# 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. |
| Parser | Fai il parsing del codice in AST (Abstract Syntax Tree). |
| Analizzatore Statico | Elabora l’AST e il CFG 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).

### Parser

Utilizza l’output del lettore del codice per generare un Abstract Syntax Tree.  
Un **Abstract Syntax Tree (AST)** è una rappresentazione strutturata e gerarchica del **significato sintattico di un codice sorgente**, senza includere dettagli inutili della sintassi (come parentesi o punteggiatura).

A screenshot of a computer

AI-generated content may be incorrect.

Il modulo utilizzerà la libreria “ast” di python per la trasformazione in Abstract Syntax Tree.

A screenshot of a computer program

AI-generated content may be incorrect.

### Analizzatore Statico

L’Analizzatore Statico è responsabile dell’identificazione di pattern di codice potenzialmente vulnerabili a partire dall’AST. 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