# Progetto CodeStresser

CodeStresser prende semplicemente del codice e ti dice se c’è qualche vulnerabilità nota.

## Organizzazione del codice

Il codice verrà organizzato in moduli.

|  |  |
| --- | --- |
| Nome | Descrizione |
| Lettore del Codice | Carica uno o più file sorgente da analizzare. Supporta vari formati (es. .py, .c, .java) ed esegue operazioni preliminari come l’estrazione da archivi e la normalizzazione del testo. |
| Analizzatore Statico | Fa il parsing del codice in AST. Elabora l’AST per identificare pattern di vulnerabilità noti. Estrae feature semantiche rilevanti (es. chiamate pericolose, flussi di input non sanitizzati) e costruisce un vettore di caratteristiche per ciascun frammento di codice. |
| Modello di Machine Learning | Riceve i vettori di feature dall’analizzatore statico e predice la probabilità di vulnerabilità tramite classificatori (es. Random Forest, GNN, LSTM). Restituisce tipo di vulnerabilità, punteggio di confidenza e metadati associati. |
| Generatore del report | Genera il report con i punteggi effettivi di vulnerabilità. |

## Moduli

### Lettore del codice

Il lettore del codice consisterà in un semplice modulo che prenderà in input uno o più file di codice per inserirli nel programma (lista di file).

### Analizzatore Statico

L’Analizzatore Statico produce l’AST a partire dal codice ed è responsabile dell’identificazione di pattern di codice potenzialmente vulnerabili. Questo modulo non esegue il codice, ma applica tecniche di analisi statica per estrarre **feature strutturali e semantiche** rilevanti alla sicurezza.

#### FunzionalitàPrincipali

* **Estrazione di Pattern a Rischio**
  + Riconoscimento di chiamate a funzioni pericolose (eval, exec, system, strcpy)
  + Identificazione di SQL dinamici (es. concatenazione di stringhe con query)
  + Rilevamento di costrutti privi di validazione degli input
* **Taint Analysis (Analisi del flusso di dati)**
  + Tracciamento del flusso da *sorgenti* (input utente, parametri HTTP, ecc.) a *sink* sensibili (file I/O, database, shell)
  + Etichettatura delle variabili lungo il flusso semantico per identificare punti di possibile sfruttamento
* **Metriche di Sicurezza**
  + Conteggio di funzioni non sicure
  + Profondità dei rami di controllo (if, while, try)
  + Presenza o assenza di gestori di eccezioni o controlli di errore

#### Output del Modulo

L’analizzatore produce come risultato un **vettore di feature**, una rappresentazione numerica e/o simbolica del codice, pronta per il modello ML. Inoltre, può generare metadati utili come:

* Nome del file e della funzione
* Numero di righe analizzate
* Indicatori di rischio (es. “3 chiamate a eval()”)

### Modello di Machine Learning

Il modulo di Machine Learning riceve in input i **vettori di feature** generati dall’Analizzatore Statico e restituisce una previsione sul rischio di vulnerabilità del codice analizzato. Il modello apprende da esempi noti di codice vulnerabile per identificare pattern pericolosi anche in codice non visto.

#### Funzionalità Principali

* **Addestramento**
  + Il modello viene addestrato su dataset etichettati (vulnerabile / sicuro).
  + Esempi di dataset:
    - **Juliet Test Suite** (C/C++)
    - **BigVul** (vulnerabilità reali da GitHub)
    - **SARD** (Software Assurance Reference Dataset)
  + Possibilità di arricchimento con esempi sintetici (vulnerabilità artificialmente iniettate)
* **Predizione**
  + In fase di esecuzione, il modello riceve il vettore di feature e restituisce:
    - **Etichetta di vulnerabilità** (es. SQL Injection, Buffer Overflow, ecc.)
    - **Punteggio di confidenza** (es. 0.87 = alta probabilità)
    - **Classe di rischio** (basso, medio, alto, critico)
* **Supporto Multi-classe (opzionale)**
  + Estensione del modello per classificare diversi tipi di vulnerabilità, non solo “vulnerabile/sicuro”.

#### Tecnologie e Algoritmi

* **Algoritmi Tabulari (vettori di feature)**:
  + **Random Forest**, **XGBoost**, **Gradient Boosting**
  + Adatti a feature numeriche ed etichette classiche
  + Implementabili con **Scikit-learn**, **LightGBM**
* **Algoritmi Sequenziali (token stream)**:
  + **LSTM**, **Transformer**
  + Adatti all’analisi semantica delle sequenze di token nel codice
  + Implementabili con **PyTorch**, **TensorFlow**, **HuggingFace Transformers**
* **Graph Neural Networks (AST/CFG)**:
  + **GNN (Graph Neural Network)**, **GGNN**, **GAT**
  + Adatti per rappresentazioni strutturate come AST o CFG
  + Implementabili con **PyTorch Geometric**, **DGL**

#### Output del Modulo

Il modulo produce una serie di risultati utilizzabili dai moduli successivi:

* **Etichetta Predetta**: vulnerabilità rilevata (es. “SQL Injection”)
* **Score di Confidenza**: valore tra 0 e 1 che rappresenta la probabilità stimata
* **Classe di Severità**: (facoltativo) stimata sulla base della vulnerabilità rilevata
* **Metadati**: ID del frammento di codice, posizione, riferimenti al file originale

### Generatore del Report

Il Generatore del Report è il modulo finale del sistema: riceve le predizioni effettuate dal modello di Machine Learning e costruisce un report leggibile e strutturato per lo sviluppatore o il team di sicurezza. Il report evidenzia i problemi rilevati, ne indica la gravità, la posizione nel codice e propone eventuali azioni correttive.

#### Funzionalità Principali

* **Compilazione delle informazioni**
  + Raccoglie i risultati di analisi: tipo di vulnerabilità, punteggio di rischio, posizione (file/linea).
  + Includere snippet di codice per maggiore contesto.
  + Mappa i risultati a linee specifiche per facilitare la correzione.
* **Formattazione del report**
  + Report leggibile per esseri umani (sviluppatori)
  + Output strutturato per strumenti CI/CD
  + Evidenziazione della gravità e della priorità di ciascun problema
* **Suggerimenti di mitigazione**
  + Per ogni vulnerabilità può essere incluso un consiglio automatico (es. "Utilizzare query parametrizzate invece di concatenazioni").

#### Output Esempio

{

"file": "auth\_handler.py",

"line": 42,

"issue": "Possible SQL Injection",

"severity": "High",

"confidence": 0.91,

"suggested\_fix": "Use parameterized queries instead of string concatenation."

}

## Analizzatore Statico – I famosi Pattern a Rischio

| **Vulnerabilità** | **Descrizione** | **Rischio** | **Esempio di sfruttamento** |
| --- | --- | --- | --- |
| eval() / exec() | Esecuzione dinamica di codice | Remote Code Execution (RCE) | eval(input(">> ")) → inserisco os.system("rm -rf /") |
| os.system() / subprocess | Comandi shell costruiti con input utente | Command Injection | os.system("ping " + user) → & rm -rf / |
| SQL dinamico (+, %, f"") | Query costruita con variabili concatenate | SQL Injection | cursor.execute("SELECT \* FROM users WHERE user='" + u + "'") → ' OR '1'='1 |
| input() non validato | Input non controllato usato direttamente | Injection / Crash | x = int(input()) → crash se passo "ciao" |
| pickle.load() | Deserializzazione non sicura di input | Arbitrary Code Execution | file .pkl contenente codice malevolo → eseguito |
| strcpy() | Copia senza controllo della lunghezza | Buffer Overflow | strcpy(dest, input); → overflow del buffer dest |
| open("file", "w") dinamico | Path pericoloso costruito da input | File Overwrite / Traversal | open(user + ".txt", "w") → ../../../etc/passwd |
| Nessuna try/except | Nessuna protezione su operazioni critiche | Denial of Service | x = 1/0 → crash del programma |
| Profonda annidazione logica | Rami if, while annidati senza controllo | Codice fragile e rischioso | if x: if y: if z: if a: ... |
| Flusso input → exec() | Dati da utente usati in funzioni critiche | Tainted Flow → RCE | x = input(); exec(x) → \_\_import\_\_('os').system('ls') |
| request.GET["x"] diretto | Parametro HTTP usato senza sanificazione | Web Parameter Injection | os.system("ls " + request.GET["path"]) |
| Variabile non inizializzata | Uso di variabile prima di assegnarla | Crash a runtime | if x > 10: print(y) → NameError: y is not defined |

## Analizzatore Statico – Code Analysis

Quando viene creata un’istanza di StaticAnalyzer, vengono inizializzati:

* **self.vulnerabilities**: una lista che conterrà tutte le vulnerabilità trovate
* **self.defined\_vars**: insieme di variabili inizializzate (serve per trovare variabili usate prima di essere assegnate)
* **self.critical\_calls\_outside\_try**: dizionario che salva le funzioni critiche chiamate fuori da un blocco try
* **self.sources**: insieme di sorgenti da cui può arrivare input non affidabile (input, sys.argv, ecc.)
* **self.sinks**: funzioni pericolose, anche se non sempre causano eccezioni (es. open, exec)
* **self.critical\_sinks\_needing\_try**: sottoinsieme di sinks che richiede protezione con try/except
* **self.tainted\_vars**: insieme di variabili contaminate (valori non affidabili da sorgenti utente)
* **self.current\_function**: nome della funzione attualmente analizzata
* **self.control\_depth, self.max\_control\_depth**: usate per misurare la profondità delle strutture di controllo (if, while, ecc.)
* **self.in\_try\_block**: flag booleano che indica se siamo dentro un blocco try

### Analisi delle funzioni (visit\_functiondef)

* salva il nome della funzione in self.current\_function
* resetta il contesto per l’analisi
* visita tutti i nodi interni alla funzione
* alla fine, se la profondità delle strutture supera 3, segnala la vulnerabilità

### Blocco try (visit\_try)

* salva il valore attuale di in\_try\_block (può essere annidato)
* imposta in\_try\_block = true
* visita i nodi figli (cioè il contenuto del try)
* ripristina il valore precedente di in\_try\_block

### Strutture di controllo (visit\_if, visit\_while, visit\_for)

* incrementano la profondità
* aggiornano max\_control\_depth se necessario
* visitano i figli
* poi decrementano la profondità

### Analisi delle assegnazioni (visit\_assign)

* controlla i target (a sinistra dell’assegnazione). se sono nomi semplici, li aggiunge a defined\_vars
* controlla il valore assegnato:
  + se è una chiamata a una funzione (ast.call), e questa funzione è una source, allora i target diventano tainted
  + se è un accesso tipo request.GET[...], anche in questo caso i target vengono marcati tainted

### Analisi delle chiamate (visit\_call)

* ottiene il nome completo della funzione chiamata (func\_name)
* se la funzione è una critical\_sink e siamo *fuori* da un try, segnala vulnerabilità
* se è in sinks, segnala vulnerabilità comunque
* se è open e il file passato è una variabile tainted, segnala uso pericoloso
* se è pickle.load, segnala deserializzazione pericolosa
* se è strcpy, segnala potenziale buffer overflow
* se è cursor.execute e il primo argomento è una stringa costruita dinamicamente, segnala sql dinamico
* per ogni argomento:
  + se è una variabile tainted e il sink è pericoloso, segnala flusso di dati non sicuro

### Analisi dell’uso delle variabili (visit\_name)

* se una variabile viene usata in lettura (ast.load) ma non è né definita né tainted, viene segnalata come usata senza inizializzazione

### Costruzione del nome funzione (get\_full\_func\_name)

* se la funzione è semplice (eval) → ritorna il nome
* se è tipo os.system → costruisce il nome completo con .

### Funzione di analisi (analyze)

* esegue ast.parse() sul codice sorgente
* visita ogni nodo dell’albero
* ritorna la lista di vulnerabilità

### Generazione del feature vector (generate\_feature\_vector)

* prende tutte le vulnerabilità trovate
* le conta per tipo:
  + chiamate pericolose
  + query sql dinamiche
  + flussi tainted
  + mancanza di try
  + struttura di controllo troppo profonda
* restituisce un dizionario numerico riassuntivo