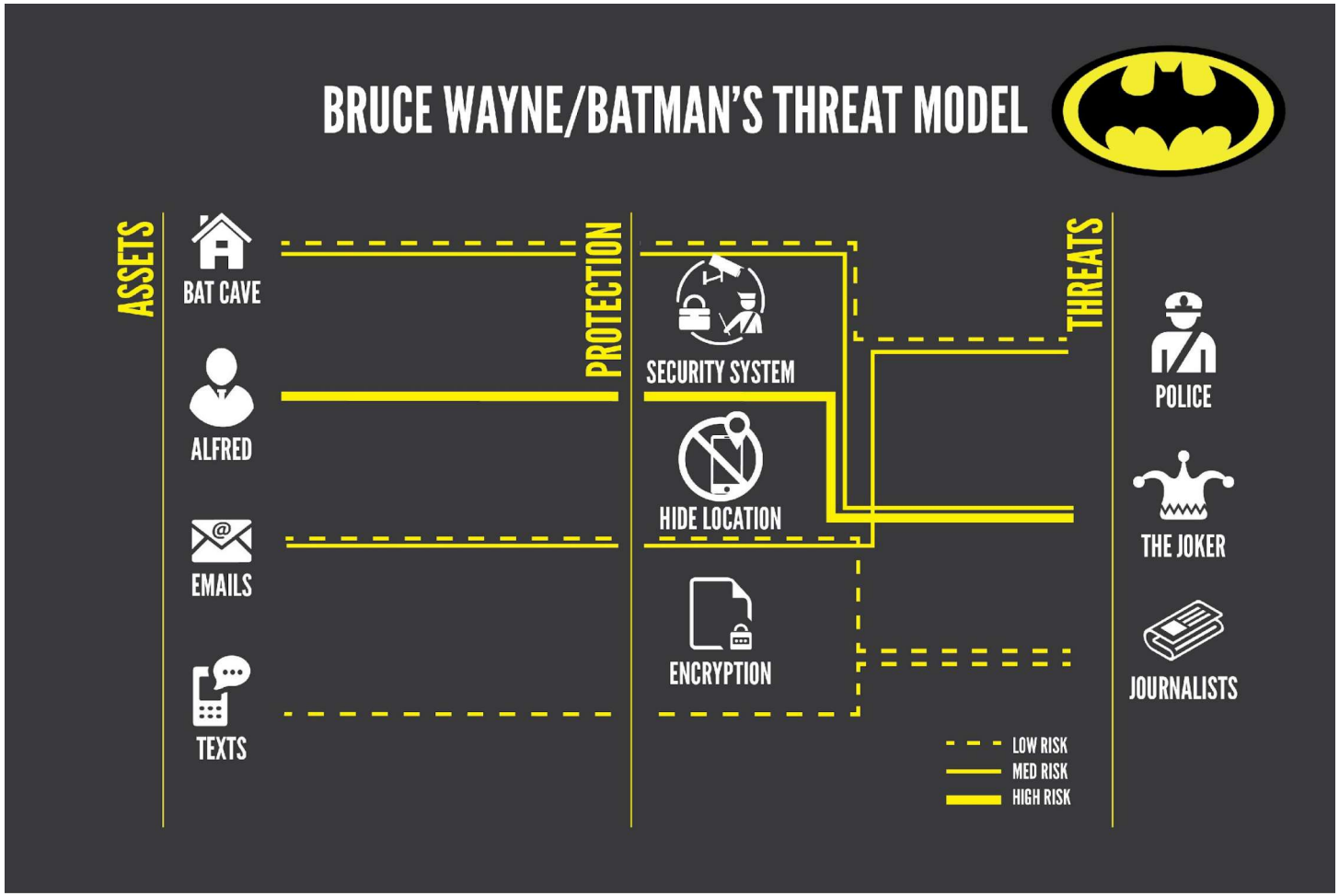
- Apply principles for secure software design
  - ✔ Prevent, mitigate and detect possible attacks
- Security principles
  - ✔ Favor Simplicity
  - ✔ Trust with Reluctance
  - ✔ Defend in Depth

Implementation



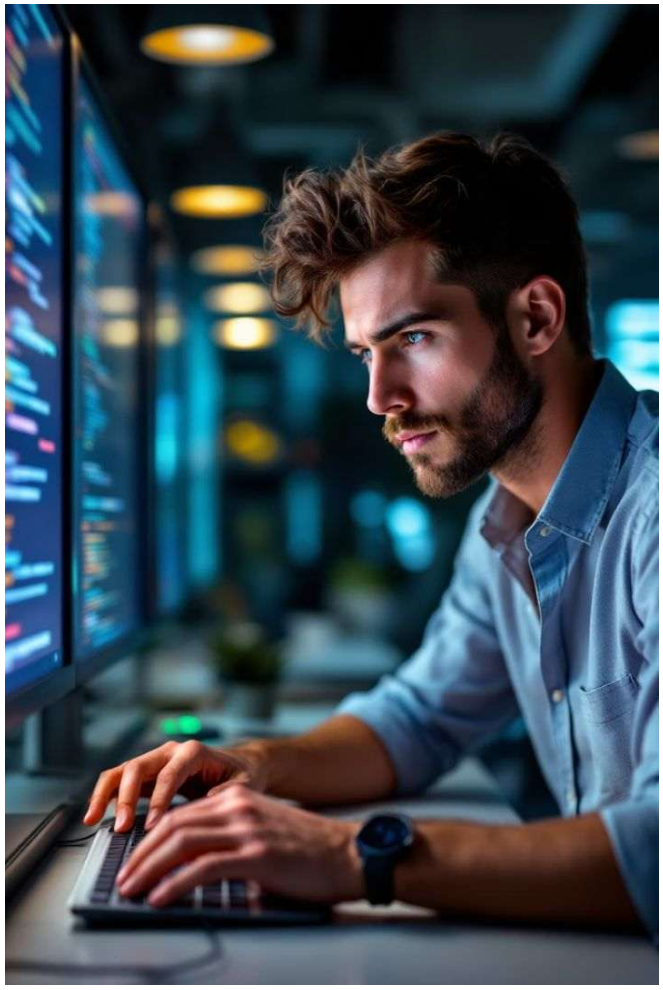
- Apply coding rules that implement secure design
- Use automated code review techniques to find potential vulnerabilities components
  - ✔ Static Analysis
  - ✔ Symbolic execution

Implementation



Injection Attacks and Input Validation in SDLC

Injection attacks pose a critical threat to applications. Attackers exploit unsanitized inputs to execute arbitrary commands. Mitigation techniques include input validation, escaping, and parameterized queries. Secure coding practices are essential in the software development lifecycle.



SQL Injection (SQLi) Explained

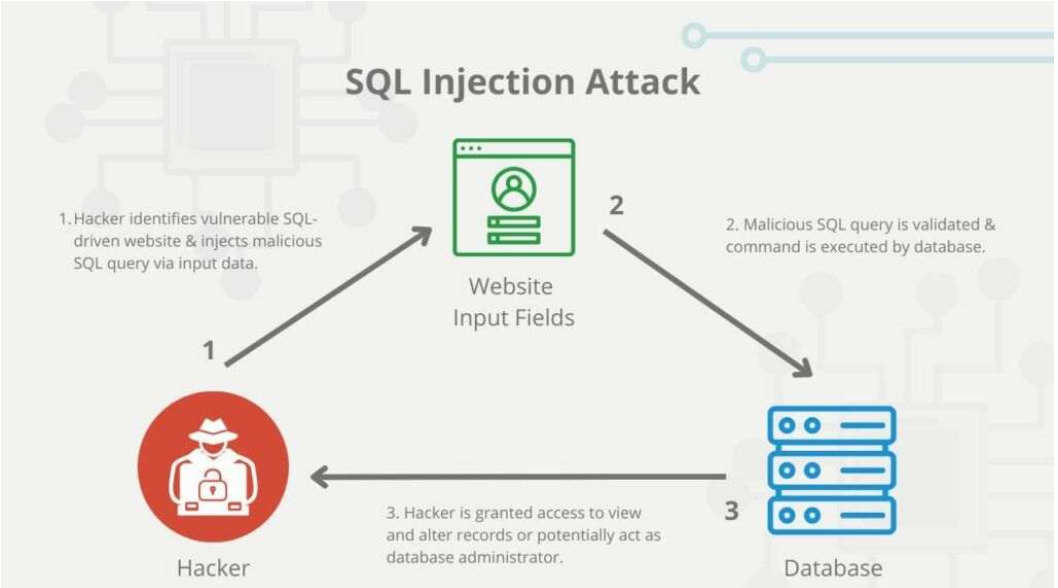
Vulnerability

Occurs when user input is concatenated into

SQL statements.
Example: **SELECT * FROM users WHERE**

username = 'admin' AND password = 'pass123'

OR '1'='1';



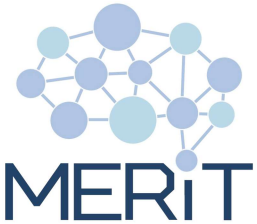
Impact

Bypasses authentication mechanisms.

Allows extraction of sensitive data from

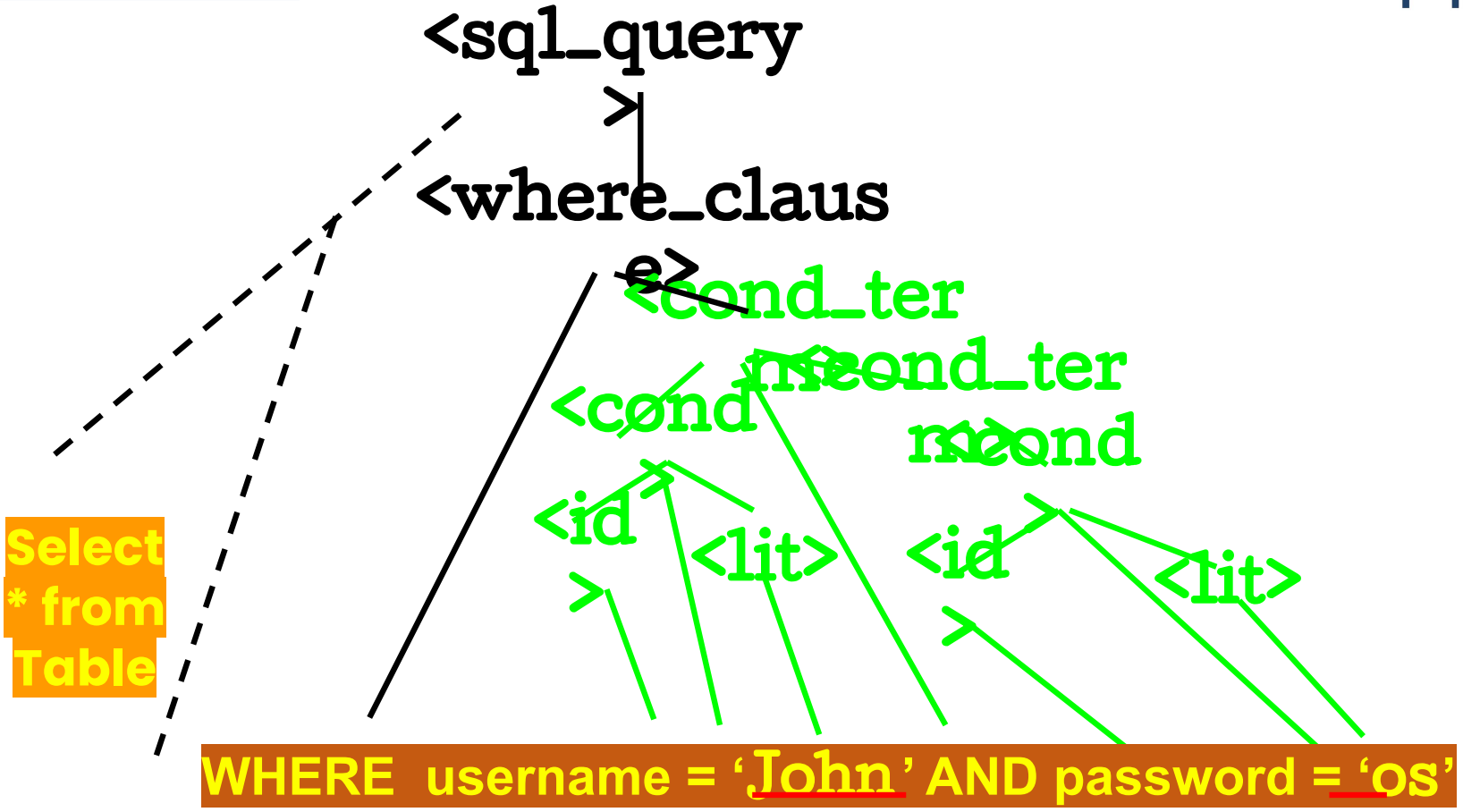
the database.

SQL Injection

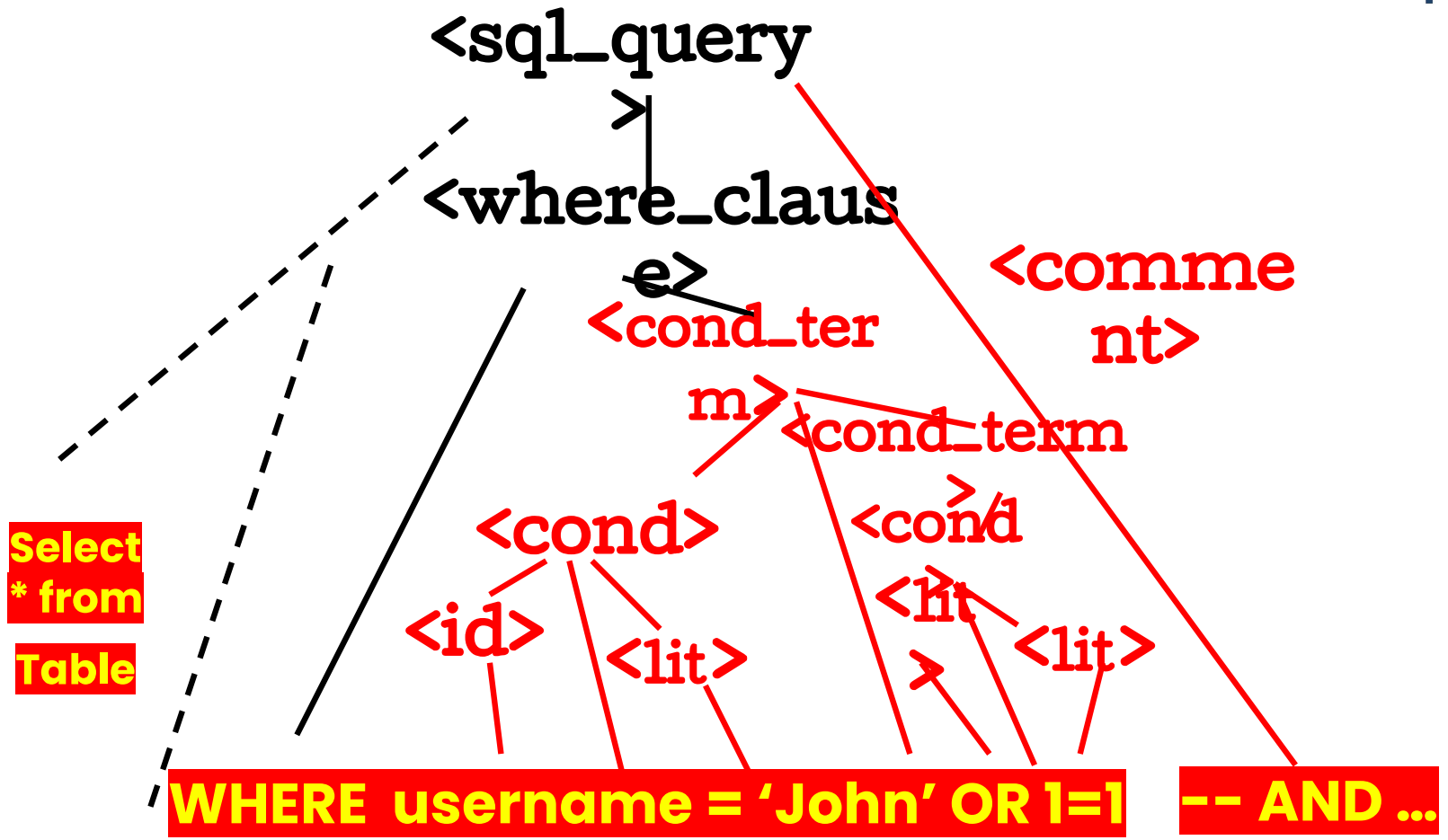


- Most systems separate code from data
- SQL queries can be constructed by arbitrary sequences of programming constructs that involve string operations
  - ✔ Concatenation, substring ….
- Such construct also involve (untrusted) user inputs
  - ✔ Inputs should be mere “data”, but in case of SQL results in “code”
- Result: Queries intended by the programmer can be “changed” by untrusted user input

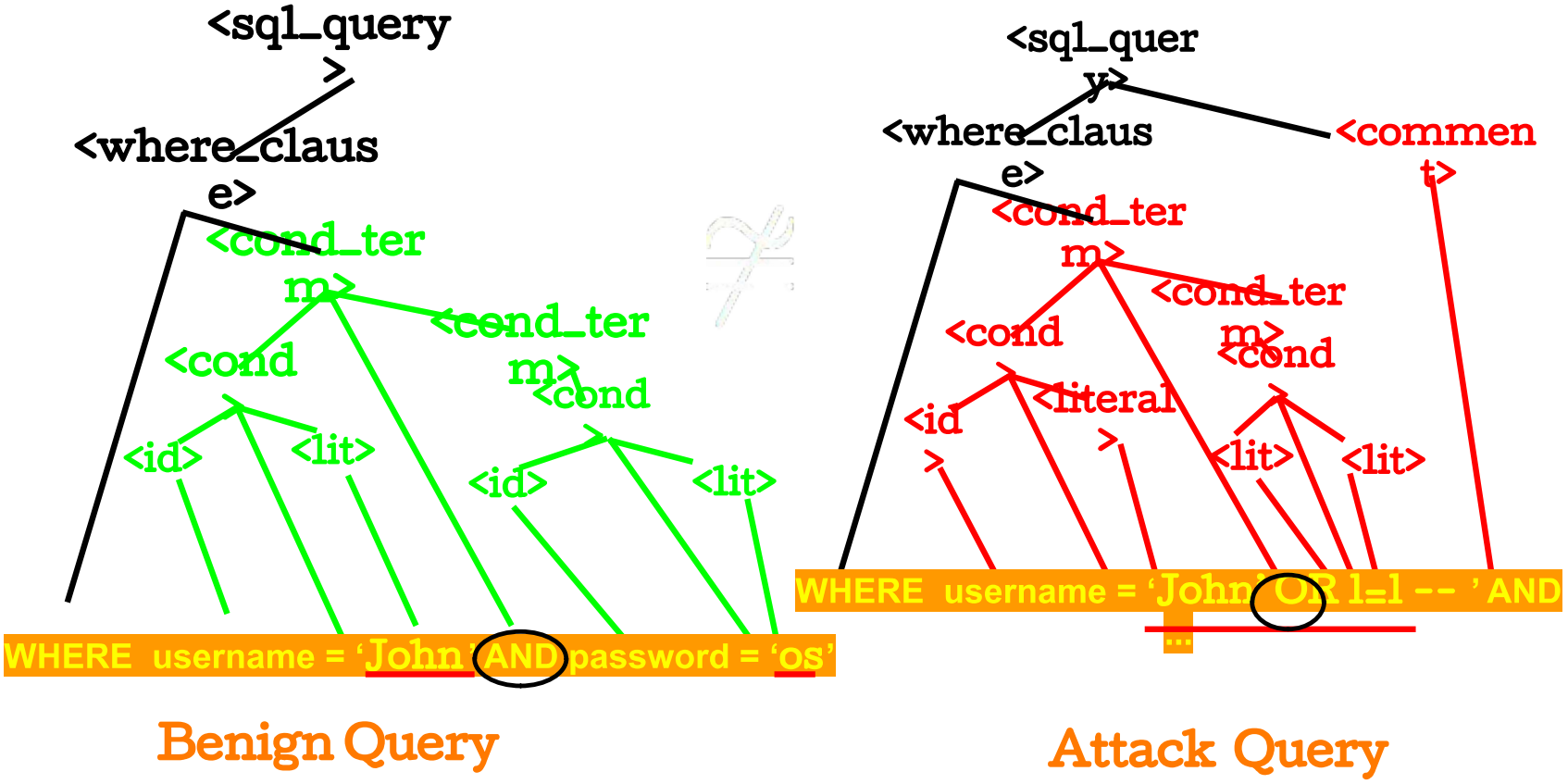
Parse Structure for a Benign Query



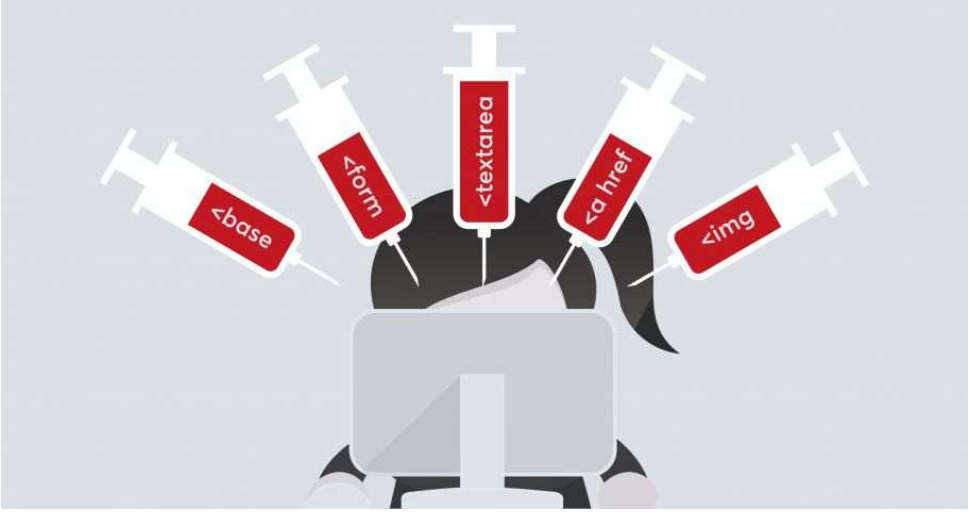
Parse Structure for a Attack Query



Attacks Change Query Structure



XML and Code Injection Risks



XML Injection

Modifies XML structure, such as SOAP requests.

XXE

XML External Entity attacks can access internal files.

Code Injection

Occurs when input is directly evaluated. Examples: **eval(user_input)**, **system(user_input)**.

Prepared Statements for Prevention

Protection

Prevent SQL injection effectively.

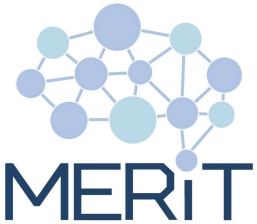
Example

```
cur.execute('SELECT * FROM users WHERE id = %s', (user_id,))
```

Separation

Separates code from data, eliminating injection risks.

ORMs for Secure Data Handling



1

Secure

Frameworks handle input securely.

2

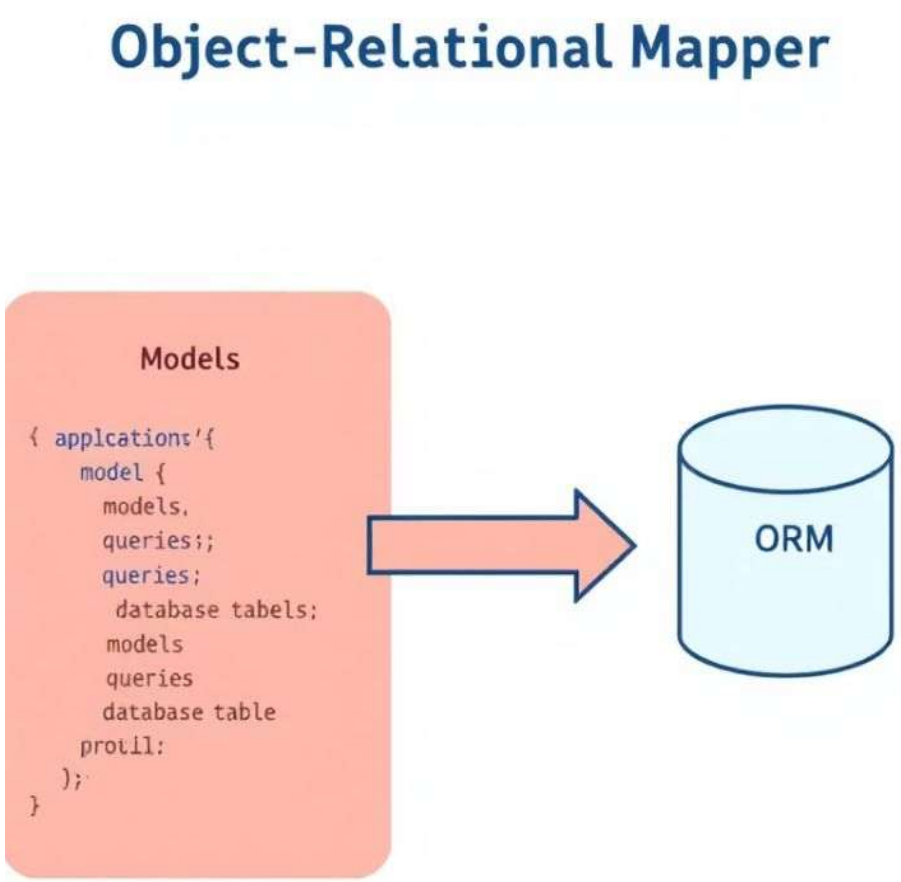
Maintainable

Improves code maintainability.

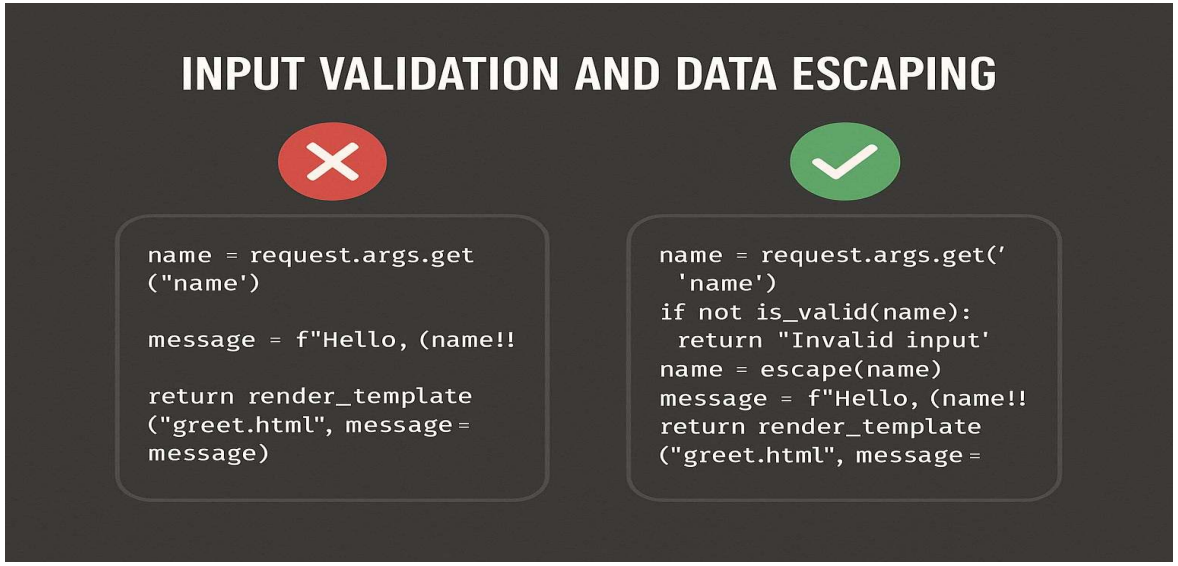
3

Example

(Django): User.objects.filter(username=username)



Input Validation and Data Escaping



Allow-lists

Enforce strict validation rules for types, formats, and lengths.

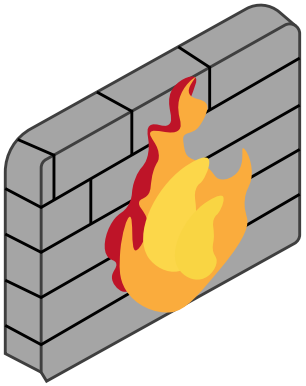
Sanitize

Reject or sanitize any unexpected or malicious input.

Escape

Escape special characters before interpreting or rendering data.

How a WAF Supports Sanitization, Validation, and Data Escaping



🔐 Combined Defense Strategy

A WAF is most effective when combined with secure development practices:

Layer	Responsibility
Application code	Proper validation, sanitization, and escaping
Database layer	Use of parameterized queries / ORM
WAF	Blocking known attack patterns & bad inputs

🚫 **Important Note**

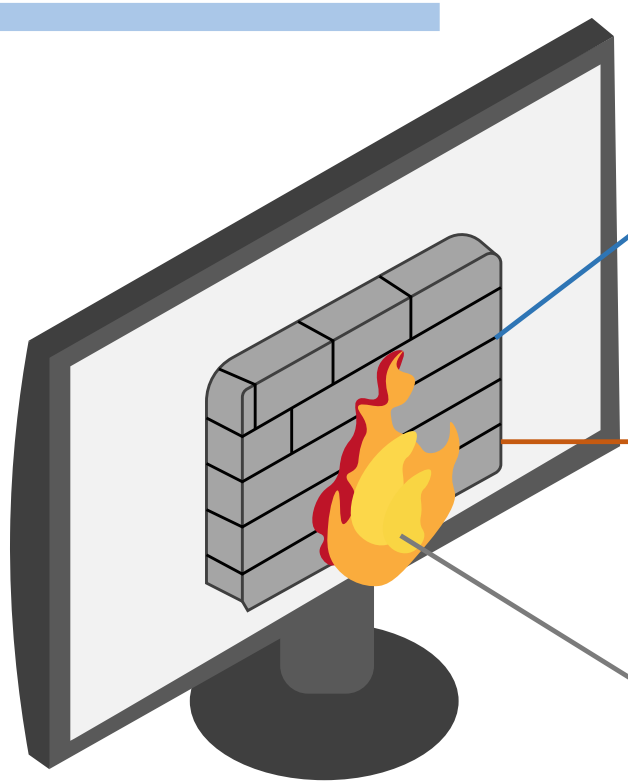
A WAF is not a substitute for secure coding.

Developers must still:

- ✔ Use prepared statements (not string concatenation).
- ✔ Validate all user input server-side.
- ✔ Sanitize and escape data before using it in queries or responses.

WEB APPLICATION FIREWALL

Types of Web Application Firewalls



01 | Hardware-Based WAF
Installed locally within LAN or local area network and deployed on a physical piece of hardware.

02 | Software Based WAF
Functions same as the hardware-based WAF but allows for increased flexibility with a lower cost.

03 | Cloud-Based WAF
Located entirely in the cloud and the service provider manages everything.

WEB APPLICATION FIREWALL

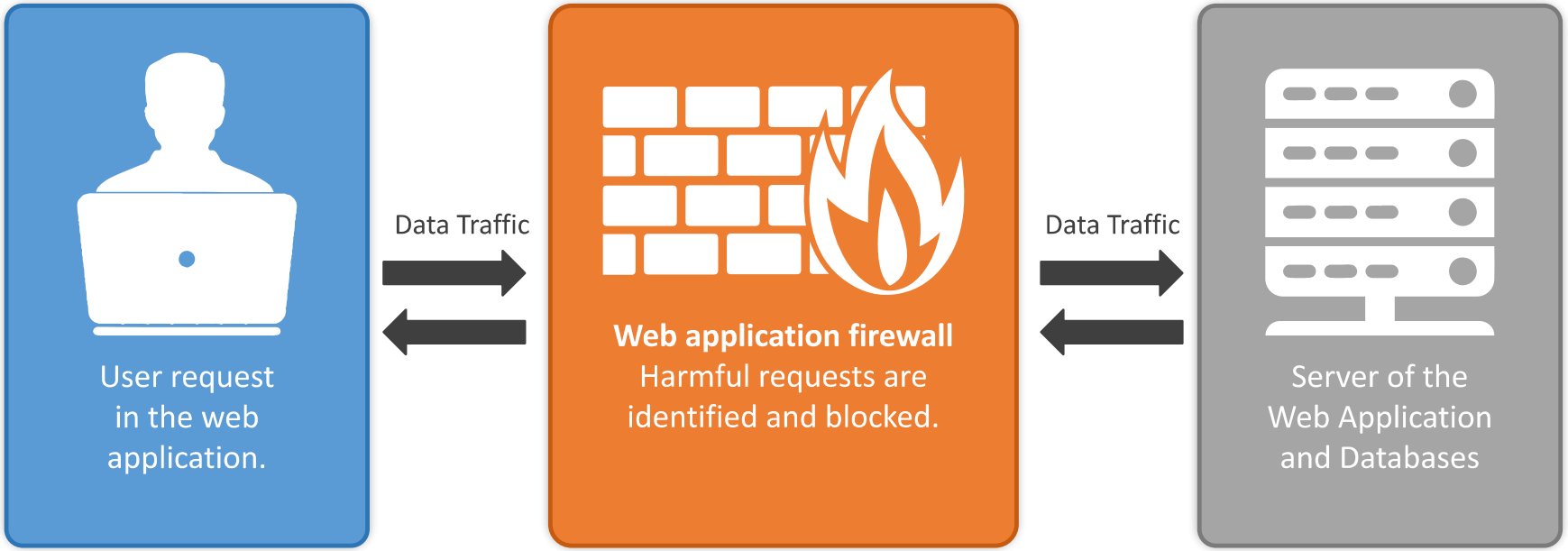
- Common WAF Features

Web Application Firewall

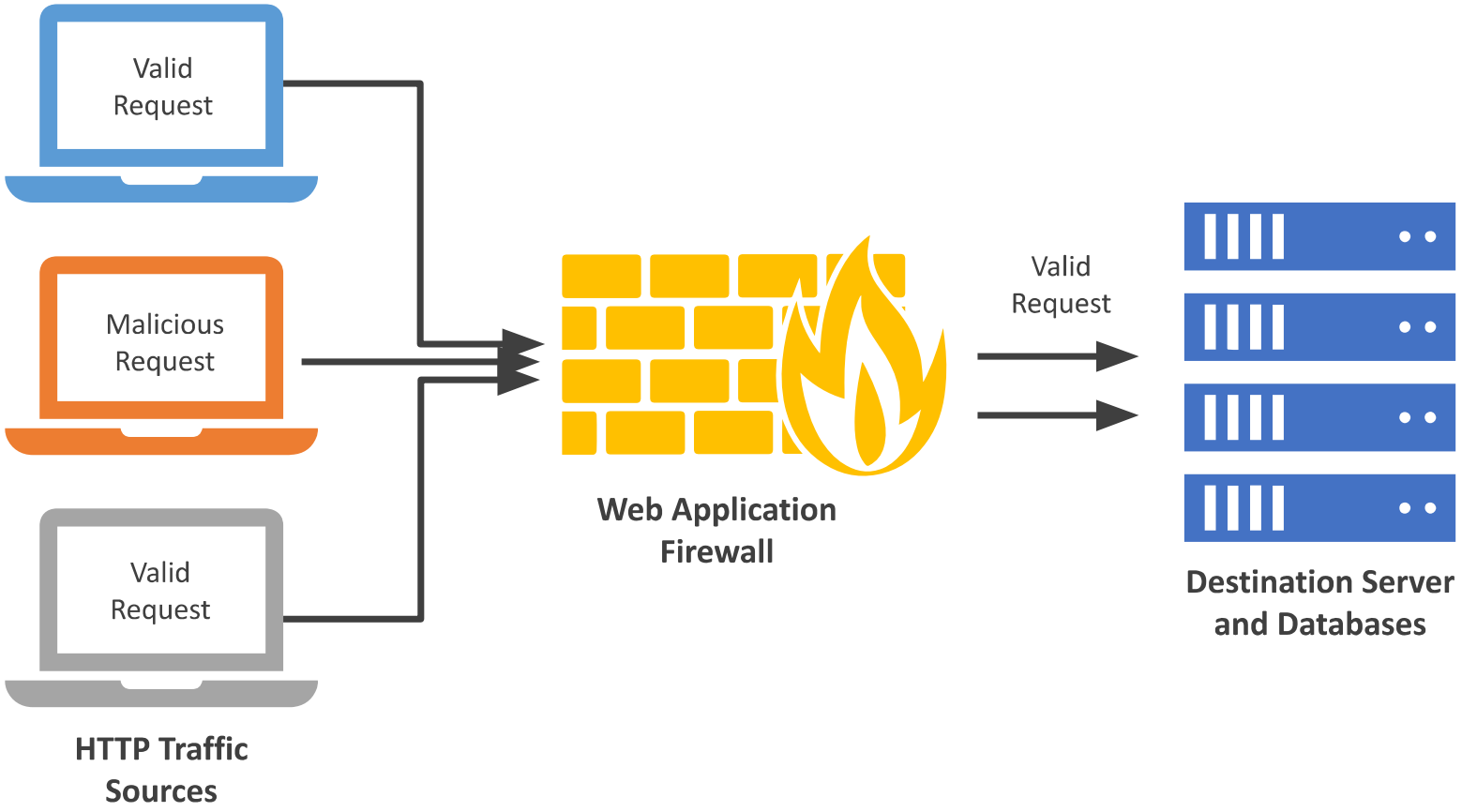
- Network Protocol Filtering
- HTTP Protocol Filtering
- Stateful Connection Monitoring
- High Availability Support
- Session Management Controls
- Cookie Monitoring/ Protection
- Hidden Field Enforcement
- Honeypot/ Honeynet Integration

WEB APPLICATION FIREWALL

- How does it Work?

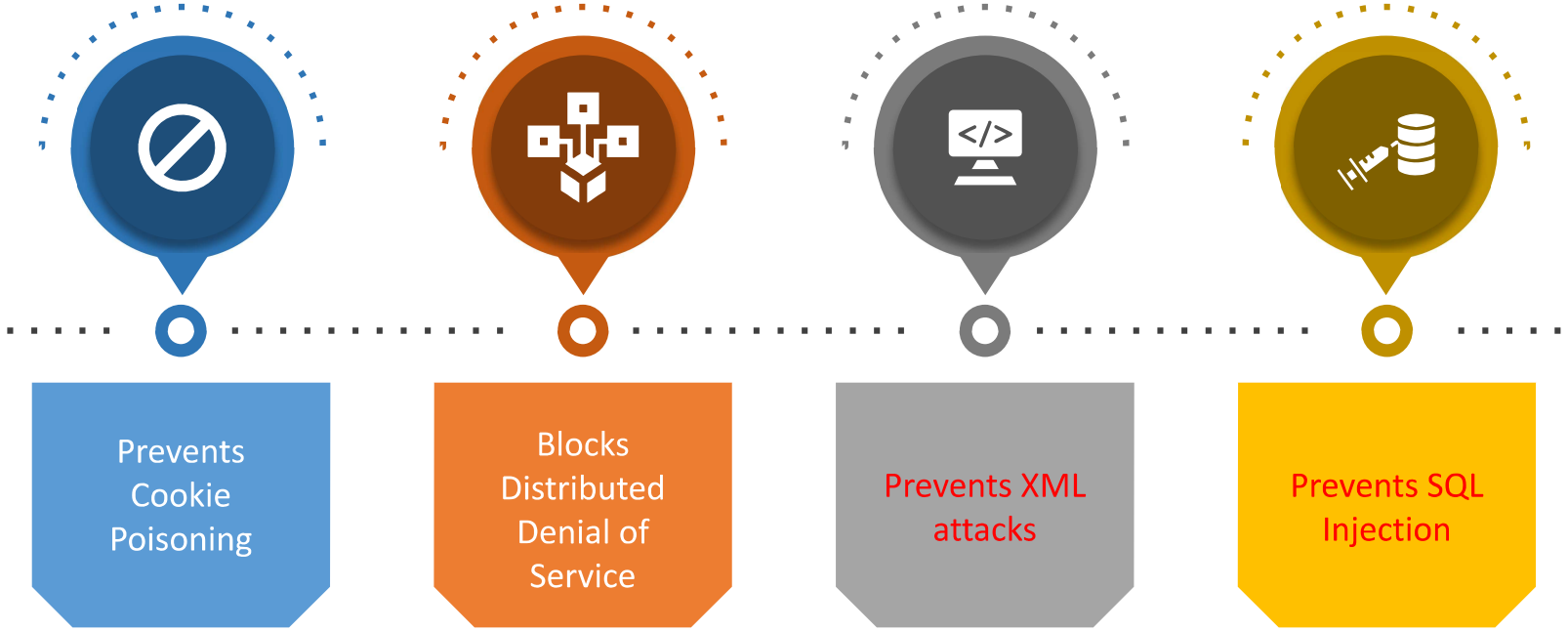


WEB APPLICATION FIREWALL

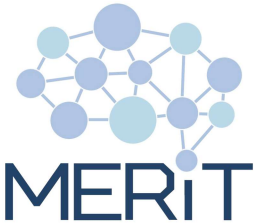


WEB APPLICATION FIREWALL

- Benefits of a Web Application Firewall



Summary



This unit explores injection attacks—such as SQL, XML, and Code Injection—that exploit insecure input handling. It emphasizes the importance of:

Prepared statements and ORMs to prevent SQL injection.
Input validation and whitelisting to control user input.
Escaping special characters to protect backend queries.
Safe XML parsing to avoid XXE and XML injection.
Web Application Firewalls (WAFs) to detect and block malicious requests targeting web applications.

Understanding these techniques is essential for building secure, resilient database applications.