🧠 Lecture 22 – SQL Injection (SQLi) and Database Exploitation

---

🔹 1. Real-World Example: Akamai DDoS Case

Before diving into SQL injection, let's recall a real event related to **Akamai** (a global CDN and DDoS protection company):

- **Akamai** once faced a massive **DDoS attack**, where they had to handle over **419 Terabytes (TB)** of malicious traffic.
- Because Akamai has **thousands of distributed servers**, they managed to **absorb and mitigate** the attack successfully.
- This demonstrates the importance of **scalable architecture and redundancy** in modern cybersecurity defense.

🔹 2. What is SQL? (Structured Query Language)

- **SQL (Structured Query Language)** is used to **store, manage, and retrieve data** from **relational databases**.
- Databases organize data into **tables** — each table has **rows (records)** and **columns (fields)**.

🔸 Key Terms:

Term	Description
Primary Key	A unique identifier for each record in a table (e.g., user ID).
Foreign Key	A key used to link two tables together (creates relationships).
Relational Database	A database that links multiple tables through relationships.

📌 Example:

- **Table 1:** Users → (UserID, Username, Password)
- **Table 2:** Orders → (OrderID, UserID, ProductID)

The **UserID** is a **primary key** in Users and a **foreign key** in Orders.

🔹 **3. What is SQL Injection (SQLi)?**

💥 **Definition:**

SQL Injection is a **web-based attack** that allows an attacker to **interfere with the SQL queries** an application makes to its database.

Simply put — the attacker **injects malicious SQL code** into a vulnerable input field (like a login box or search bar) to:

- Retrieve sensitive information,
- Modify or delete data,
- Bypass authentication,
- Or even gain full database control.

💡 **Example:**

Suppose a website login form runs this query in the backend:

SELECT * FROM users WHERE username = 'john' AND password = '12345';

If the input fields are **not properly sanitized**, an attacker could type:

- Username: ' or 1=1 -- -
- Password: *(anything)*

The resulting query becomes:

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

Explanation:

- OR 1=1 → Always true condition.
- -- - → Comment out the rest of the query.

👉 This tricks the database into **logging the attacker in without knowing the real password!**

🔹 4. How SQL Injection Works (Step-by-Step)

1. **Attacker finds a vulnerable input field** — like a login form, search bar, or URL parameter.
2. **Injects SQL code** into that input box.
3. The application takes that input and **directly includes it in an SQL query**.
4. The database **executes** that malicious query because **input validation is missing**.
5. The attacker then gains **unauthorized access** or extracts **sensitive data**.

🔹 5. Why SQL Injection Happens

✅ **Root Causes:**

- Poor **input validation**.
- Concatenating user input directly in SQL statements.
- Not using **parameterized queries** or **prepared statements**.
- Outdated web applications or old frameworks.
- Weak database error handling (leaking information).

🔹 6. Types of SQL Injection Attacks

SQL Injection has **four main types** — each works differently:

🧩 1️⃣ Error-Based SQL Injection

- Relies on **database error messages** to extract information.
- The attacker intentionally sends wrong queries and **analyzes the returned errors** to understand database structure.

📌 Example:

' ORDER BY 5 -- -

If the website returns “Unknown column 5”, the attacker learns how many columns exist.

🧠 **Purpose:** Find database names, table names, and structure.

🧩 2️⃣ Union-Based SQL Injection

- Uses the SQL **UNION operator** to **combine results from multiple queries**.
- Attacker can retrieve data from other tables in the same database.

📌 Example:

' UNION SELECT username, password FROM users -- -

This merges the attacker’s query with the legitimate one and displays user credentials.

🧩 3️⃣ Blind SQL Injection (Boolean-Based)

- When the website **doesn’t display error messages**, the attacker tests conditions that return **True or False**.

📌 Example:

' AND 1=1 -- -

✅ Page loads normally → condition True.

' AND 1=2 -- -

❌ Page fails → condition False.

From this, the attacker can infer **whether the query executed successfully** and slowly **extract data bit by bit**.

🧩 ⏱️ Time-Based Blind SQL Injection

- When no visible response difference exists, attackers use **time delays** to test conditions.

📌 Example:

' OR IF(1=1, SLEEP(5), 0) -- -

If the page takes 5 seconds longer to respond, the attacker knows the condition was **True**.

Used to extract data **without any visible output** — based purely on **response time**.

🔹 7. Tools Used for SQL Injection

🛠️ sqlmap (Automated SQLi Tool)

- **sqlmap** is an open-source penetration testing tool that automates the process of detecting and exploiting SQL injection flaws.
- It supports **multiple databases** (MySQL, Oracle, PostgreSQL, Microsoft SQL Server, etc.).

⚙️ Common sqlmap Commands:

Command	Explanation
sqlmap -h	Shows all available options and help menu.
sqlmap --url http://testphp.vulnweb.com --crawl 2 -batch --threads 3	Scans the website for vulnerabilities and crawls 2 levels deep.
sqlmap --url http://testphp.vulnweb.com --crawl 2 -batch --threads 3 -dbs	Lists all databases found on the target.

Command	Explanation
sqlmap --url http://testphp.vulnweb.com --crawl 2 -batch --threads 3 -D acuart -T artists --dump	Dumps (downloads) data from table artists in database acuart.

📌 **Website used for practice:**
http://testphp.vulnweb.com/ → A legal testing site for learning SQL injection.

🔹 8. Manual SQL Injection Example

If a website is vulnerable, you can bypass login like this:

Input in Login Page:

Username: ' or 1=1 -- -

Password: admin

Resulting SQL Query:

SELECT * FROM accounts WHERE username='' OR 1=1 -- -' AND password='admin';

Because 1=1 is always true, the database grants access — **no valid username or password needed!**

🔹 9. Real Impact of SQL Injection

A successful SQLi attack can:

- Steal **usernames, passwords, and financial data**.
- **Modify or delete** entire databases.
- Bypass **login authentication**.
- Execute **remote system commands** in advanced cases.
- Lead to **complete system compromise**.

🔹 10. Prevention and Defense Against SQL Injection

✅ 1. Use Parameterized Queries (Prepared Statements)

- Never directly concatenate user input into SQL queries.
- Use parameter binding to keep user input separate from query structure.

📌 **Example (Safe Code):**

```
cursor.execute("SELECT * FROM users WHERE username = ? AND password = ?", (user, pwd))
```

✅ 2. Input Validation and Sanitization

- Validate user input type (e.g., numbers, emails, etc.).
- Reject special characters like ', ", --, ;, #.
- Use whitelisting over blacklisting.

✅ 3. Use Stored Procedures

- Execute pre-defined queries instead of dynamic SQL strings.

✅ 4. Limit Database Privileges

- Application users should have **read/write access only**, not admin privileges.

✅ 5. Error Handling

- Don’t display **database error messages** to users — log them securely instead.
- Custom error pages prevent attackers from learning about your database structure.

✅ 6. Web Application Firewall (WAF)

- Tools like **Akamai Kona**, **Cloudflare WAF**, **AWS WAF**, and **ModSecurity** filter malicious SQL queries before they reach your server.

✅ 7. Regular Security Audits

- Use vulnerability scanners and penetration tests.
- Keep **databases and web frameworks updated** to patch SQLi vulnerabilities.

🔹 11. Example: Secure vs Vulnerable Code

❌ Vulnerable PHP Code:

```
$query = "SELECT * FROM users WHERE username = '" . $_POST['user'] . "' AND password = '" . $_POST['pass'] . "'";
```

✅ Secure PHP Code:

```
$stmt = $pdo->prepare("SELECT * FROM users WHERE username = :user AND password = :pass");
$stmt->execute(['user' => $_POST['user'], 'pass' => $_POST['pass']]);
```

🔹 12. Summary Table

Concept	Description
SQL Injection	Injecting malicious SQL commands through user input.
Types	Error-Based, Union-Based, Blind Boolean, Blind Time-Based.
Tool	sqlmap – used to detect and exploit SQLi automatically.
Bypass Login Query	' or 1=1 -- -
Impact	Data theft, authentication bypass, full DB compromise.

Concept	Description
Prevention	Parameterized queries, validation, WAF, limited privileges.

✅ 13. Key Takeaways

1. SQL Injection is one of the **most dangerous and common web vulnerabilities**.
2. Always **validate and sanitize user inputs**.
3. Use **parameterized queries** or **ORM frameworks** (like SQLAlchemy, Hibernate).
4. Never display **database errors** publicly.
5. Use **WAFs** and **security patches** regularly.
6. Conduct **vulnerability testing** using tools like **sqlmap** and fix issues immediately.