MARS

Machine-learning AI per Riconoscere SQLi

Andrea Fiorella
Vito Claudio Giammaria
Danilo Campanale

Dipartimento di Informatica
Università degli studi di Bari

Tavola dei Contenuti

Capitolo 1

Introduzione

La sicurezza delle applicazioni web è un elemento fondamentale per il successo delle imprese. Secondo la "**Top 10 Web Application Security Risks**" pubblicata dalla fondazione **OWASP** nel 2021 [9], le vulnerabilità di tipologia "**Injection**", come ad esempio la "**SQL Injection**", rappresentano ancora oggi vulnerabilità critiche e ampiamente diffuse.

La SQL Injection (**SQLi**) permette ad un utente malevolo di manipolare le query SQL, inviate da una applicazione ad un database, inserendo comandi arbitrari per poter raggiungere un obiettivo. Questo tipo di attacco può portare conseguenze devastanti per una impresa e i suoi clienti, come l'accesso non autorizzato a dati sensibili, la compromissione dell'integrità delle informazioni, la perdita di riservatezza o il passaggio ad attacchi più complessi e dannosi.

Il problema è aggravato dalla rapida evoluzione della tecnologia che ha come conseguenza la proliferazione di **sistemi legacy**. Tali sistemi, costruiti decenni fa, rappresentano una sfida in quanto il loro codice non può essere facilmente aggiornato o riscritto per correggere eventuali vulnerabilità. I costi associati a queste operazioni correttive, sia in termini economici che di tempo, sono molto spesso proibitivi, portando molte organizzazioni a sottovalutare o ignorare del tutto il problema.

Molte di queste applicazioni vulnerabili rimangono tutt'ora esposte ad attacchi, creando una superficie di minaccia che utenti malevoli possono sfruttare con strumenti sempre più sofisticati.

L'impatto di una vulnerabilità di questo tipo può essere colossale e devastante per aziende di ogni dimensione, dal furto di dati personali, finanziari o aziendali, alla perdita della fiducia dei clienti, fino ad azioni legali e danni alla reputazione incalcolabili.

1.1 Tipologie di SQL Injection

La SQLi è suddivisa in diverse classi [12], a seconda della tecnica utilizzata dall'attaccante per sfruttare la vulnerabilità e raggiungere il suo obiettivo. Le principali classi sono:

- **SQL Injection Basata su Errori (Error-Based SQLi)**: questa tecnica sfrutta i messaggi di errore prodotti dal DBMS utilizzando input malformati per ottenere informazioni sensibili, come la versione del DBMS, la struttura delle tabelle e i nomi delle colonne.
- **Blind SQL Injection (Blind SQLi)**: si verifica quando il DBMS non restituisce messaggi di errore o risposte dirette, rendendo necessarie tecniche indirette per estrarre informazioni, come un oracolo. La Blind SQLi si divide in:
  - **Boolean-Based**: l'attaccante manipola le query per ottenere risposte binarie (True/-False), osservando il comportamento dell'applicazione per dedurre informazioni.
  - **Time-Based**: l'attacco induce ritardi intenzionali nelle query, analizzando il tempo di risposta del server per ricavare informazioni.
- **SQL Injection a Unione (Union-Based SQLi)**: sfrutta l'istruzione `UNION` per combinare i risultati di una query legittima con quelli di una query arbitraria inserita dall'attaccante, consentendo di accedere a informazioni riservate contenute nel database.
- **SQL Injection di Secondo Ordine (Second-Order SQLi)**: questa tecnica utilizza dati precedentemente memorizzati nel database apparentemente innocui, come ad esempio un username malformato, che vengono successivamente usati in una query non adeguatamente sanificata, concretizzando l'attacco.

1.2 Stato dell'arte per la protezione dalla SQL Injection

Lo stato dell'arte nella protezione contro le SQLi si basa su una combinazione di pratiche di sviluppo sicuro, strumenti e strategie pro-attive per ridurre i rischi associati a queste vulnerabilità [8]. Tra le tecniche più diffuse vi sono:

1. **Web Application Firewall (WAF)**:

   I WAF sono strumenti che filtrano il traffico in entrata per identificare e bloccare richieste dannose, inclusi tentativi di SQLi. Operano tipicamente tramite:

   - **Espressioni regolari e parole chiave:** Analizzano le richieste per individuare modelli tipici di attacchi, come parole riservate SQL (`DROP`, `SELECT`, ecc.).
   - **Blacklist e whitelist:** Impediscono l'esecuzione di specifiche query note per essere pericolose, permettendo solo input ritenuti sicuri.

2. **Sanifica e validazione degli input**

   Una delle migliori pratiche è prevenire l'inserimento di caratteri non desiderati nei campi utente. La sanifica trasforma o rimuove caratteri pericolosi, mentre la validazione verifica che l'input sia conforme alle aspettative. L'uso di **query parametrizzate** o **stored procedures** riduce drasticamente la vulnerabilità agli attacchi, eliminando la possibilità che l'input dell'utente venga interpretato come codice.

3. **Logging e monitoraggio**

   Molte soluzioni includono sistemi avanzati di logging che registrano richieste sospette per permettere analisi post-evento.

4. **Integrazione della sicurezza nei processi di sviluppo software**

   Grazie all'approccio "**secure-by-design**", la sicurezza è integrata fin dalle prime fasi del ciclo di vita dello sviluppo del software. Prima di scrivere il codice di un prodotto, gli sviluppatori analizzano i rischi potenziali e prevedono dei requisiti di sicurezza. Utilizzando tecniche che applicano automaticamente misure di protezione, combinate con strumenti e metodologie, come l'analisi statica e l'analisi dinamica del codice, è possibile rilevare eventuali vulnerabilità durante lo sviluppo, consentendo la loro risoluzione prima della distribuzione in produzione.

1.3 Limiti delle soluzioni attuali

Nonostante i progressi significativi nella protezione contro la SQLi, esistono ancora rischi non completamente mitigati dalle tecnologie e metodologie attuali.

Strumenti, come Web Application Firewall e l'impiego di pratiche di sviluppo sicuro, sono efficaci ma richiedono molte risorse per essere adottati e mantenuti in modo corretto, rendendo difficile il loro impiego per piccole e medie imprese con budget limitati.

I sistemi legacy rappresentano un ulteriore problema, infatti un grande numero di organizzazioni opera con applicazioni e infrastrutture obsolete che non supportano le pratiche moderne di sicurezza, come l'uso di query parametrizzate o la revisione automatizzata del codice. Modificare, riscrivere e riprogettare tali sistemi per eliminare le vulnerabilità è spesso impossibile, per i costi, per la complessità e per il rischio di introdurre errori in sistemi critici.

Molte soluzioni sono progettate per proteggere applicazioni moderne e non forniscono una protezione retroattiva ed economica per ambienti già esistenti. Questo evidenzia la necessità di un nuovo approccio che sia al contempo economico, semplice da implementare e capace di operare sui sistemi legacy, senza richiedere risorse e competenze significative.

Capitolo 2

Metodologia

La soluzione che proponiamo è un sistema di protezione dalla SQLi, basato su un modello di intelligenza artificiale in grado di analizzare in tempo reale i parametri delle richieste inviate dagli utenti.

Il cuore della soluzione è un modello di machine learning che è in grado di identificare correttamente i parametri che contengono un payload di SQLi, separandoli da quelli che contengono dati conformi. Questo approccio supera i limiti dei tradizionali WAF, che spesso falliscono nel riconoscere varianti nuove o sofisticate di SQLi, o quando le richieste sono manipolate in modo da eludere le regole statiche. Il modello potrebbe essere esteso per rilevare anche altre tipologie di injection, come **Cross-Site Scripting** (**XSS**) e **Local File Inclusion** (**LFI**), garantendo una protezione più ampia.

Il sistema è progettato per essere facilmente integrato in qualsiasi infrastruttura, senza la necessità di modificare il codice delle applicazioni sottostanti. Questo lo rende particolarmente utile per sistemi legacy. Il sistema non solo protegge da SQLi, ma offre anche un livello di visibilità e monitoraggio che consente di adattarsi rapidamente a nuove minacce.

Questa soluzione prova a migliorare e semplificare la protezione contro le SQLi rispetto ai sistemi tradizionali offrendo un approccio più flessibile e integrabile in qualsiasi tipo di architettura IT, riducendo significativamente i costi e il tempo necessari per garantire una sicurezza continua e aggiornata.

2.1 Model Development

2.1.1 Dataset

I dati utilizzati per lo sviluppo dei modelli di intelligenza artificiale sono classificati come: **allowed** e **sqli**.

2.1.1.1 Raccolta dei dati classificati come "allowed"

Per la raccolta dei dati classificati con label **allowed** sono state utilizzate tre fonti.

La prima fonte è costituita dal traffico di rete generato da un client durante l'interazione con diverse applicazioni web. Questo traffico è stato intercettato e registrato utilizzando **mitmproxy** [6], uno strumento open-source che consente di monitorare in tempo reale il traffico HTTP e HTTPS. Prima di arrivare a mitmproxy, è stato tentato l'utilizzo del proxy **Squid**, un altro strumento noto per la gestione del traffico HTTP. Tuttavia, durante le prove, Squid si è rivelato inadatto alle nostre esigenze specifiche. In particolare, non siamo riusciti a configurarlo in modo da salvare i dettagli dei parametri effettivi inviati nelle richieste HTTP.

mitmproxy, al contrario, si è dimostrato più versatile grazie alla possibilità di estenderne le funzionalità tramite script Python. Questo ci ha permesso di raccogliere un log completo e dettagliato, rispondendo in maniera ottimale alle necessità del progetto.

La scelta di mitmproxy è stata motivata dalla necessità di ottenere un campione rappresentativo di richieste benigne provenienti da un client reale. Attraverso questo approccio, è stato possibile raccogliere dati autentici e realistici che riflettono i comportamenti legittimi degli utenti. Questi dati includono una vasta gamma di parametri inviati durante normali sessioni di utilizzo, come campi di input provenienti da form, parametri di URL e dati di interazione con le varie applicazioni web.

La seconda fonte di dati classificati come **allowed** è stata la famosa wordlist **rockyou** [11], una raccolta di milioni di password che è diventata un riferimento standard in ambito di cybersecurity. La wordlist è stata scelta per rappresentare parametri che sono dati d'input e non contengono elementi malevoli o indicativi di un attacco.

La scelta della wordlist rockyou è motivata dalla sua natura eterogenea, che include una varietà di pattern utili per addestrare il modello a distinguere input legittimi da potenziali payload di SQLi. Pur essendo originariamente una raccolta di password, la diversità dei contenuti in rockyou la rende idonea per simulare parametri di input che possono essere inviati durante l'interazione con applicazioni web, fornendo così un campione rappresentativo per il nostro dataset.

Utilizzare dati da rockyou ha permesso di ampliare la varietà dei dati classificati come legittimi nel dataset, migliorando la capacità del modello di apprendere la differenza tra comportamenti normali e anomali. Inoltre, l'uso di una wordlist preesistente ha consentito di

risparmiare tempo e risorse nella fase di raccolta dei dati, sfruttando una risorsa ampiamente riconosciuta e utilizzata nella comunità della sicurezza informatica.

La terza e ultima fonte è stata generata utilizzando uno script Python sviluppato appositamente. Questo script produce sequenze di parole simili a frasi in linguaggio naturale, che, pur non avendo un senso compiuto, contengono le parole chiave del linguaggio SQL, come `SELECT`, `FROM`, e `WHERE`. Lo scopo di includere queste sequenze nel dataset è insegnare al modello che le parole chiave SQL, di per sé, non sono indicative di un attacco e possono essere usate in contesti legittimi, ad esempio come contenuto generato dagli utenti (**User-Generated Content**).

Questo approccio risponde a una lacuna nelle altre fonti di dati raccolte. I dati intercettati tramite *mitmproxy* e quelli derivati dalla wordlist *rockyou* non coprono scenari che rappresentano campioni di contenuti testuali generati dagli utenti, come commenti su forum o post sui social media. Poiché queste situazioni costituiscono un'ampia parte delle interazioni in ambienti web moderni, era essenziale includere una simulazione di tali dati nel dataset. Lo script Python è stato ritenuto una soluzione pratica, rapida ed efficace per creare input che simulino tali comportamenti.

2.1.1.2 Raccolta dei dati classificati come "sqli"

Per la raccolta dei dati classificati con label **sqli** sono state utilizzate due fonti.

La prima fonte è stata ottenuta utilizzando **SQLMap** [13], uno strumento open-source progettato per automatizzare il rilevamento e lo sfruttamento delle vulnerabilità SQLi.

SQLMap è stato scelto per la sua flessibilità, la vasta gamma di tecniche supportate e la capacità di generare una varietà di payload per testare diversi tipi di SQLi su diversi tipi di DBMS. Lo strumento è stato utilizzato per inviare richieste malevole a un endpoint scritto in Python (flask), eseguito in locale sul nostro dispositivo, in modo da poter memorizzare le caratteristiche di ogni richiesta per un'analisi successiva.

Questi payload sono stati poi riutilizzati su un'applicazione volutamente vulnerabile, la **Damn Vulnerable Web Application** (**DVWA**), eseguita anch'essa in locale. Questo processo ha permesso di osservare e monitorare l'effetto dei payload sull'applicazione target, consentendo di organizzare i payload di SQLi in base al tipo di risposta ottenuta dal server.

Ad esempio, le *blind SQLi time-based* sono state identificate osservando un ritardo significativo nei tempi di risposta del server, suggerendo che il payload aveva influenzato il tempo di esecuzione delle query. Le *SQLi error-based*, invece, sono state riconosciute grazie ai messaggi di errore restituiti dal DBMS. Infine, le *SQLi union-based* sono state classificate in base alla capacità del payload di manipolare le query consentendo di estrarre dati sensibili e visualizzarli all'interno della pagina web attaccata.

I parametri che non generavano alcun effetto sul comportamento del server sono stati esclusi dal dataset, garantendo che solo le richieste malevole con un impatto significativo fossero incluse.

Questo approccio, che valuta non solo la richiesta dell'utente ma anche la risposta del server, è stato fondamentale per costruire un dataset realistico e dettagliato, in grado di rappresentare accuratamente le diverse tipologie di SQLi.

La seconda fonte dei payload classificati come **sqli** è stata derivata da una piccola parte del dataset utilizzato per l'addestramento di **waf-a-mole** [2], un progetto open-source dedicato a testare e aggirare le regole statiche implementate nei WAF. Questo dataset, disponibile pubblicamente, include query SQL complete progettate per contenere vulnerabilità di injection. Tuttavia, poiché il nostro obiettivo era concentrarci esclusivamente sulla parte dell'injection, è stato necessario effettuare un pre-processing per estrarre solo la porzione pertinente di ciascuna query.

Una volta estratti i payload relativi alle SQLi, questi sono stati testati contro l'applicazione web volutamente vulnerabile **DVWA**, come già fatto con i dati raccolti tramite **SQLMap**. Anche per questi payload, quelli che non causavano alcun effetto sull'applicazione sono stati rimossi, garantendo che il dataset finale fosse composto esclusivamente da richieste rilevanti e utili per l'addestramento del modello.

L'uso di una piccola parte del dataset di waf-a-mole ha permesso di integrare i nostri dati con altre permutazioni di payload malevoli, aumentando la varietà e la rappresentatività delle SQL injection incluse nel nostro dataset, contribuendo così a migliorare l'accuratezza e la generalizzazione del modello di intelligenza artificiale sviluppato.

2.1.1.3 Creazione del dataset

Il dataset finale comprende 430'000 righe classificate e divise esattamente a metà tra **sqli** e **allowed**.

È stato creato utilizzando i dati raccolti in precedenza e uno script scritto in python che:

1. **Importa le librerie e crea il DataFrame**: Per iniziare, vengono importate le librerie necessarie e viene creato un DataFrame vuoto, che sarà popolato con i dati delle varie sorgenti.

```
import pandas as pd
import random
import csv

dataframe = pd.DataFrame(columns=["Testo", "Label"])

def add_list_to_dataframe(input_list, label):
    global dataframe
    new_data = pd.DataFrame({"Testo": input_list, "Label": [
       label] * len(input_list)})
    dataframe = pd.concat([dataframe, new_data],
       ignore_index=True)
```

   La funzione `add_list_to_dataframe` consente di aggiungere facilmente una lista di dati con l'etichetta corrispondente (allowed o sqli) al DataFrame.

2. **Lettura delle righe uniche dai file sorgenti**: I file contenenti le diverse sorgenti di dati vengono letti tramite una funzione che restituisce solo righe uniche, eliminando eventuali duplicati.

```
def read_unique_lines(file_path):
    unique_lines = set()
    with open(file_path, 'r', encoding='utf-8', errors='
       ignore') as file:
        for line in file:
            unique_lines.add(line.strip())
    return unique_lines, len(unique_lines)
```

   Questa funzione utilizza un `set` per assicurare l'unicità delle righe.

3. **Definizione delle fonti e calcolo delle proporzioni**: Le fonti dei dati vengono definite, e si calcolano le proporzioni per determinare quante righe devono essere estratte da ciascuna fonte.

```
# Input files paths
allowed_network_path = "data/allowed_network.txt"
allowed_phrases_path = "data/allowed_phrases.txt"
allowed_wordlist_path = "data/rockyou.txt"
sqli_sqlmap_path = "data/sqli_sqlmap.txt"
sqli_wafamole_path = "data/sqli_wafamole.txt"

# Input data and counters
allowed_network, n_allowed_network = read_unique_lines(
    allowed_network_path)
allowed_phrases, n_allowed_phrases = read_unique_lines(
    allowed_phrases_path)
allowed_wordlist, n_allowed_wordlist = read_unique_lines(
    allowed_wordlist_path)
sqli_sqlmap, n_sqli_sqlmap = read_unique_lines(
    sqli_sqlmap_path)
sqli_wafamole, n_sqli_wafamole = read_unique_lines(
    sqli_wafamole_path)

# Data and counters for dataset
needed_rows = 430000
needed_sqli = int(needed_rows * 0.5)
needed_allowed = needed_rows - needed_sqli
needed_sqli_from_sqlmap = int(needed_sqli * 0.6)
needed_sqli_from_wafamole = needed_sqli -
    needed_sqli_from_sqlmap
needed_allowed_from_network = int(needed_sqli * 0.20)
needed_allowed_from_wordlist= int(needed_sqli * 0.40)
needed_allowed_from_phrases = needed_allowed - (
    needed_allowed_from_network + needed_allowed_from_wordlist
    )
```

Questa parte calcola quante righe devono essere prese da ciascuna fonte per mantenere le proporzioni desiderate.

4. **Selezione casuale e popolamento del DataFrame**: Le righe vengono selezionate casualmente da ciascuna sorgente e aggiunte al DataFrame.

```
# Sample data taken from input
sample_sqli_sqlmap = random.sample(list(sqli_sqlmap),
   needed_sqli_from_sqlmap)
sample_sqli_wafamole = random.sample(list(sqli_wafamole),
   needed_sqli_from_wafamole)
sample_allowed_network = random.sample(list(allowed_network)
   , needed_allowed_from_network)
sample_allowed_phrases = random.sample(list(allowed_phrases)
   , needed_allowed_from_phrases)
sample_allowed_wordlist = random.sample(list(
   allowed_wordlist), needed_allowed_from_wordlist)

# Adding data to dataframe
add_list_to_dataframe(sample_sqli_sqlmap, "sqli")
add_list_to_dataframe(sample_sqli_wafamole, "sqli")
add_list_to_dataframe(sample_allowed_network, "allowed")
add_list_to_dataframe(sample_allowed_phrases, "allowed")
add_list_to_dataframe(sample_allowed_wordlist, "allowed")
```

La funzione `random.sample` garantisce che i dati siano selezionati in modo casuale da ogni sorgente.

5. **Randomizzazione e salvataggio del dataset**: Infine, il dataset viene randomizzato e salvato in un file CSV pronto per essere utilizzato.

```
# Randomizing and saving dataset
dataframe = dataframe.sample(frac=1).reset_index(drop=True)
dataframe.to_csv("dataset_430k.csv", index=False, quoting=
   csv.QUOTE_ALL)
```

Questo passaggio assicura che i dati nel dataset siano mescolati, evitando correlazioni non intenzionali tra righe consecutive.

I dati presenti nel dataset da 430'000 righe sono quindi divisi in:

- `50%` allowed, di cui:
  - `20%` proviene dal **traffico di rete**
  - `40%` proviene dalla wordlist **rockyou**
  - `40%` proviene dalle frasi generate che simulano il linguaggio naturale
- `50%` sqli, di cui:
  - `60%` proviene dai payload di **SQLMap**
  - `40%` proviene dal dataset degli attacchi di **waf-a-mole**

Fonte	**Numero di righe**
allowed_network	43'000
allowed_phrases	86'000
allowed_wordlist	86'000
TOTALE ROW CLASSIFICATE allowed	**215'000**
sqli_sqlmap	150'500
sqli_wafamole	64'500
TOTALE ROW CLASSIFICATE sqli	**215'000**

Tabella 2.1: Numero di righe nel dataset divise per fonte.

2.1.2 Preprocessing dei dati (RNN)

Per la preparazione dei dati testuali in ingresso al modello, è stata sviluppata una funzione di tokenizzazione personalizzata, denominata **getTokenizer**, che si occupa di pre-processare i testi e trasformarli in sequenze numeriche. La funzione utilizza una combinazione di espressioni regolari (regex), per identificare parole chiave SQL, simboli speciali e parole del testo naturale. Questo approccio ci permette di distinguere tra parti di query SQL e testo generico, fondamentale per l'analisi e il rilevamento di SQL injection.

```python
def getTokenizer(max_words, max_len, texts):

    sql_keywords = r"\b(SELECT|INSERT|UPDATE|DELETE|DROP|TABLE|FROM|WHERE|AND|OR|UNION|LIKE|IS|NULL|BETWEEN|HAVING)\b"
    sql_symbols = r"[=\-+\*/<>;,'\"|()`!@#$%^&*[]"

    # Combinare tutti i pattern in uno
    combined_regex = f"({sql_symbols}|{sql_keywords}|[\w]+)"

    all_tokens = []
    for i, text in enumerate(texts):
        tokens = re.findall(combined_regex, text)
        tokens = [' '.join(token) if isinstance(token, tuple) else token for token in tokens]
        all_tokens.append(' '.join(tokens))

    tokenizer = Tokenizer(num_words=max_words, filters='')
    tokenizer.fit_on_texts(all_tokens)
    sequences = tokenizer.texts_to_sequences(all_tokens)

    data = pad_sequences(sequences, maxlen=max_len)
```

1. **Estrazione dei token:** La funzione inizia definendo due espressioni regolari per catturare le parole chiave e simboli del linguaggio SQL. Unendo queste due espressioni regolari con un terzo pattern che cattura parole generiche, la funzione è in grado di identificare e separare le parole chiave e i simboli SQL dal testo naturale.

2. **Pre-processing dei testi:** Per ogni testo nel dataset, la funzione applica la regex combinata utilizzando il metodo **re.findall**, che estrae tutti i token (parole chiave SQL, simboli e parole generiche). I token ottenuti vengono poi concatenati in una singola stringa, separati da spazi, creando una versione "tokenizzata" del testo originale. Questo approccio facilita l'analisi dei singoli componenti di una query SQL e del linguaggio naturale.

3. **Creazione del tokenizer:** Il tokenizer di Keras, viene addestrato sui testi tokenizzati tramite il metodo **fit_on_texts**, che costruisce un dizionario di 3000 token in cui ad ognuno di essi viene associato un indice numerico che rappresenta la sua frequenza nel dataset. Successivamente, il metodo **texts_to_sequences** converte i testi tokenizzati in sequenze di numeri.

4. **Padding delle Sequenze:** Poiché le sequenze numeriche potrebbero avere lunghezze differenti, la funzione applica il padding utilizzando il metodo **pad_sequences** di Keras. Questo assicura che tutte le sequenze abbiano la stessa lunghezza (max_len = 60, la scelta è stata fatta calcolando la media delle lunghezze).

2.1.3 Creazione del modello (RNN)

Il modello creato è una rete neurale sequenziale (Sequential Model), che utilizza un'architettura di tipo RNN (Recurrent Neural Network) con un LSTM bidirezionale. Si basa su una combinazione di embedding layer e LSTM per estrarre e modellare le caratteristiche sequenziali delle sqli e delle stringhe lecite. Codice del modello in Figura 2.1

```
model = Sequential()
model.add(Embedding(input_dim=MAX_WORDS, output_dim=128, input_length=MAX_LEN, embeddings_regularizer=l2_EMBEDDING))
model.add(Dropout(0.2))  # Dropout dopo l'embedding
model.add(Bidirectional(LSTM(128, return_sequences=False, kernel_regularizer=l2_LSTM)))
model.add(Dropout(0.5))  # Dropout dopo la LSTM
model.add(Dense(2, activation='softmax', kernel_regularizer=l2_DENSE))
```

Figura 2.1: Modello.

2.1.3.1 Input Layer

L'ingresso del modello accetta sequenze di token di lunghezza 60 con batch variabile.

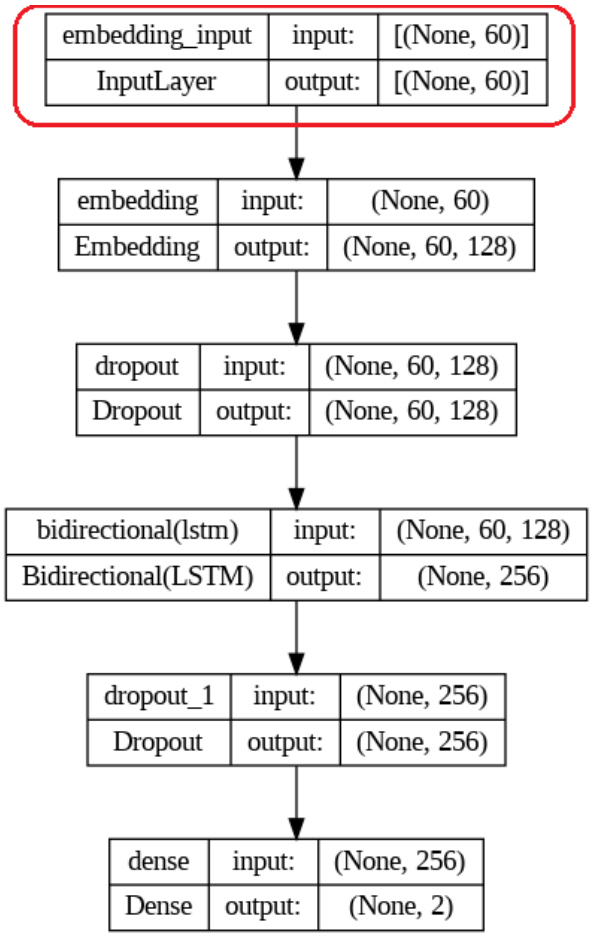

Figura 2.2: Input Layer.

2.1.3.2 Embedding Layer

Converte ciascun indice della sequenza in un vettore denso di dimensione 128. L'output di questo layer è un tensore di forma (None, 60, 128). Sui pesi di questo layer viene applicata una regolarizzazione L2, con penalità lieve di coefficiente **0.0005**

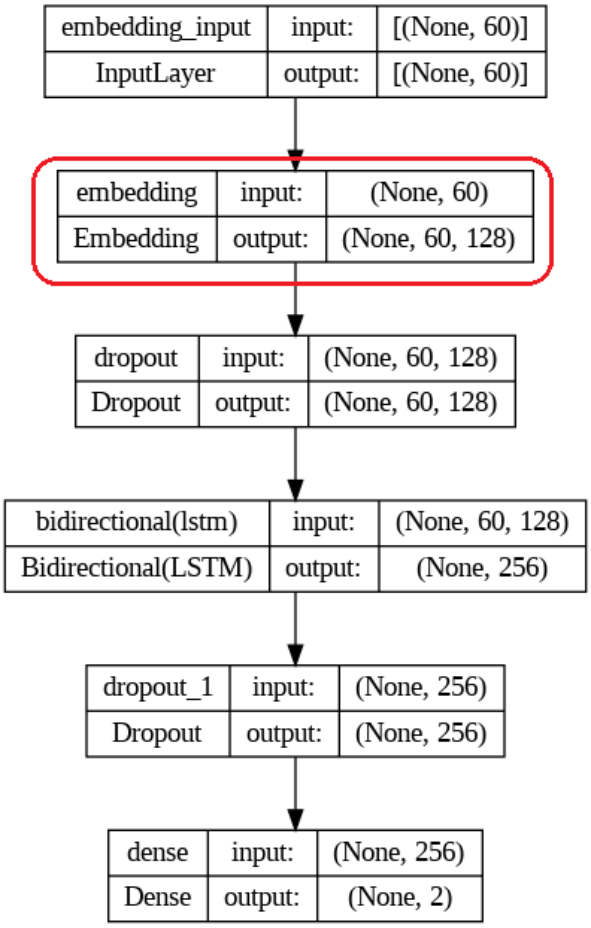

Figura 2.3: Embedding Layer.

2.1.3.3 Dropout Layer

Questi layers vengono utilizzati per ridurre il rischio di overfitting. In particolare, disattiva casualmente il 20 percento dei neuroni dopo il layer di embedding e il 50 percento dei neuroni dopo il layer Bi-LSTM (poiché il tasso di dropout è rispettivamente 0.2 e 0.5).

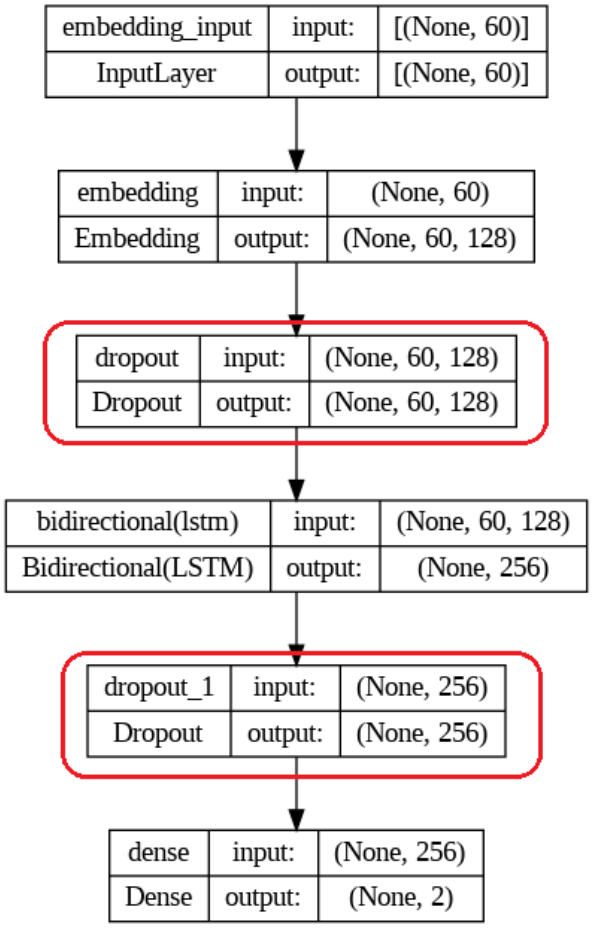

Figura 2.4: Dropout Layers.

2.1.3.4 Bidirectional LSTM Layer

Questo layer applica un LSTM bidirezionale, che permette al modello di catturare le informazioni non solo in direzione temporale (passato → futuro), ma anche nella direzione inversa (futuro → passato). Con 128 unità LSTM per ogni direzione (per un totale di 256 unità LSTM). Sui pesi di questo layer viene applicata una regolarizzazione L2, con penalità di coefficiente **0.005**

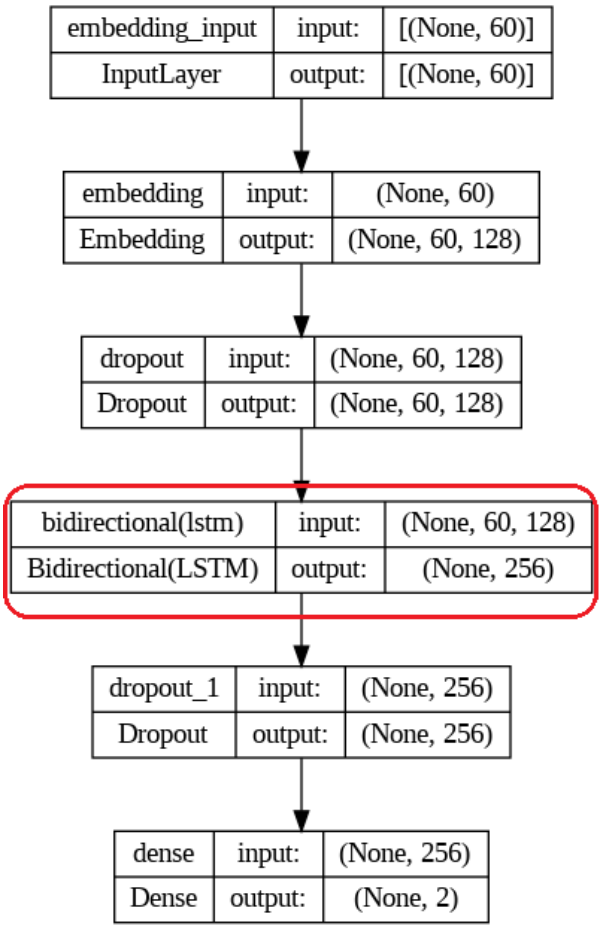

Figura 2.5: Bidirectional LSTM Layer.

2.1.3.5 Dense Layer

Questa è un layer completamente connesso (fully connected layer). Il modello produce due uscite, indicanti la probabilità per ciascuna delle due classi. La funzione di attivazione softmax è utilizzata per calcolare le probabilità di appartenenza alle due classi. L'output di questo layer è un tensore di forma (None, 2), che rappresenta la probabilità di appartenenza a ciascuna delle due classi. Sui pesi di questo layer viene applicata una regolarizzazione L2, con penalità di coefficiente **0.001**

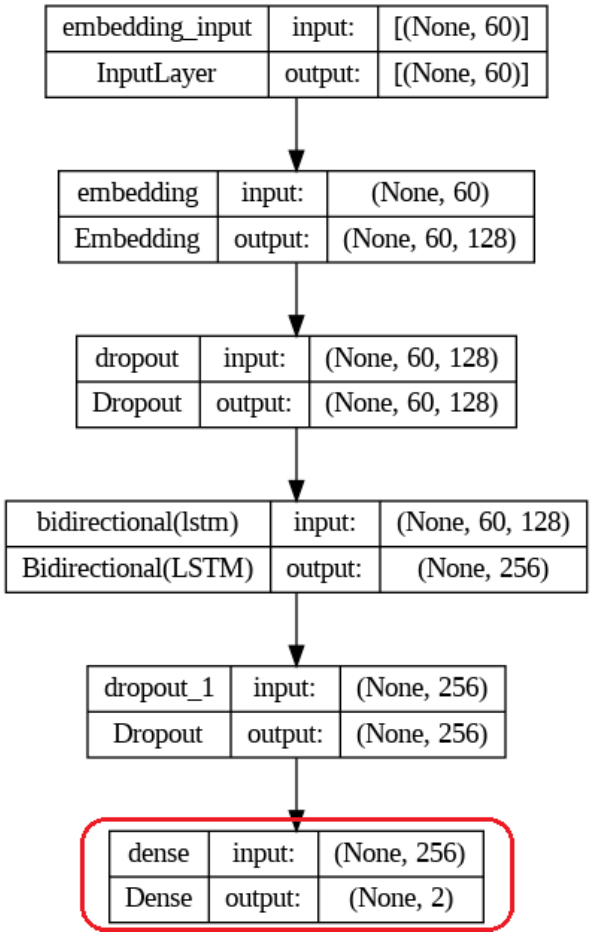

Figura 2.6: Dense Layer.

2.1.4 Train e Risutltati modello (RNN)

2.1.4.1 Configurazione del training

E' stata adotta la tecnica di Stratified k-fold cross validation per mantenere le proporzioni delle classi in ognuno dei 10 fold. Per ogni fold il modello è stato addestrato su 10 epoche, con un batch size di 32 per favorire la generalizzazione, introducendo "rumore" nel training, impedendo al modello di sovradattarsi ai dati. Sono stati utilizzati due metodi di callback per migliorare l'efficienza e la tracciabilità del modello. In particolare, il callback **ModelCheckpoint** è stato configurato per monitorare l'accuracy sul set di validazione e salvare automaticamente i pesi del modello corrispondenti alle migliori prestazioni raggiunte. Inoltre, il callback **TensorBoard** è stato utilizzato per registrare i log del processo di addestramento, consentendo una visualizzazione dettagliata delle metriche come la perdita e l'accuracy neldurante i fold, attraverso l'interfaccia grafica di TensorBoard.
Ottimizzatore: Adam con learning rate di 5×10^{-4}.
Funzione di perdita: Categorical cross entropy: $L = -\sum_{j=1}^{C} y_j \log(\hat{y}_j)$

2.1.4.2 Dati di training e validazione

Il dataset è stato suddiviso in tre parti, il test set che rappresenta il 10% del totale del dataset, ovvero 43'000 rows. Dopodiché la parte rimanente è stata a sua volta suddivisa in training set e validation set. Nello specifico, secondo la configurazione della Stratified k-fold cross validation, il validation set è costituito da 38'700 rows e il training set dalle restanti 348'300 rows.

2.1.4.3 Risultati

I risultati ottenuti dai grafici di validation accuracy Figura 2.7 e validation loss Figura 2.8, mostrano che il modello ha raggiunto performance abbastanza stabili e molto buone su tutti i fold della validazione incrociata. Ciò è attribuibile alla complessità moderata del task, poichè le SQLi seguono spesso pattern ripetitivi, come l'uso di parole chiave SQL (ad esempio, SELECT, UNION, DROP) o simboli specifici (', !, –, ;).

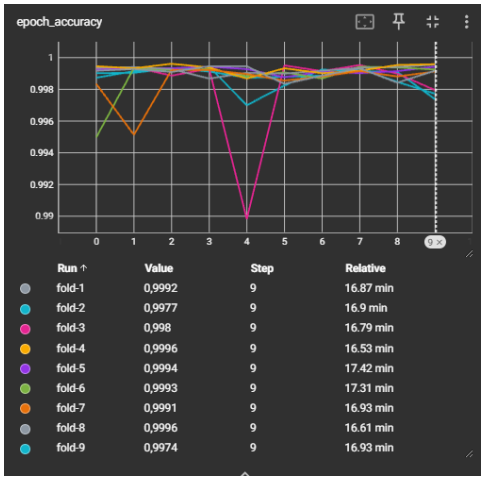

Figura 2.7: Epoch accuracy calcolata per ogni fold.

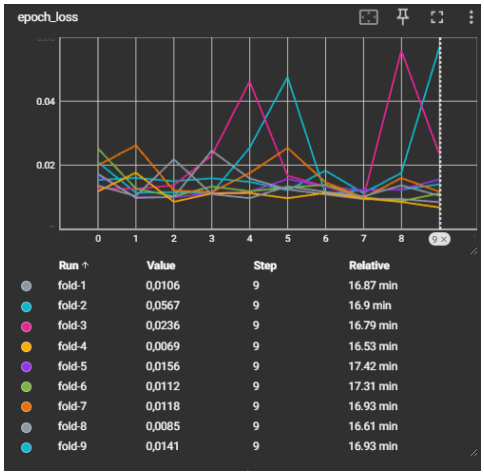

Figura 2.8: Epoch loss calcolata per ogni fold.

Le metriche utilizzate per valutare le prestazioni dei vari modelli sono: Accuracy, Precision, Recall ed F1-score:

$$
\begin{aligned}
precision &= \frac{TP}{TP + FP} \\
recall &= \frac{TP}{TP + FN} \\
F1 &= \frac{2 \times precision \times recall}{precision + recall} \\
accuracy &= \frac{TP + TN}{TP + FN + TN + FP}
\end{aligned}
$$

I risultati dei fold sono presenti nella Tabella 2.2

Tabella 2.2: Risultati modelli

Fold	Accuracy	Precision	recall	F1-Score
FOLD 1	0.99962	0.99948	0.99976	0.99962
FOLD 2	0.99923	0.99897	0.99948	0.99923
FOLD 3	0.99939	0.99907	0.99972	0.99939
FOLD 4	0.99944	0.99934	0.99953	0.99944
FOLD 5	0.99941	0.99925	0.99958	0.99941
FOLD 6	0.99944	0.99916	0.99972	0.99944
FOLD 7	0.99920	0.99916	0.99925	0.99920
FOLD 8	0.99944	0.99897	0.99990	0.99944
FOLD 9	0.99927	0.99920	0.99934	0.99927
FOLD 10	0.99948	0.99916	0.99981	0.99948

Per la scelta del modello più prestante ci siamo affidati alla metrica F1-score. Come si può notere dalla Figura 2.9, il modello con il risultato più alto è attribuibile al Fold 1. Nella stessa figura è riportata anche la matrice di confusione relativa al modello del Fold 1.

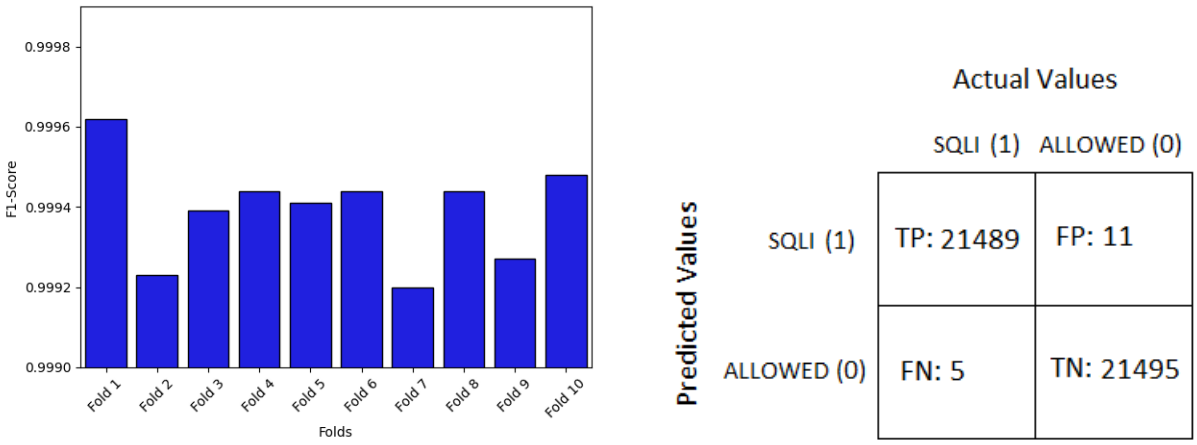

Figura 2.9: Sinistra: F1 score per epoch. Destra: Matrice di confusione Fold 1.

2.1.5 Creazione del modello (Finetune BERT)

Il secondo modello implementato per il rilevamento di SQLi è un fine-tune di **BERT** (**Bidirectional Encoder Representations from Transformers**), un modello di machine learning avanzato progettato da **Google** per la comprensione del linguaggio naturale. BERT è particolarmente potente grazie alla sua architettura bidirezionale che analizza il contesto di una parola in entrambe le direzioni, rendendolo ideale per il rilevamento di pattern come gli attacchi SQLi.

La scelta di utilizzare BERT nasce dalla voglia di sperimentare una soluzione alternativa e per avere un paragone di confronto con il modello custom (RNN). Per adattare BERT al task specifico, è stato effettuato un fine-tune utilizzando un dataset composto da 4'000 righe di dati ottenuto dalle stesse fonti precedenti. Il fine-tuning è stato limitato a tre epoche per un bilanciamento tra qualità del risultato e costi computazionali. In tali condizioni, un numero limitato di epoche riduce il rischio di overfitting che comprometterebbe la generalizzazione su dati nuovi mai visti.
Lo script scritto in python per il fine-tune di BERT si occupa di:

1. **Importazione delle librerie e definizione del dataset**: In questa fase, vengono importate le librerie necessarie, come **pandas** per la gestione dei dati e **transformers** per utilizzare il modello BERT. Il dataset viene caricato in un DataFrame e le etichette vengono mappate da testo a numeri (0 per "allowed" e 1 per "sqli").

```python
import torch
import pandas as pd
from transformers import BertForSequenceClassification,
    BertTokenizerFast
from torch.utils.data import Dataset
from transformers import TrainingArguments, Trainer
from sklearn.metrics import accuracy_score,
    precision_recall_fscore_support

# Caricamento del dataset
dataset_path = "bert_dataset_4000.csv"
dataframe = pd.read_csv(dataset_path)
labels = dataframe['Label'].unique().tolist()
labels = [s.strip() for s in labels]
n_lables = len(labels)

# Mappa dei label
id2label = {0: 'allowed', 1: 'sqli'}
label2id = {'allowed': 0, 'sqli': 1}
```

2. **Preprocessing dei dati**: Qui mappiamo i valori testuali delle etichette in numeri e prepariamo il tokenizer per la tokenizzazione del testo.

```
# Mappa del valore testuale di "Label" (allowed/sqli) a numerico (0/1)
dataframe["Label"] = dataframe.Label.map(lambda x: label2id[x.strip()])

# Tokenizer e modello base di BERT
tokenizer = BertTokenizerFast.from_pretrained("bert-base-uncased", max_lenght=512)
model = BertForSequenceClassification.from_pretrained("bert-base-uncased", num_labels=n_lables, id2label=id2label, label2id=label2id)
```

In questa sezione, le etichette vengono mappate ai rispettivi valori numerici. Inoltre, il modello BERT e il tokenizer vengono caricati con la configurazione predefinita per il modello `bert-base-uncased`.

3. **Suddivisione dei dati in Training, Validation e Test**: Il dataset viene suddiviso in modo che metà delle righe vengano usate per l'addestramento, un quarto per la validazione e un quarto per il test.

```
# Divisione dei dati in training, validazione e test
SIZE = dataframe.shape[0]
train_texts = [str(text) for text in dataframe.Testo[:SIZE//2]]
val_texts = [str(text) for text in dataframe.Testo[SIZE//2:(3*SIZE)//4]]
test_texts = [str(text) for text in dataframe.Testo[(3*SIZE)//4:]]

train_labels = list(dataframe.Label[:SIZE//2])
val_labels = list(dataframe.Label[SIZE//2:(3*SIZE)//4])
test_labels = list(dataframe.Label[(3*SIZE)//4:])
```

4. **Tokenizzazione dei dati**: I testi vengono tokenizzati utilizzando il metodo `tokenizer()`, che applica il padding e la troncatura necessaria per gestire testi di lunghezza variabile.

```
# Tokenizer applicato ai dati di input
train_encodings = tokenizer(train_texts, truncation=True,
    padding=True)
val_encodings = tokenizer(val_texts, truncation=True,
    padding=True)
test_encodings = tokenizer(test_texts, truncation=True,
    padding=True)
```

5. **Creazione della classe DataLoader per PyTorch**: Creiamo una classe `DataLoader` personalizzata per `PyTorch`, che permette di gestire in modo efficiente i dati tokenizzati.

```
# Classe Dataset personalizzata per PyTorch
class DataLoader(Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, index):
        item = {key: torch.tensor(value[index]) for key,
            value in self.encodings.items()}
        item['labels'] = torch.tensor(self.labels[index])
        return item

    def __len__(self):
        return len(self.labels)
```

Questa classe si occupa di incapsulare i dati tokenizzati e le etichette in un formato compatibile con PyTorch, facilitando l'iterazione durante l'addestramento.

6. **Impostazione del Trainer e training del modello**: Qui, il modello viene addestrato per 3 epoche. La funzione `compute_metrics` calcola le principali metriche di valutazione (`accuracy`, `precision`, `recall` e `F1`) durante il training.

```
# Creazione dei DataLoader per i dati
train_dataloader = DataLoader(train_encodings, train_labels)
val_dataloader = DataLoader(val_encodings, val_labels)
test_dataloader = DataLoader(test_encodings, test_labels)

# Funzione per calcolare le metriche
def compute_metrics(pred):
    labels = pred.label_ids
    preds = pred.predictions.argmax(-1)
    precision, recall, f1, _ =
       precision_recall_fscore_support(labels, preds, average
       ='macro')
    accuracy = accuracy_score(labels, preds)

    return {
        'Accuracy': accuracy,
        'F1': f1,
        'Precision': precision,
        'Recall': recall
    }

# Argomenti di training
training_args = TrainingArguments(
    output_dir = 'output/training_output',
    do_train = True,
    do_eval = True,
    num_train_epochs = 3,
    per_device_train_batch_size = 16,
    per_device_eval_batch_size = 32,
    warmup_steps = 100,
    weight_decay = 0.01,
    logging_strategy = 'steps',
    logging_dir = 'output/training_logging',
    logging_steps = 100,
    evaluation_strategy = "steps",
```

```
    eval_steps = 100,
    save_strategy = "steps",
    load_best_model_at_end = True
)

trainer = Trainer(
    model = model,
    args = training_args,
    train_dataset = train_dataloader,
    eval_dataset = val_dataloader,
    compute_metrics = compute_metrics
)

trainer.train()
```

7. **Valutazione del modello e salvataggio**: Infine, il modello viene valutato sui dati di addestramento, validazione e test, e il modello finale viene salvato.

```
# Valutazione sui dati di train, validation e test
evaluation_results = [
    trainer.evaluate(eval_dataset=train_dataloader),
    trainer.evaluate(eval_dataset=val_dataloader),
    trainer.evaluate(eval_dataset=test_dataloader)
]

# DataFrame contenente le metriche
evaluation_df = pd.DataFrame(evaluation_results, index=['train', 'validation', 'test'])
evaluation_df.to_csv("output/evaluation_metrics.csv", index=True)

# Salvataggio del modello finetunato e del tokenizer
model_path = "output/finetuned_model"
trainer.save_model(model_path)
tokenizer.save_pretrained(model_path)
```

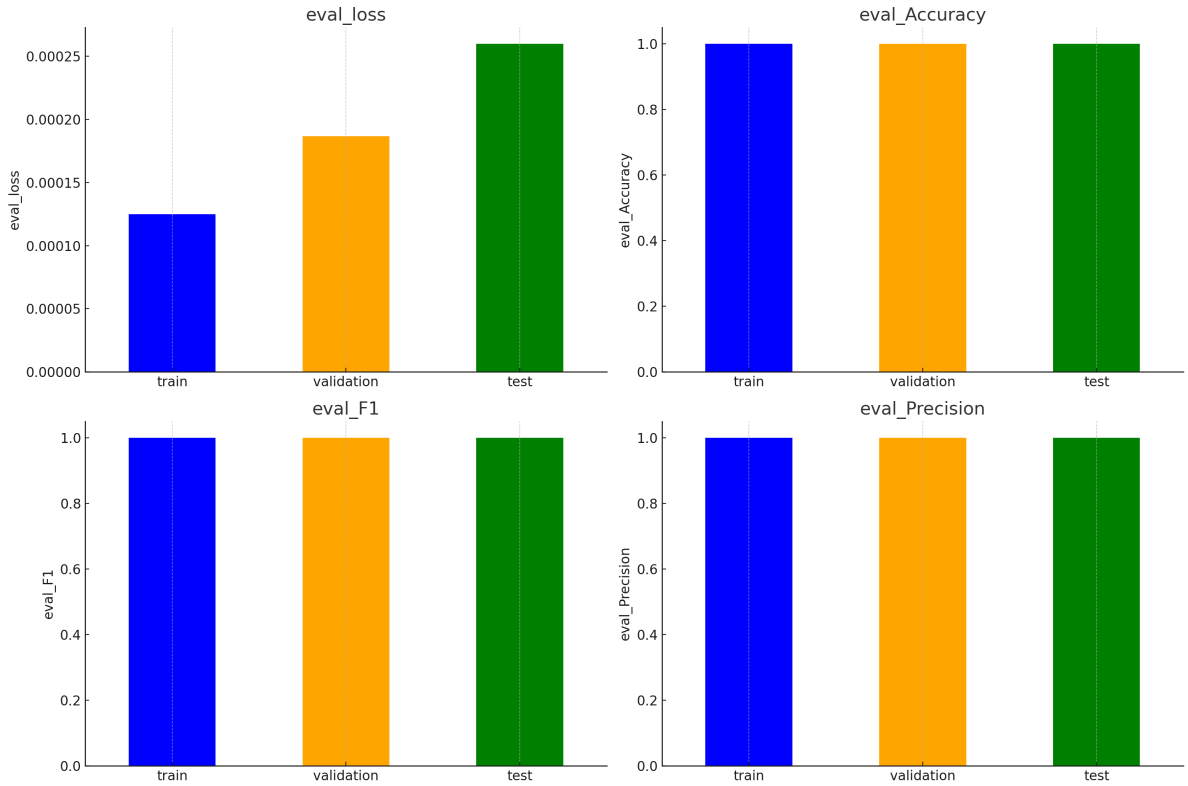

Figura 2.10: Visualizzazione risultati training.

2.1.6 Risultati modello (Finetune BERT)

Nella Figura 2.10 è mostrata la visualizzazione dei risultati ottenuti durante la fase di training, validazione e test del fine-tune di BERT.

1. **Loss**: Mostra un valore estremamente basso per tutte e tre le fasi, indicando che il modello ha raggiunto una convergenza ottimale.

2. **Accuracy**: Perfetta (1.0) in tutte le fasi, suggerendo che il modello è stato in grado di distinguere correttamente tra le classi `allowed` e `sqli`.

3. **Precision** e **F1 Score**: Entrambi perfetti (1.0), riflettendo un bilanciamento tra la capacità di evitare falsi positivi e identificare tutti i veri positivi.

I risultati ottenuti durante la fase di training, validazione e test mostrano metriche estremamente elevate, con una **loss** minima e valori perfetti (1.0) per **Accuracy**, **Precision**, **Recall**, e **F1 Score** in tutte le fasi. Questo era prevedibile considerando che il dataset utilizzato per il fine-tuning era relativamente piccolo, composto da solo 4'000 righe. Di queste, il 50% (2'000 righe) è stato utilizzato per la fase di training, mentre il restante 50% è stato suddiviso equamente tra validation e test (1'000 righe ciascuno). Con un campione di dati così limitato, risultati

così alti non sono inaspettati, poiché il modello ha avuto meno varietà di dati da processare e memorizzare.

Tuttavia, proprio per questo motivo, si è deciso di valutare le prestazioni del modello sul dataset utilizzato per lo sviluppo della RNN.

Questo dataset, molto più ampio, composto da 430'000 righe, ci ha permesso di ottenere un'analisi significativa e verificare la capacità del modello di generalizzare efficacemente a nuovi dati in scenari più realistici e complessi.

Lo script di Python che si occupa di questa analisi esegue le seguenti operazioni:

1. **Importazione delle librerie e setup iniziale**: Questa sezione configura il contesto di calcolo, sfruttando una GPU se presente, per migliorare le prestazioni durante il test.

```
from datetime import datetime
import pandas as pd
import torch
from transformers import BertForSequenceClassification,
   BertTokenizerFast
from sklearn.metrics import f1_score, precision_score,
   recall_score, accuracy_score, confusion_matrix

def append_item(item, file_path):
with open(file_path, "a") as file:
    file.write(f'{item}\n')

device = torch.device("cuda" if torch.cuda.is_available()
   else "cpu")
```

2. **Caricamento del modello e tokenizer**: Carichiamo il modello fine-tunato e il tokenizer corrispondente.

```
model_path = "sqli-detector-bert"
model = BertForSequenceClassification.from_pretrained(
   model_path).to(device)
tokenizer = BertTokenizerFast.from_pretrained(model_path)
```

3. **Preparazione dei dati**: Carichiamo il dataset e mappiamo le etichette testuali a valori numerici.

```
data_path = "dataset_430k.csv"
data = pd.read_csv(data_path)

label_mapping = {"allowed": 0, "sqli": 1}
data['Label'] = data['Label'].map(label_mapping)
```

4. **Funzione di predizione**: Definiamo una funzione per effettuare predizioni utilizzando il modello BERT.

```
def predict(text):
    inputs = tokenizer(text, padding=True, truncation=True,
       max_length=512, return_tensors='pt').to(device)
    outputs = model(**inputs)
    probs = outputs.logits.softmax(1)
    pred_label_idx = probs.argmax().item()
    return pred_label_idx
```

5. **Inizializzazione delle variabili**: Inizializziamo le variabili per il monitoraggio e creiamo file per memorizzare i risultati.

```
# Variabili per tracciare i risultati
true_labels = data['Label'].tolist()
texts = data['Testo'].tolist()
pred_labels = []

# File di output per classificazioni
false_positives_path = "false_positive.txt"
false_negatives_path = "false_negative.txt"
true_positives_path = "true_positive.txt"
true_negatives_path = "true_negative.txt"

# Contatori e setup iniziale
counter_fp = 0
counter_fn = 0
counter_tp = 0
counter_tn = 0

```

```
total_rows = len(texts)
start_time = datetime.now()
current_row = 0
```

6. **Test e registrazione dei risultati**: Iteriamo su ciascun dato nel dataset, effettuando predizioni e classificando i risultati in veri/falsi positivi e negativi.

```
# Eseguire predizioni e registrare risultati
for text, true_label in zip(texts, true_labels):
    current_row += 1
    print(f'Tested {current_row}/{total_rows} rows', end='\r')

    try:
        pred_label = predict(str(text))
        pred_labels.append(pred_label)

        if pred_label == true_label:
            # True Positives e True Negatives
            if pred_label == 1:
                append_item(text, true_positives_path)
                counter_tp += 1
            else:
                append_item(text, true_negatives_path)
                counter_tn += 1
        else:
            # False Positives e False Negatives
            if pred_label == 1:
                append_item(text, false_positives_path)
                counter_fp += 1
            else:
                append_item(text, false_negatives_path)
                counter_fn += 1
    except Exception as e:
        print(f'Broken @ row {current_row}: {str(text)}')

end_time = datetime.now()
```

7. **Calcolo delle metriche**: Calcoliamo le metriche di valutazione del modello, come accuracy, precision, recall e F1 score.

```
# Calcolo delle metriche
accuracy = accuracy_score(true_labels, pred_labels)
precision = precision_score(true_labels, pred_labels,
    average='binary')
recall = recall_score(true_labels, pred_labels, average='
    binary')
f1 = f1_score(true_labels, pred_labels, average='binary')
conf_matrix = confusion_matrix(true_labels, pred_labels)

# Output dei risultati
output = f'''
Accuracy: {accuracy}
Precision: {precision}
Recall: {recall}
F1 Score: {f1}
N True Positive: {counter_tp}
N True Negative: {counter_tn}
N False Positive: {counter_fp}
N False Negative: {counter_fn}
Confusion Matrix:
{conf_matrix}
'''
print(output)

with open("statistics.txt", "w") as file:
    file.write(output)
```

I risultati dimostrano un'elevata accuratezza con metriche eccellenti:

- **Accuracy**: 0.99927
- **Precision**: 0.99857
- **Recall**: 0.99997
- **F1 Score**: 0.99927

La **matrice di confusione** evidenzia **214'994 true positive** e **214'693 true negative**, con solo **307 false positive** e **6 false negative**, confermando una performance quasi perfetta nel distinguere tra parametri malevoli e legittimi. La scelta di valutare il modello su un dataset esteso è stata motivata dalla necessità di testarne la capacità di generalizzazione su un'ampia varietà di dati, rappresentativi di scenari reali.

I falsi negativi individuati sono trascurabili in quanto rappresentano un numero estremamente ridotto. Sebbene siano interessanti da analizzare, il loro impatto complessivo sul modello è minimo, e potrebbero non richiedere interventi significativi.

I falsi positivi, invece, includono principalmente stringhe lunghe in linguaggi come **GraphQL** o **JSON**. Queste stringhe, che seguono schemi tecnici strutturati, sono state erroneamente classificate come malevole, evidenziando che il modello potrebbe necessitare un'inclusione più ampia di esempi di questo tipo nel dataset di training.

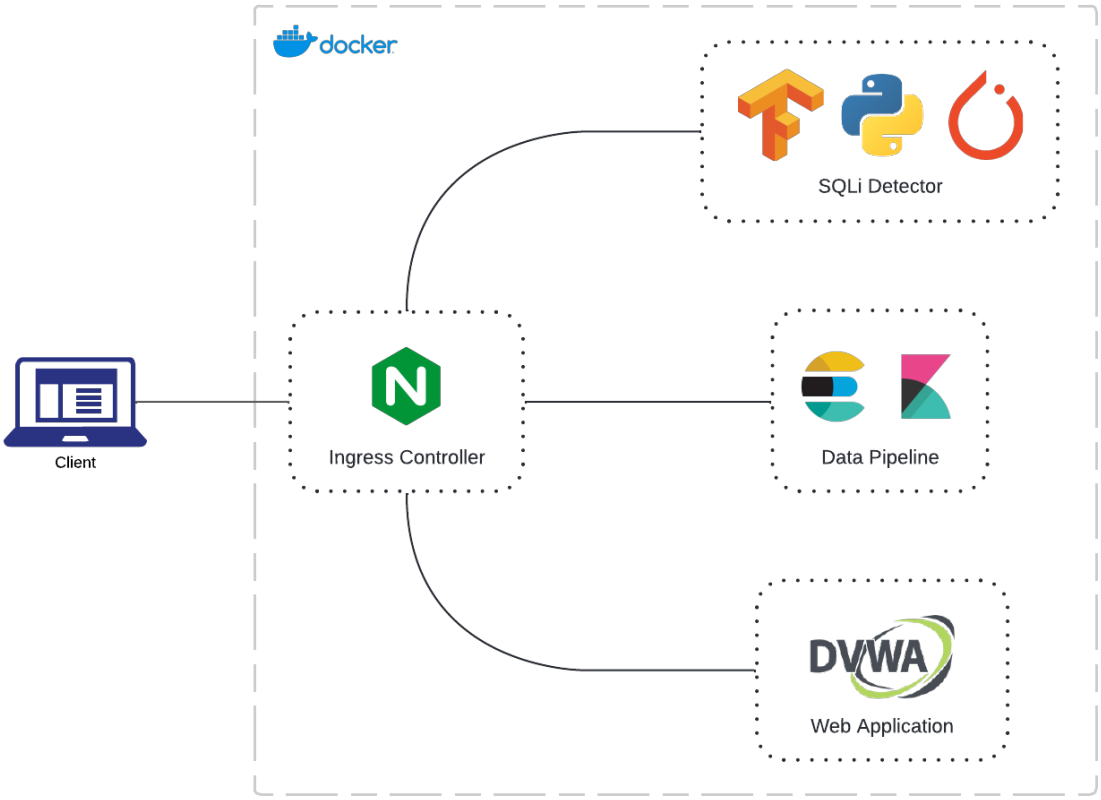

Figura 2.11: Diagramma architettura.

2.2 Model Deployment

La nostra soluzione è stata progettata con un'architettura modulare e scalabile, composta da componenti standard e ampiamente utilizzati nell'industria. L'obiettivo principale è garantire una gestione efficiente delle richieste, un'analisi accurata dei parametri e una visibilità completa sui dati elaborati.

Il client si interfaccia con un server **NGINX**, configurato come **Ingress Controller**, che si occupa di gestire il traffico in ingresso. Questo componente funge da barriera protettiva tra il client e l'applicazione web, assicurando che le richieste siano analizzate e filtrate prima di raggiungere il server dell'applicazione.

La soluzione include una **data pipeline** basata su **Elasticsearch** e **Kibana**: Elasticsearch memorizza tutti i dati delle richieste, mentre Kibana fornisce strumenti avanzati di visualizzazione, consentendo il monitoraggio in tempo reale attraverso dashboard e grafici personalizzati.

L'architettura comprende anche l'applicazione web stessa, che nel nostro caso è la **Damn Vulnerable Web Application (DVWA)**, utilizzata come ambiente di test per simulare applicazioni vulnerabili. A supporto di questa struttura, il componente fondamentale è l'**SQLi**

Detector, un endpoint che utilizza modelli di intelligenza artificiale per analizzare i parametri delle richieste, identificando eventuali tentativi di SQL injection.

Tutti i componenti della soluzione sono stati containerizzati utilizzando **Docker** come ambiente di virtualizzazione. Questa scelta garantisce la portabilità dell'intero sistema, che può essere facilmente distribuito e scalato su qualsiasi piattaforma compatibile con Docker, sia in ambienti locali che su infrastrutture cloud. Docker semplifica anche il processo di setup, riducendo al minimo le dipendenze specifiche dell'ambiente e aumentando la replicabilità della soluzione.

Questa architettura si basa su tecnologie consolidate e standard nell'industria, selezionate per la loro affidabilità, flessibilità e capacità di integrarsi senza difficoltà in ambienti esistenti, rendendo la soluzione versatile e facilmente adattabile a diverse esigenze operative.

2.2.1 Docker

Docker [3] è una piattaforma open-source che consente di creare, distribuire e gestire applicazioni all'interno di container leggeri e portabili. Un container è un'unità standard di software che contiene tutto ciò che è necessario per eseguire un'applicazione, inclusi codice, runtime, librerie e dipendenze, garantendo che questa funzioni in modo coerente indipendentemente dall'ambiente in cui viene eseguita. La scelta di Docker per il progetto è stata motivata da diversi fattori.

La **portabilità** è uno dei principali vantaggi: i container Docker possono essere eseguiti senza modifiche su qualsiasi sistema operativo che supporti Docker, dal laptop di sviluppo a un cluster di produzione su cloud. Questo garantisce che l'applicazione sia facilmente trasferibile tra ambienti senza preoccuparsi delle differenze nelle configurazioni del sistema.

Un altro punto a favore è la **facilità di messa in uso**. Docker consente di creare immagini preconfigurate che includono tutte le dipendenze richieste. Questo semplifica enormemente l'installazione e la configurazione dell'ambiente di esecuzione per chiunque voglia utilizzare o contribuire al progetto, riducendo drasticamente il rischio di errori legati alla configurazione.

Dal punto di vista della **scalabilità**, Docker facilita l'orchestrazione e il bilanciamento del carico quando si lavora con applicazioni complesse. Piattaforme di orchestrazione come Kubernetes possono gestire facilmente l'esecuzione di più container, garantendo che l'applicazione possa scalare orizzontalmente per gestire aumenti di traffico o utilizzo.

Infine, Docker offre **efficienza e isolamento**. Poiché i container condividono il kernel del sistema operativo, essi sono molto più leggeri rispetto alle tradizionali macchine virtuali, consumano meno risorse e si avviano rapidamente. Questo isolamento consente anche di mantenere un elevato livello di sicurezza, poiché ogni container opera in un ambiente autonomo, riducendo il rischio di interferenze tra componenti.

Queste caratteristiche rendono Docker la scelta ideale per il progetto, garantendo un'implementazione affidabile, scalabile e facilmente accessibile.

Per la gestione dei container, non è stata utilizzata l'interfaccia grafica fornita da Docker Desktop. Al suo posto, è stato scelto **Portainer** [10], una soluzione containerizzata che offre una comoda interfaccia grafica per la gestione dei container Docker. Portainer permette di monitorare e gestire i container direttamente attraverso un browser, semplificando operazioni come il monitoraggio dello stato dei servizi, la distribuzione di nuovi container e la gestione delle reti.

2.2.2 NGINX - Ingress Controller

Per gestire il traffico in ingresso, il sistema utilizza un server **NGINX**, configurato come **Ingress Controller**, che rappresenta un componente essenziale per garantire alte prestazioni e sicurezza. NGINX è uno dei web server più utilizzati al mondo, conosciuto per la sua capacità di gestire un elevato numero di connessioni concorrenti grazie alla sua architettura asincrona basata su eventi. Questo lo rende ideale per applicazioni web moderne, poiché funge sia da server web che da reverse proxy, bilanciatore di carico e firewall applicativo.

Nel progetto, NGINX non si limita a gestire il traffico, ma svolge un ruolo attivo nella protezione delle applicazioni. Agendo come una barriera protettiva davanti all'applicazione web, riduce il rischio di sfruttamento delle vulnerabilità già esistenti, senza richiedere modifiche al codice dell'applicazione. Questo approccio è particolarmente vantaggioso per applicazioni legacy, dove il codice potrebbe non essere facilmente modificabile.

La scelta di **OpenResty** [7], una versione estesa di NGINX, è stata motivata dalla sua capacità di integrare nativamente il linguaggio **Lua**, consentendo l'esecuzione di script direttamente all'interno del web server senza configurazioni aggiuntive. OpenResty permette così di aggiungere logiche personalizzate per l'analisi delle richieste, estendendo le funzionalità standard di NGINX in maniera efficiente.

Nel progetto, OpenResty utilizza uno script Lua per gestire diverse funzionalità chiave:

- **Estrazione dei parametri e informazioni dalle richieste:** ogni richiesta HTTP viene analizzata per estrarre i dati necessari.
- **Interazione con il modulo "SQLi Detector":** lo script invia i parametri delle richieste al modello AI per determinare se contengono payload malevoli.
- **Registrazione su Elasticsearch:** le richieste vengono salvate su Elasticsearch per il monitoraggio.
- **Decisioni sul trattamento della richiesta:** in base al verdetto del modello, il sistema può consentire il passaggio della richiesta all'applicazione web, bloccarla, ritardarla, o mostrare una risposta statica. Queste azioni sono completamente configurabili tramite il file di configurazione di NGINX.

Nello specifico:

1. **Configurazione iniziale**: In questa sezione vengono configurati gli endpoint di Elasticsearch e del servizio di controllo, con le relative credenziali. Si inizializzano inoltre le librerie JSON e HTTP per gestire le richieste.

```
local API_ENDPOINT = os.getenv("API_ENDPOINT")

-- Elasticsearch endpoint and credentials
local elastic_url = os.getenv("ELASTIC_ENDPOINT")
local username = os.getenv("ELASTIC_USERNAME")
local password = os.getenv("ELASTIC_PASSWORD")

-- Auth token for elastic
local credentials = username .. ":" .. password
local auth_header_value = "Basic " .. ngx.encode_base64(
   credentials)

-- JSON and HTTP libs
local cjson = require "cjson"
local http = require "resty.http"

-- HTTP client for elastic connection
local httpc_elastic = http.new()

-- HTTP client for checker request
local httpc = http.new()
httpc:set_timeout(1000)
```

2. **Estrazione dei parametri dalla richiesta**: I parametri GET e POST della richiesta vengono estratti per essere analizzati.

```
-- Extract GET and POST params from request
local args = ngx.req.get_uri_args()
ngx.req.read_body()
local post_args = ngx.req.get_post_args()
```

3. **Funzione per il controllo dei parametri**: Questa funzione invia i parametri a un endpoint esterno (SQLi Detector) per determinare se la richiesta contiene una SQLi.

```
local function check_param(param_value)
    local body_data = {
        p = param_value
    }
    local json_body = require("cjson").encode(body_data)
    local res, err = httpc:request_uri(API_ENDPOINT, {
        method = "POST",
        body = json_body,
        headers = {
            ["Content-Type"] = "application/json",
        },
    })
    if not res then
        ngx.log(ngx.ERR, "Request failed: ", err)
        return nil
    end
    return cjson.decode(res.body)
end
```

4. **Funzione per salvare dati su Elasticsearch**:

```
local function save_data_to_elastic(json_data)
    local res_elastic, err_elastic = httpc_elastic:
       request_uri(elastic_url, {
        method = "POST",
        body = json_data,
        headers = {
            ["Content-Type"] = "application/json",
            ["Authorization"] = auth_header_value
        }
    })

    if not res_elastic then
        ngx.say("Failed to send request to Elasticsearch: ",
            err_elastic)
    end
end
```

5. **Analisi dei parametri GET e POST**: Tutti i parametri GET e POST vengono controllati e classificati come malevoli o sicuri.

```
-- Check GET params
for k, v in pairs(args) do
    local response = check_param(v)
    if response then
        results[k] = { value = v, verdict = response.verdict
           , malicious = response.malicious }
    else
        results[k] = { value = v, verdict = "Not checked,
           checker is unreachable.", malicious = false }
    end
end

-- Check POST params
for k, v in pairs(post_args) do
    local response = check_param(v)
    if response then
        results[k] = { value = v, verdict = response.verdict
           , malicious = response.malicious }
    else
        results[k] = { value = v, verdict = "Not checked,
           checker is unreachable.", malicious = false }
    end
end

-- Set flag to true if malicious request
local has_malicious = false
for _, result in pairs(results) do
    if result.malicious then
        has_malicious = true
        break
    end
end
```

6. **Preparazione del log dei dati**: Viene creato un dizionario contenente tutte le informazioni sulla richiesta per il salvataggio e il monitoraggio.

```
-- Extract request's cookies
local cookies_string = ngx.var.http_cookie or ""
local cookies = {}

-- Split the cookie string into key-value pairs
for key, value in string.gmatch(cookies_string, "([^;=]+)=([^;]*)") do
    cookies[key:gsub("^%s*(.-)%s*$", "%1")] = value:gsub("^%s*(.-)%s*$", "%1") -- Trim whitespace
end

local log_data = {
    id = ngx.var.request_id,  -- Unique request ID
    timestamp = ngx.localtime(),  -- ISO 8601 format
    client_ip = ngx.var.remote_addr,  -- Client's IP address
    uri = ngx.var.request_uri,  -- Request URI
    url = ngx.var.scheme .. "://" .. ngx.var.host .. ngx.var.request_uri,  -- Full URL
    method = ngx.req.get_method(),  -- HTTP method
    http_version = ngx.req.http_version(),  -- HTTP version
    headers = ngx.req.get_headers(),
    query_params = args,  -- GET params
    post_params = post_args,  -- POST params
    is_malicious = has_malicious,  -- Detection result
    results = results
    ...
}
```

7. **Serializzazione e salvataggio**: Il dizionario viene serializzato in JSON e inviato a Elasticsearch. Infine, la richiesta viene reindirizzata in base al risultato dell'analisi.

```
local key_order = { "id", "timestamp", "client_ip", "uri", "
   url", "method", "http_version", "scheme", "server_host", "
   server_port", "user_agent", "referer", "content_length", "
   accept_language", "body", "cookies", "query_params", "
   post_params", "headers", "is_malicious", "results" }

-- Serialize JSON following EXPLICIT key order
local function ordered_json_encode(data, keys)
    local json_parts = {}
    table.insert(json_parts, "{")
    for i, key in ipairs(keys) do
        local value = cjson.encode(data[key])
        table.insert(json_parts, '"' .. key .. '":' .. value
           )
        if i < #keys then
            table.insert(json_parts, ",")
        end
    end
    table.insert(json_parts, "}")
    return table.concat(json_parts)
end

-- Log request's data to Elasticsearch
local json_data = ordered_json_encode(log_data, key_order)
save_data_to_elastic(json_data)
```

8. **Risposa alla richiesta**: Se una richiesta viene classificata come malevola, lo script lua ci permette di poterla gestire in cinque modi diversi.

```
if has_malicious then
    ngx.exec("@block");
    -- ngx.exec("@monitor");
    -- ngx.exec("@static");
    -- ngx.exec("@redirect");
    -- ngx.exec("@timeout");
else
    ngx.exec("@proxy");
end
```

   - @block: Risponde con il codice 403 e un messaggio predefinito.
   - @monitor: Permette alla richiesta di passare comunque.
   - @static: Mostra una pagina html statica.
   - @redirect: Reindirizza ad un URL esterno.
   - @timeout: Pausa la richiesta in modo da farla scadere senza risposta.

2.2.3 SQLi Detector

Il componente **SQLi Detector** si occupa del rilevamento delle SQL injection. Si tratta di un'applicazione scritta in Flask che espone un endpoint `/api/v1/check`, progettato per ricevere richieste **POST** contenenti il parametro da analizzare. Come mostrato nel codice fornito, l'endpoint utilizza una funzione Python per estrarre il parametro dalla richiesta, processarlo attraverso un modello di intelligenza artificiale, e restituire un verdetto sotto forma di JSON:

```
model = init_model(DEFAULT_MODEL_NAME)

@app.route('/api/v1/check', methods=['POST'])
def check_param():
    try:
        data = request.json
        param = data['p']
        return jsonify(model.get_verdict(param))
    except Exception as e:
        ...
```

Il cuore dell'applicazione risiede nel **manager del modello**, che è responsabile di caricare dinamicamente il modello di AI e il relativo tokenizer basandosi sul nome specificato nella configurazione (`DEFAULT_MODEL_NAME`).

Il manager supporta attualmente due modelli, `sqli-detector-claudio` e `sqli-detector-bert`. Il codice per il caricamento dei modelli si adatta al formato e alle specifiche di ciascun modello:

- Per il modello `sqli-detector-claudio`, il manager carica un file `.h5` e un tokenizer serializzato in formato pickle.
- Per il modello `sqli-detector-bert`, il caricamento avviene attraverso la libreria **Transformers**, usando il `BertForSequenceClassification` e il `BertTokenizerFast`.

```
MODELS = {
    0: "sqli-detector-claudio",
    1: "sqli-detector-bert"
}

from models.ModelClaudio import ModelClaudio
from models.ModelBERT import ModelBERT

def init_model(model_name):

```

```
    model = None

    if model_name == MODELS[0]:
        model = ModelClaudio()
    elif model_name == MODELS[1]:
        model = ModelBERT()

    if model is None:
        print("Model not found. Check model's name.")
        sys.exit()
    else:
        model.load_model_and_tokenizer()

    return model
```

Il sistema è progettato per flessibilità e modularità, permettendo l'aggiunta di nuovi modelli con minime modifiche al codice.

La funzione `get_verdict` ci permette di richiedere un responso al modello:
Per `sqli-detector-claudio`:

- Il parametro viene tokenizzato e trasformato in sequenze numeriche.
- Le sequenze vengono elaborate in input pre-processati tramite padding e reshaping.
- Il modello genera una previsione, identificando il parametro come `allowed` o `sqli`.

```
class ModelClaudio:

    def __init__(self):
        self.model = None
        self.tokenizer = None
        self.modelname = "sqli-detector-claudio"

    ...

    def get_verdict(self, text: str):
        # Parametri per il preprocessing (usati durante l'
           addestramento)
        MAP = {0: "allowed", 1: "sqli"}
        max_len = 60

        sql_keywords = r"\b(SELECT|INSERT|UPDATE|DELETE|DROP|
           TABLE|FROM|WHERE|AND|OR|UNION|LIKE|IS|NULL|BETWEEN|
           HAVING)\b"
        sql_symbols = r"[=\-+\*/<>;,\"'|()`!@#$%^&*[]"

        # Combinare tutti i pattern in uno (senza \s+ per gli
           spazi)
        combined_regex = f"({sql_symbols}|{sql_keywords}|[\w]+)"

        try:
            tokens = re.findall(combined_regex, text)
            tokens = [' '.join(token) if isinstance(token, tuple
               ) else token for token in tokens]
            preprocessed_texts = ' '.join(tokens)
            #print(preprocessed_texts)
```

```
26              sequences = self.tokenizer.texts_to_sequences([
                    preprocessed_texts])
27              padded_sequences = pad_sequences(sequences, maxlen=
                    max_len)
28              padded_sequences = np.array(padded_sequences, dtype=
                    np.int32)
29              prediction = self.model.predict(padded_sequences)
30              predicted_label = np.argmax(prediction, axis=1)
31              probability = prediction[0][predicted_label[0]]
32              verdict = MAP[predicted_label[0]]
33              #print(verdict, probability, predicted_label)
34
35              if verdict == "allowed":
36                  malicious =  False
37              else:
38                  malicious = True
39
40              return {'verdict': str(verdict), 'malicious': str(
                    malicious)}
41          except Exception as e:
42              print("\033[91m {}\033[00m" .format(f'Crashed @ "
                    get_verdict_from_claudio": {e}'))
43              return {'verdict': 'None', 'malicious': 'Not checked
                    .'}
```

Per `sqli-detector-bert`:

- Una pipeline preconfigurata per il task di classificazione utilizza il modello e il tokenizer per analizzare il parametro.
- Il risultato, una probabilità etichettata come `allowed` o `sqli`, viene restituito.

```
class ModelBERT():
    def __init__(self):
        self.model = None
        self.tokenizer = None
        self.modelname = "sqli-detector-bert"

    ...

    def get_verdict(self, text: str):
        try:
            nlp = pipeline("text-classification", model=self.
               model, tokenizer=self.tokenizer)
            result = nlp(text)
            verdict = result[0]['label']

            if verdict == "allowed":
                malicious =  False
            else:
                malicious = True

            return {'verdict': str(verdict), 'malicious': str(
               malicious)}
        except Exception as e:
            print("\033[91m {}\033[00m" .format(f'Crashed @ "
               get_verdict_from_bert": {e}'))
            return {'verdict': 'None', 'malicious': 'Not checked
               .'}
```

La separazione tra la logica del modello e l'endpoint Flask garantisce che l'applicazione sia facilmente estendibile, mentre l'utilizzo di Flask la rende leggera e ottimale per operazioni RESTful. Il design permette inoltre una rapida integrazione con altre componenti del sistema, come il server NGINX. Questo approccio garantisce che il SQLi Detector sia altamente modulare, configurabile e in grado di adattarsi a scenari futuri senza stravolgere l'infrastruttura esistente.

2.2.4 Data Pipeline

La **data pipeline** del sistema è composta da due componenti principali: **Elasticsearch** [4] e **Kibana** [5], che lavorano in sinergia per memorizzare e analizzare i dati relativi alle richieste HTTP monitorate. Elasticsearch è un motore di ricerca e analisi distribuito e open-source, progettato per gestire grandi quantità di dati strutturati o semi-strutturati in tempo reale. I dati delle richieste, comprese le informazioni sui parametri inviati, i verdetti del modello AI e altri metadati, vengono memorizzati in Elasticsearch. Questa scelta è stata dettata dalla sua capacità di scalare orizzontalmente, gestire query complesse in tempi rapidi e fornire un'archiviazione efficiente per enormi volumi di dati.

Kibana, il front-end di visualizzazione di Elasticsearch, consente di monitorare lo stato del sistema in tempo reale. Grazie a grafici interattivi, dashboard personalizzabili e strumenti di analisi avanzati, Kibana offre un modo intuitivo per interpretare i dati delle richieste. È possibile osservare metriche come il numero di richieste bloccate, il tasso di rilevamento di SQLi ecc... Questa capacità di visualizzazione è cruciale per il monitoraggio continuo e per prendere decisioni informate sul comportamento del sistema.

La scelta di questi strumenti è motivata dalla loro ampia diffusione nell'industria e dalla loro comprovata affidabilità in ambienti produttivi. Inoltre, essendo open-source, è facilmente integrabile con gli altri componenti del progetto, riducendo i costi di adozione e garantendo un'elevata flessibilità.

2.2.5 Web Application

La **Damn Vulnerable Web Application (DVWA)** [1] è un'applicazione web deliberatamente progettata per essere vulnerabile, utilizzata principalmente per scopi didattici e di test nell'ambito della sicurezza informatica. Essa offre una piattaforma ideale per studiare, rilevare e sfruttare vulnerabilità comuni nelle applicazioni web, come le SQL injection, l'XSS (Cross-Site Scripting), e altre minacce. DVWA è open-source, il che significa che è accessibile gratuitamente e può essere facilmente personalizzata per esperimenti specifici.

La scelta di DVWA come applicazione di test nel nostro caso di studio è stata guidata da diversi fattori. Prima di tutto, la sua ampia diffusione e notorietà la rendono uno **standard de facto** per esperimenti e pubblicazioni accademiche nel campo della sicurezza informatica. Inoltre, DVWA è disponibile come container Docker, consentendo una facile installazione e configurazione in ambienti isolati, perfettamente compatibile con la nostra infrastruttura basata su Docker.

Un altro vantaggio chiave di questa configurazione è che, essendo DVWA containerizzata e avendo NGINX a gestire il traffico, essa può essere facilmente sostituita con una qualsiasi altra applicazione web, anche preesistente. Questa flessibilità rende il sistema particolarmente utile per testare e proteggere ambienti web già in uso, senza richiedere modifiche strutturali.

2.2.6 Configurazione predefinita dei servizi

Portainer

- Porta 9443: espone la web UI per Portainer. Le credenziali di default per l'accesso sono `admin`:`password0000`

NGINX - Ingress Controller

- Porta 80: espone il server HTTP ma effettua il redirect alla porta 443
- Porta 443: espone il server HTTPS

SQLi Detetctor

- Porta 8899: espone l'api endpoint per le richieste POST per il controllo di un parametro. Questa porta è ri-configurabile attraverso il file `config.py`

Data Pipeline

- Elasticsearch, porta 9200: espone l'endpoint per l'accesso a Elasticsearch. Le credenziali di default per l'accesso sono `elastic`:`password`
- Kibana, porta 5601: espone l'endpoint per l'accesso alla webui di Kibana. Le credenziali di default per l'accesso sono `elastic`:`password`

DVWA (Damn Vulnerable Web Application)

- Porta 9999: espone l'endpoint per l'accesso alla webui di DWVA Le credenziali di default per l'accesso sono `admin`:`password`

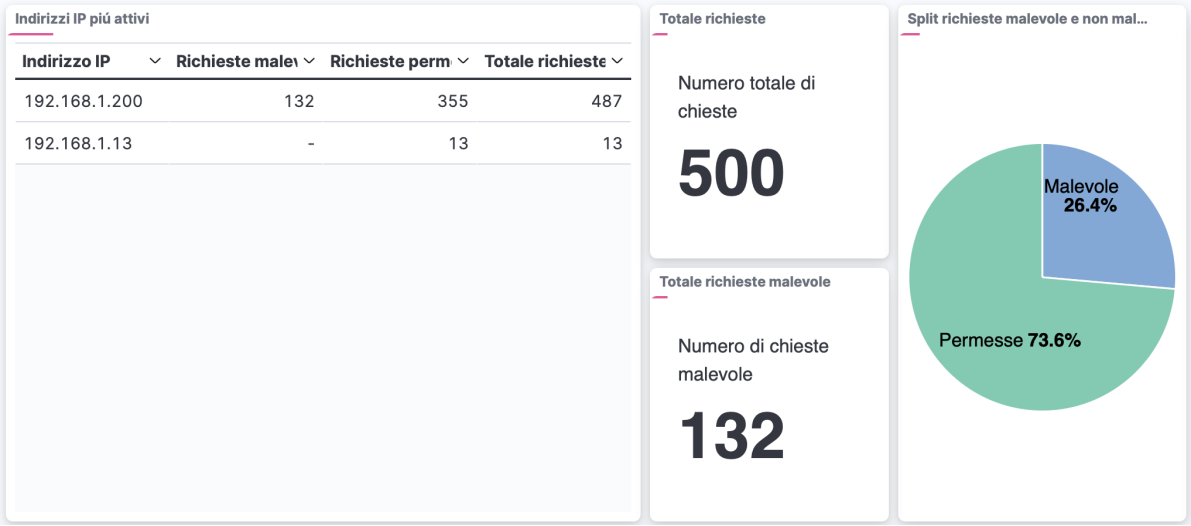

Figura 2.12: Informazioni di base

2.3 User Interface

L'interfaccia grafica del sistema è stata realizzata utilizzando Kibana. Attraverso dashboard interattive e personalizzabili, gli utenti possono monitorare in tempo reale il traffico di rete, identificare pattern sospetti e valutare l'efficacia delle misure di protezione adottate. Questo approccio garantisce una visione chiara e immediata delle attività, migliorando la capacità di risposta e la gestione complessiva della sicurezza.

La dashboard mostra le seguenti informazioni:

- **Informazioni di base sulle richieste in entrata** [Figure 2.12], come indirizi IP più attivi e suddivisione delle richieste.
- **User-Agent più utilizzati** [Figure 2.13].
- **Numero di richieste ricevute per intervallo temporale** [Figure 2.14].
- **Origine del traffico** [Figure 2.15], per monitorare le pagine con più interazioni.
- **Informazioni aggiuntive sulle richieste** [Figure 2.16], come i metodi HTTP più utilizzati, le lingue accettate e la dimensione delle richieste.
- **Frequenza dei parametri GET e POST** [Figure 2.17].

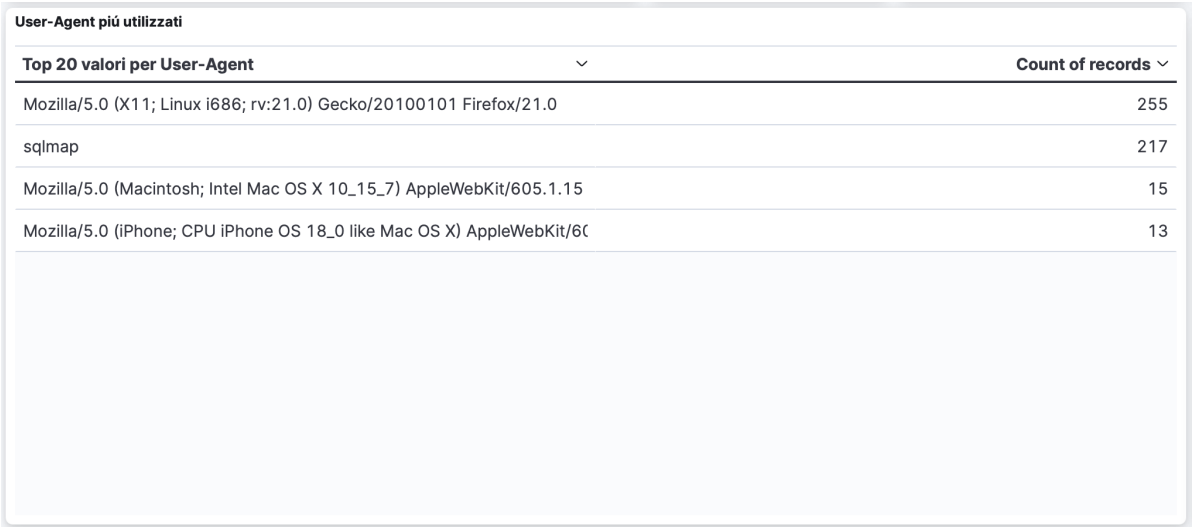

Figura 2.13: User-Agent più utilizzati

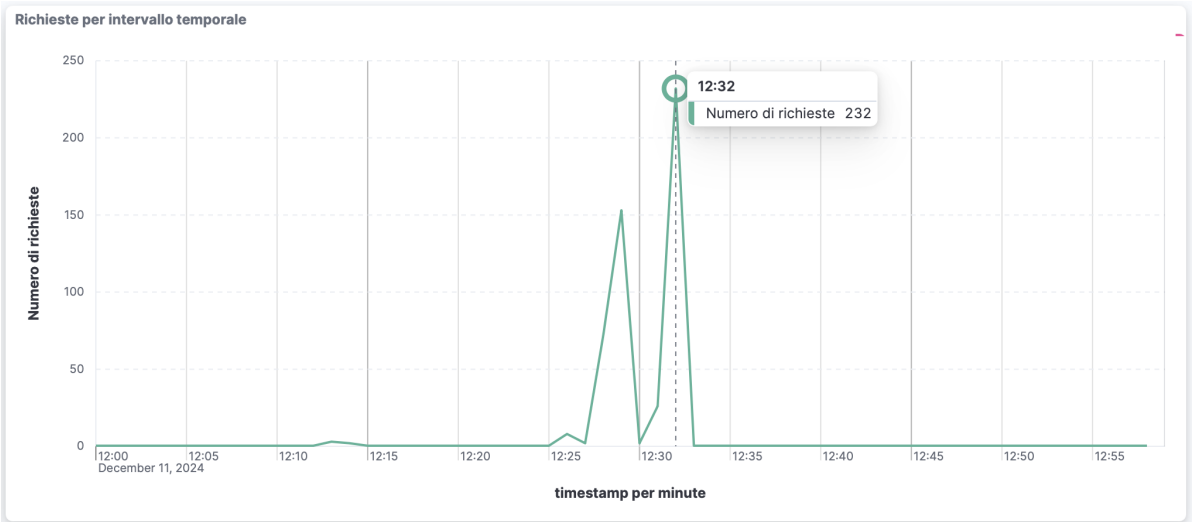

Figura 2.14: N° richieste per intervallo temporale

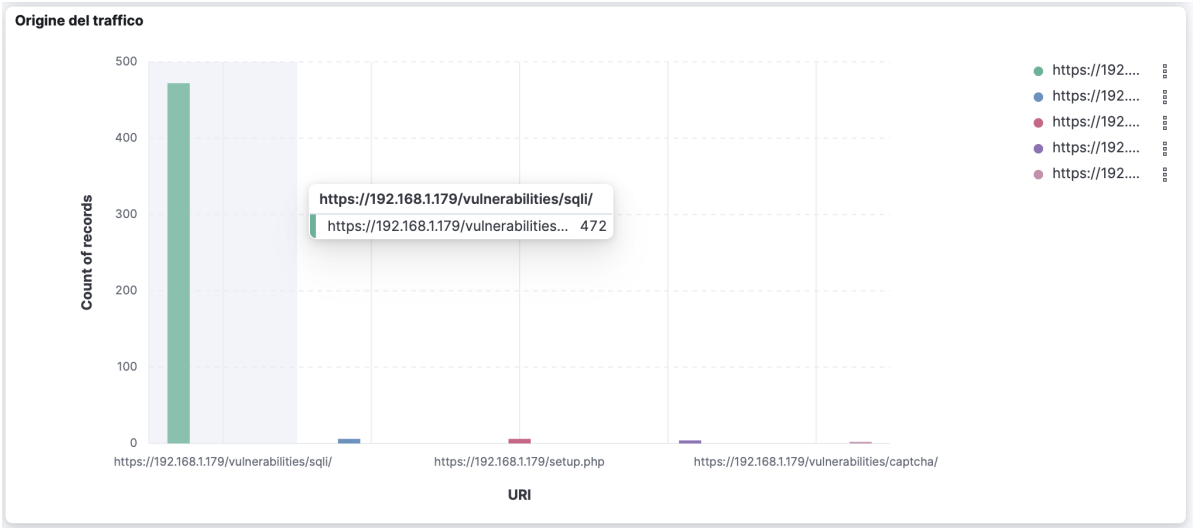

Figura 2.15: Origine del traffico

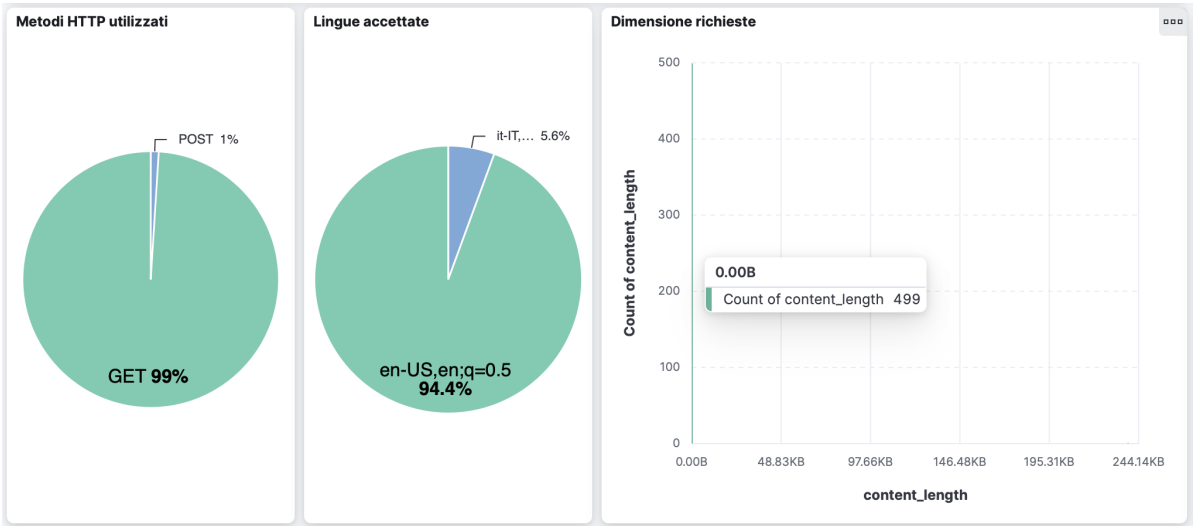

Figura 2.16: Informazioni aggiuntive sulle richieste

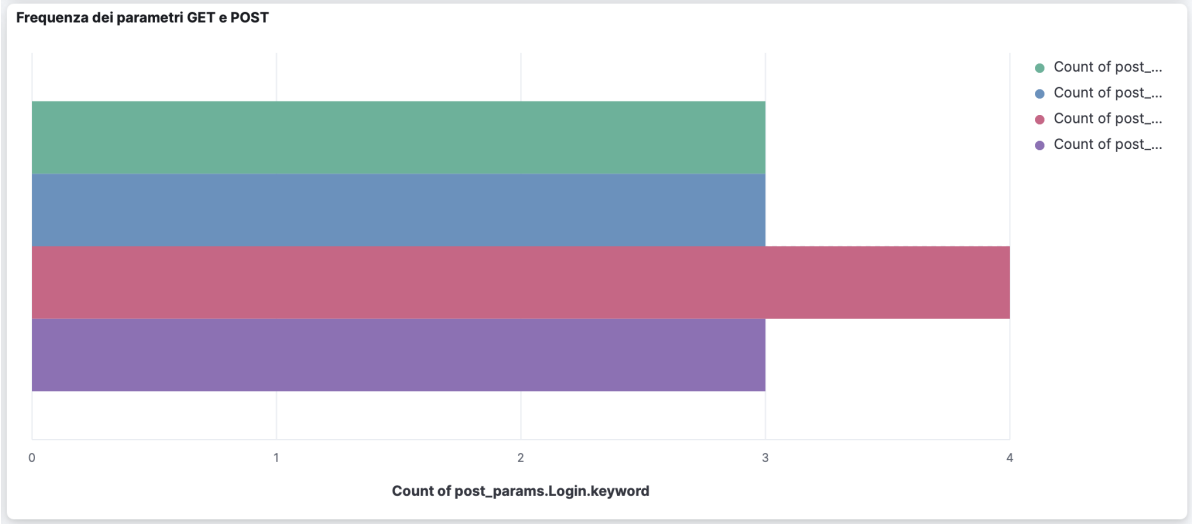

Figura 2.17: Frequenza dei parametri GET e POST

Capitolo 3

Conclusioni

Il sistema sviluppato rappresenta un passo significativo verso una protezione più efficace e adattabile contro le SQL Injection, sfruttando tecnologie avanzate di intelligenza artificiale e machine learning. Grazie alla sua architettura modulare e scalabile, supportata da strumenti standard come NGINX, Elasticsearch e Kibana, la soluzione offre un framework robusto e facilmente integrabile in ambienti esistenti, compresi i sistemi legacy. I risultati ottenuti dimostrano l'efficacia dei modelli nell'identificare e mitigare le minacce, riducendo al minimo gli errori e fornendo un monitoraggio dettagliato in tempo reale.

Questa esperienza ha permesso di comprendere come l'intelligenza artificiale possa essere un alleato prezioso nella sicurezza informatica, non solo per affrontare sfide complesse come le SQL Injection, ma anche per migliorare e supportare le tecnologie esistenti. L'IA non si pone come sostituto delle soluzioni tradizionali, ma come un elemento complementare capace di colmare lacune e potenziare i sistemi di difesa attualmente in uso.

Le possibilità di estensione futura, come l'integrazione di altri tipi di attacchi (ad esempio XSS o LFI), rendono il sistema una base solida per ulteriori sviluppi nel campo della sicurezza delle applicazioni web. Il lavoro svolto evidenzia come un approccio innovativo e guidato dall'intelligenza artificiale possa offrire una protezione più semplice e adattabile, ponendo le basi per soluzioni sempre più evolute in risposta alle crescenti minacce del panorama digitale.

Bibliografia

[1] Damn Vulnerable Web Application.
`https://github.com/digininja/DVWA`.

[2] Luca Demetrio, Andrea Valenza, Gabriele Costa, and Giovanni Lagorio.
Waf-a-mole: Evading web application firewalls through adversarial machine learning.
Proceedings of the 35th Annual ACM Symposium on Applied Computing, 2020.

[3] Docker.
`https://docs.docker.com/engine/install/`.

[4] ElasticSearch.
`https://www.elastic.co/elasticsearch`.

[5] Kibana.
`https://www.elastic.co/kibana`.

[6] mitmproxy.
`https://mitmproxy.org`.

[7] OpenResty.
`https://openresty.org/en/`.

[8] OWASP.
Preventing sql injections in online applications.
`https://wiki.owasp.org/images/5/57/OWASP-AppSecEU08-Janot.pdf`.

[9] OWASP.
Top 10 web application security risks 2021.
`https://owasp.org/www-project-top-ten/`.

[10] Portainer.
`https://www.portainer.io`.

[11] rockyou Wordlist.
`https://www.kaggle.com/datasets/wjburns/common-password-list-rockyoutxt`.

[12] Hilmi Salih Abdullah and Adnan Mohsin Abdulazeez.
Detection of sql injection attacks based on supervised machine learning algorithms: A review.
International Journal of Informatics, Information System and Computer Engineering (INJIISCOM), 5(2):152–165, Apr. 2024.

[13] SQLMap.
`https://sqlmap.org`.