i need to develop a WAAP in that iam developing a API security module1. First we will start with REST API later we can use other API types which i will mention later.

i need to protect against the OWASP top 10 API. so give me a api security module for a WAAP. i need another module2 which is a spider or crawler which crawls through all the API end points and save it and protect those end points against the attacks, as we know API endpoints across different web applications will be different , so my module code should be same and should work with any web application.

and module3 : is a AI/ML model, where i need to train the WAAP with all the current attacks vectors and then when i come across a different behavior , it should be blocked.

so give me complete working modules for all this purposes. except module3 which i will ask later.

Now I'll create comprehensive working Python modules for your WAAP system. I'll provide Module 1 (API Security with OWASP Top 10 protection) and Module 2 (API Endpoint Discovery/Spider).

Module 1: API Security Module for OWASP Top 10 Protection

This module provides comprehensive protection against the OWASP API Security Top 10 2023 vulnerabilities:[1] [2]

```
"""
WAAP - API Security Module
Protects REST APIs against OWASP Top 10 API Security Risks 2023
"""

import re
import hashlib
import time
import json
import ipaddress
from typing import Dict, List, Optional, Any, Callable
from datetime import datetime, timedelta
from functools import wraps
from collections import defaultdict
import jwt
import secrets
from urllib.parse import urlparse, parse_qs
import logging
```

```
# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)


class RateLimiter:
    """API4:2023 - Unrestricted Resource Consumption"""

    def __init__(self, max_requests: int = 100, window_seconds: int = 60):
        self.max_requests = max_requests
        self.window_seconds = window_seconds
        self.requests = defaultdict(list)

    def is_allowed(self, identifier: str) -> bool:
        """Check if request is within rate limits"""
        now = time.time()
        window_start = now - self.window_seconds

        # Clean old requests
        self.requests[identifier] = [
            req_time for req_time in self.requests[identifier]
            if req_time > window_start
        ]

        # Check limit
        if len(self.requests[identifier]) >= self.max_requests:
            logger.warning(f"Rate limit exceeded for {identifier}")
            return False

        self.requests[identifier].append(now)
        return True


class AuthorizationManager:
    """
    API1:2023 - Broken Object Level Authorization (BOLA)
    API5:2023 - Broken Function Level Authorization
    """

    def __init__(self, secret_key: str):
        self.secret_key = secret_key
        self.role_permissions = {
            'admin': ['read', 'write', 'delete', 'admin'],
            'user': ['read', 'write'],
            'guest': ['read']
        }
        # Object ownership tracking
        self.object_ownership = {}

    def generate_token(self, user_id: str, role: str, expires_in: int = 3600) -> str:
        """Generate JWT token with role-based claims"""
        payload = {
            'user_id': user_id,
            'role': role,
            'exp': datetime.utcnow() + timedelta(seconds=expires_in),
```

```
            'iat': datetime.utcnow(),
            'jti': secrets.token_urlsafe(16)  # Unique token ID
        }
        return jwt.encode(payload, self.secret_key, algorithm='HS256')

    def verify_token(self, token: str) -> Optional[Dict]:
        """Verify and decode JWT token"""
        try:
            payload = jwt.decode(token, self.secret_key, algorithms=['HS256'])
            return payload
        except jwt.ExpiredSignatureError:
            logger.error("Token has expired")
            return None
        except jwt.InvalidTokenError:
            logger.error("Invalid token")
            return None

    def check_object_authorization(self, user_id: str, object_id: str,
                                   object_type: str) -> bool:
        """
        Check if user has access to specific object
        Prevents BOLA attacks
        """
        key = f"{object_type}:{object_id}"
        owner = self.object_ownership.get(key)

        if owner is None:
            logger.warning(f"Object {key} not found")
            return False

        if owner != user_id:
            logger.warning(f"User {user_id} attempted unauthorized access to {key}")
            return False

        return True

    def register_object(self, object_id: str, object_type: str, owner_id: str):
        """Register object ownership"""
        key = f"{object_type}:{object_id}"
        self.object_ownership[key] = owner_id

    def check_function_authorization(self, role: str, required_permission: str) -> bool:
        """
        Check if role has permission for function
        Prevents unauthorized access to admin/privileged functions
        """
        permissions = self.role_permissions.get(role, [])

        if required_permission not in permissions:
            logger.warning(f"Role {role} lacks permission {required_permission}")
            return False

        return True


class InputValidator:
```

```
    """
    API3:2023 - Broken Object Property Level Authorization
    API8:2023 - Security Misconfiguration
    """

    def __init__(self):
        # SQL Injection patterns
        self.sql_patterns = [
            r"(\bunion\b.*\bselect\b)",
            r"(\bselect\b.*\bfrom\b)",
            r"(\binsert\b.*\binto\b)",
            r"(\bdelete\b.*\bfrom\b)",
            r"(\bdrop\b.*\btable\b)",
            r"(;.*--)",
            r"('.*or.*'.*=.*')",
        ]

        # XSS patterns
        self.xss_patterns = [
            r"<script[^>]*>.*?</script>",
            r"javascript:",
            r"onerror\s*=",
            r"onload\s*=",
            r"<iframe",
        ]

        # Command injection patterns
        self.cmd_patterns = [
            r"[;&|`$]",
            r"\$\(.+\)",
            r"`[^`]+`",
        ]

    def validate_input(self, data: Any, field_name: str = "") -> tuple[bool, str]:
        """Comprehensive input validation"""
        if isinstance(data, str):
            # Check SQL injection
            for pattern in self.sql_patterns:
                if re.search(pattern, data, re.IGNORECASE):
                    logger.warning(f"SQL injection attempt detected in {field_name}: {dat
                    return False, "Invalid input: Potential SQL injection"

            # Check XSS
            for pattern in self.xss_patterns:
                if re.search(pattern, data, re.IGNORECASE):
                    logger.warning(f"XSS attempt detected in {field_name}: {data[:50]}")
                    return False, "Invalid input: Potential XSS"

            # Check command injection
            for pattern in self.cmd_patterns:
                if re.search(pattern, data):
                    logger.warning(f"Command injection attempt in {field_name}: {data[:50
                    return False, "Invalid input: Potential command injection"

        elif isinstance(data, dict):
            for key, value in data.items():
```

```
                valid, msg = self.validate_input(value, key)
                if not valid:
                    return False, msg

        elif isinstance(data, list):
            for item in data:
                valid, msg = self.validate_input(item, field_name)
                if not valid:
                    return False, msg

        return True, "Valid"

    def sanitize_output(self, data: Dict, allowed_fields: List[str]) -> Dict:
        """
        Prevent excessive data exposure
        Only return allowed fields to client
        """
        return {k: v for k, v in data.items() if k in allowed_fields}

    def validate_mass_assignment(self, input_data: Dict,
                                 allowed_fields: List[str]) -> tuple[bool, str]:
        """
        Prevent mass assignment vulnerabilities
        Only allow modification of whitelisted fields
        """
        for field in input_data.keys():
            if field not in allowed_fields:
                logger.warning(f"Mass assignment attempt: unauthorized field '{field}'")
                return False, f"Field '{field}' cannot be modified"

        return True, "Valid"


class SSRFProtection:
    """API7:2023 - Server Side Request Forgery"""

    def __init__(self):
        self.blocked_domains = ['localhost', '127.0.0.1', '0.0.0.0']
        self.allowed_domains = []  # Whitelist of allowed domains

        # Private IP ranges (RFC 1918)
        self.private_ranges = [
            ipaddress.ip_network('10.0.0.0/8'),
            ipaddress.ip_network('172.16.0.0/12'),
            ipaddress.ip_network('192.168.0.0/16'),
            ipaddress.ip_network('127.0.0.0/8'),
            ipaddress.ip_network('169.254.0.0/16'),
        ]

    def is_safe_url(self, url: str) -> tuple[bool, str]:
        """Validate URL to prevent SSRF attacks"""
        try:
            parsed = urlparse(url)

            # Check scheme
            if parsed.scheme not in ['http', 'https']:
```

```
                return False, "Only HTTP/HTTPS schemes allowed"

            # Check for blocked domains
            hostname = parsed.hostname
            if hostname in self.blocked_domains:
                logger.warning(f"SSRF attempt: blocked domain {hostname}")
                return False, "Access to internal resources blocked"

            # Check if whitelist is enabled
            if self.allowed_domains and hostname not in self.allowed_domains:
                return False, "Domain not in whitelist"

            # Check for private IP addresses
            try:
                ip = ipaddress.ip_address(hostname)
                for private_range in self.private_ranges:
                    if ip in private_range:
                        logger.warning(f"SSRF attempt: private IP {ip}")
                        return False, "Access to private IP ranges blocked"
            except ValueError:
                pass  # Not an IP address, continue with domain checks

            return True, "Valid URL"

        except Exception as e:
            logger.error(f"URL validation error: {e}")
            return False, "Invalid URL format"


class BusinessFlowProtection:
    """API6:2023 - Unrestricted Access to Sensitive Business Flows"""

    def __init__(self):
        self.flow_limits = defaultdict(lambda: defaultdict(list))
        # Configure sensitive flows: flow_name -> (max_requests, window_seconds)
        self.flow_configs = {
            'purchase': (5, 3600),      # 5 purchases per hour
            'password_reset': (3, 300),  # 3 resets per 5 minutes
            'account_creation': (2, 60), # 2 accounts per minute
            'file_upload': (10, 600),    # 10 uploads per 10 minutes
        }

    def check_flow_limit(self, flow_name: str, identifier: str) -> bool:
        """Check if business flow usage is within acceptable limits"""
        if flow_name not in self.flow_configs:
            return True

        max_requests, window_seconds = self.flow_configs[flow_name]
        now = time.time()
        window_start = now - window_seconds

        # Clean old requests
        self.flow_limits[flow_name][identifier] = [
            req_time for req_time in self.flow_limits[flow_name][identifier]
            if req_time > window_start
        ]
```

```
        # Check limit
        if len(self.flow_limits[flow_name][identifier]) >= max_requests:
            logger.warning(f"Business flow limit exceeded: {flow_name} by {identifier}")
            return False

        self.flow_limits[flow_name][identifier].append(now)
        return True


class SecurityConfiguration:
    """API8:2023 - Security Misconfiguration"""

    def __init__(self):
        self.secure_headers = {
            'X-Content-Type-Options': 'nosniff',
            'X-Frame-Options': 'DENY',
            'X-XSS-Protection': '1; mode=block',
            'Strict-Transport-Security': 'max-age=31536000; includeSubDomains',
            'Content-Security-Policy': "default-src 'self'",
            'Referrer-Policy': 'strict-origin-when-cross-origin',
        }
        self.cors_allowed_origins = []
        self.api_version = "v1"
        self.deprecated_versions = []

    def get_security_headers(self) -> Dict[str, str]:
        """Return security headers to be added to responses"""
        return self.secure_headers.copy()

    def is_version_allowed(self, version: str) -> bool:
        """API9:2023 - Improper Inventory Management"""
        if version in self.deprecated_versions:
            logger.warning(f"Access to deprecated API version: {version}")
            return False
        return True

    def validate_cors(self, origin: str) -> bool:
        """Validate CORS origin"""
        if not self.cors_allowed_origins:
            return False
        return origin in self.cors_allowed_origins


class ThirdPartyAPIValidator:
    """API10:2023 - Unsafe Consumption of APIs"""

    def __init__(self):
        self.trusted_apis = set()

    def validate_external_data(self, data: Any, source: str) -> tuple[bool, Any]:
        """
        Validate data from third-party APIs
        Apply same rigor as user input validation
        """
        validator = InputValidator()
```

```
        if isinstance(data, dict):
            for key, value in data.items():
                valid, msg = validator.validate_input(value, f"{source}.{key}")
                if not valid:
                    logger.error(f"Invalid data from {source}: {msg}")
                    return False, None

        return True, data

    def is_trusted_api(self, api_endpoint: str) -> bool:
        """Check if external API is in trusted list"""
        return api_endpoint in self.trusted_apis


class WAAPAPISecurityModule:
    """
    Main WAAP API Security Module
    Integrates all OWASP Top 10 API Security protections
    """

    def __init__(self, secret_key: str):
        self.auth_manager = AuthorizationManager(secret_key)
        self.rate_limiter = RateLimiter(max_requests=100, window_seconds=60)
        self.input_validator = InputValidator()
        self.ssrf_protection = SSRFProtection()
        self.business_flow = BusinessFlowProtection()
        self.security_config = SecurityConfiguration()
        self.third_party_validator = ThirdPartyAPIValidator()

        logger.info("WAAP API Security Module initialized")

    def authenticate_request(self, token: str) -> Optional[Dict]:
        """API2:2023 - Broken Authentication"""
        return self.auth_manager.verify_token(token)

    def authorize_object_access(self, user_id: str, object_id: str,
                                object_type: str) -> bool:
        """API1:2023 - Broken Object Level Authorization"""
        return self.auth_manager.check_object_authorization(user_id, object_id, object_ty

    def authorize_function(self, role: str, permission: str) -> bool:
        """API5:2023 - Broken Function Level Authorization"""
        return self.auth_manager.check_function_authorization(role, permission)

    def validate_request_data(self, data: Any) -> tuple[bool, str]:
        """API3:2023 - Broken Object Property Level Authorization"""
        return self.input_validator.validate_input(data)

    def check_rate_limit(self, identifier: str) -> bool:
        """API4:2023 - Unrestricted Resource Consumption"""
        return self.rate_limiter.is_allowed(identifier)

    def check_business_flow(self, flow_name: str, identifier: str) -> bool:
        """API6:2023 - Unrestricted Access to Sensitive Business Flows"""
        return self.business_flow.check_flow_limit(flow_name, identifier)
```

```
    def validate_url(self, url: str) -> tuple[bool, str]:
        """API7:2023 - Server Side Request Forgery"""
        return self.ssrf_protection.is_safe_url(url)

    def get_response_headers(self) -> Dict[str, str]:
        """API8:2023 - Security Misconfiguration"""
        return self.security_config.get_security_headers()

    def validate_api_version(self, version: str) -> bool:
        """API9:2023 - Improper Inventory Management"""
        return self.security_config.is_version_allowed(version)

    def validate_external_data(self, data: Any, source: str) -> tuple[bool, Any]:
        """API10:2023 - Unsafe Consumption of APIs"""
        return self.third_party_validator.validate_external_data(data, source)


# Decorator for protecting API endpoints
def protect_endpoint(waap_module: WAAPAPISecurityModule,
                     required_permission: str = 'read',
                     object_type: Optional[str] = None,
                     business_flow: Optional[str] = None):
    """
    Decorator to protect API endpoints with WAAP security
    """
    def decorator(func: Callable):
        @wraps(func)
        def wrapper(*args, **kwargs):
            # Extract request context (adapt based on your framework)
            request = kwargs.get('request') or (args[^1_0] if args else None)

            if not request:
                return {'error': 'Invalid request'}, 400

            # Get token from Authorization header
            auth_header = getattr(request, 'headers', {}).get('Authorization', '')
            token = auth_header.replace('Bearer ', '') if auth_header else None

            if not token:
                logger.warning("Missing authentication token")
                return {'error': 'Unauthorized'}, 401

            # Authenticate
            user_data = waap_module.authenticate_request(token)
            if not user_data:
                return {'error': 'Invalid or expired token'}, 401

            user_id = user_data['user_id']
            role = user_data['role']

            # Rate limiting
            if not waap_module.check_rate_limit(user_id):
                return {'error': 'Rate limit exceeded'}, 429

            # Function-level authorization
```

```
            if not waap_module.authorize_function(role, required_permission):
                return {'error': 'Insufficient permissions'}, 403

            # Business flow protection
            if business_flow:
                if not waap_module.check_business_flow(business_flow, user_id):
                    return {'error': 'Business flow limit exceeded'}, 429

            # Object-level authorization (if object_type specified)
            if object_type:
                object_id = kwargs.get('object_id') or kwargs.get('id')
                if object_id:
                    if not waap_module.authorize_object_access(user_id, object_id, object
                        return {'error': 'Access denied to resource'}, 403

            # Input validation
            if hasattr(request, 'json') and request.json:
                valid, msg = waap_module.validate_request_data(request.json)
                if not valid:
                    return {'error': msg}, 400

            # Execute the actual endpoint function
            response = func(*args, **kwargs)

            # Add security headers to response
            if isinstance(response, tuple) and len(response) == 2:
                data, status_code = response
                headers = waap_module.get_response_headers()
                return data, status_code, headers

            return response

        return wrapper
    return decorator


# Example usage with Flask
if __name__ == "__main__":
    # Initialize WAAP module
    SECRET_KEY = secrets.token_urlsafe(32)
    waap = WAAPAPISecurityModule(SECRET_KEY)

    # Example: Register object ownership
    waap.auth_manager.register_object('user-123', 'profile', 'user-123')

    # Example: Generate token
    token = waap.auth_manager.generate_token('user-123', 'user')
    print(f"Generated token: {token[:50]}...")

    # Example: Validate input
    test_data = {
        'username': 'john_doe',
        'email': 'john@example.com',
        'comment': 'This is a safe comment'
    }
```

```
    valid, msg = waap.validate_request_data(test_data)
    print(f"Input validation: {valid} - {msg}")

    # Example: Test malicious input
    malicious_data = {
        'username': "admin' OR '1'='1",
        'comment': '<script>alert("XSS")</script>'
    }

    valid, msg = waap.validate_request_data(malicious_data)
    print(f"Malicious input validation: {valid} - {msg}")

    # Example: Test SSRF protection
    safe_url = "https://api.example.com/data"
    unsafe_url = "http://127.0.0.1:8080/admin"

    print(f"Safe URL: {waap.validate_url(safe_url)}")
    print(f"Unsafe URL: {waap.validate_url(unsafe_url)}")
```

Module 2: API Endpoint Discovery Spider/Crawler

This module automatically discovers and catalogs API endpoints across any web application:[3] [4]

```
"""
WAAP - API Endpoint Discovery Spider/Crawler Module
Discovers and catalogs all API endpoints for protection
"""

import re
import json
import hashlib
import asyncio
import aiohttp
from typing import Dict, List, Set, Optional, Tuple
from urllib.parse import urljoin, urlparse, parse_qs
from dataclasses import dataclass, asdict
from datetime import datetime
import logging
from collections import defaultdict

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)


@dataclass
class APIEndpoint:
    """Represents a discovered API endpoint"""
    url: str
    method: str
    path: str
    parameters: List[str]
    headers: Dict[str, str]
    response_codes: Set[int]
    content_type: Optional[str]
```

```
    authentication_required: bool
    discovered_at: str
    last_seen: str
    request_count: int = 0

    def to_dict(self) -> Dict:
        """Convert to dictionary for serialization"""
        data = asdict(self)
        data['response_codes'] = list(self.response_codes)
        return data


class EndpointNormalizer:
    """Normalizes API endpoints to identify unique patterns"""

    def __init__(self):
        # Common ID patterns
        self.id_patterns = [
            (r'/\d+/', '/[ID]/'),                           # Numeric IDs
            (r'/[0-9a-f-]{36}/', '/[UUID]/'),               # UUIDs
            (r'/[0-9a-fA-F]{24}/', '/[MONGODB_ID]/'),       # MongoDB IDs
            (r'/[a-zA-Z0-9_-]{20,}/', '/[TOKEN]/'),         # Long tokens
            (r'/v\d+/', '/[VERSION]/'),                      # API versions
        ]

    def normalize_path(self, path: str) -> str:
        """
        Normalize API path by replacing IDs with placeholders
        Example: /api/users/12345/posts -> /api/users/[ID]/posts
        """
        normalized = path

        for pattern, replacement in self.id_patterns:
            normalized = re.sub(pattern, replacement, normalized)

        return normalized

    def extract_path_variables(self, path: str) -> List[str]:
        """Extract variable segments from path"""
        variables = []

        # Find numeric IDs
        variables.extend(re.findall(r'/(\d+)/', path))

        # Find UUIDs
        variables.extend(re.findall(r'/([0-9a-f-]{36})/', path))

        # Find MongoDB IDs
        variables.extend(re.findall(r'/([0-9a-fA-F]{24})/', path))

        return variables


class APISpider:
    """
    Crawls web application to discover API endpoints
```

```
    Works with any application by analyzing HTTP traffic patterns
    """

    def __init__(self, base_url: str, max_depth: int = 5,
                 max_concurrent: int = 10):
        self.base_url = base_url.rstrip('/')
        self.max_depth = max_depth
        self.max_concurrent = max_concurrent

        # Storage
        self.endpoints: Dict[str, APIEndpoint] = {}
        self.visited_urls: Set[str] = set()
        self.url_queue: asyncio.Queue = asyncio.Queue()

        # Configuration
        self.normalizer = EndpointNormalizer()
        self.api_indicators = [
            '/api/', '/v1/', '/v2/', '/v3/', '/rest/', '/graphql',
            '/swagger', '/openapi', '/docs'
        ]

        # HTTP methods to test
        self.http_methods = ['GET', 'POST', 'PUT', 'PATCH', 'DELETE', 'OPTIONS']

        # Statistics
        self.stats = {
            'total_requests': 0,
            'endpoints_discovered': 0,
            'errors': 0,
            'start_time': None,
            'end_time': None
        }

        logger.info(f"API Spider initialized for {base_url}")

    def is_api_endpoint(self, url: str) -> bool:
        """Determine if URL is likely an API endpoint"""
        parsed = urlparse(url)
        path = parsed.path.lower()

        # Check for API indicators in path
        for indicator in self.api_indicators:
            if indicator in path:
                return True

        # Check for common API patterns
        if re.search(r'/api/|\.json$|/data/', path):
            return True

        return False

    def generate_endpoint_key(self, method: str, normalized_path: str) -> str:
        """Generate unique key for endpoint"""
        return f"{method}:{normalized_path}"

    async def fetch_url(self, session: aiohttp.ClientSession, url: str,
```

```
                    method: str = 'GET', **kwargs) -> Optional[Dict]:
        """Fetch URL and return response details"""
        try:
            async with session.request(method, url, **kwargs) as response:
                self.stats['total_requests'] += 1

                content_type = response.headers.get('Content-Type', '')

                # Try to read response body
                try:
                    if 'application/json' in content_type:
                        body = await response.json()
                    else:
                        body = await response.text()
                except:
                    body = None

                return {
                    'url': url,
                    'method': method,
                    'status_code': response.status,
                    'headers': dict(response.headers),
                    'content_type': content_type,
                    'body': body
                }

        except aiohttp.ClientError as e:
            logger.error(f"Error fetching {url}: {e}")
            self.stats['errors'] += 1
            return None
        except Exception as e:
            logger.error(f"Unexpected error fetching {url}: {e}")
            self.stats['errors'] += 1
            return None

    def extract_links(self, body: str, base_url: str) -> List[str]:
        """Extract links from HTML/JSON content"""
        links = []

        if isinstance(body, str):
            # Find URLs in HTML
            url_pattern = r'href=["\'](.*?)["\']|src=["\'](.*?)["\']'
            matches = re.findall(url_pattern, body)
            for match in matches:
                link = match[^1_0] or match[^1_1]
                if link:
                    absolute_url = urljoin(base_url, link)
                    links.append(absolute_url)

        elif isinstance(body, dict):
            # Extract URLs from JSON
            def extract_urls_from_json(obj):
                if isinstance(obj, dict):
                    for value in obj.values():
                        extract_urls_from_json(value)
                elif isinstance(obj, list):
```

```
                    for item in obj:
                        extract_urls_from_json(item)
                elif isinstance(obj, str):
                    if obj.startswith('http'):
                        links.append(obj)

            extract_urls_from_json(body)

        return links

    def extract_api_schema(self, response_data: Dict) -> Dict:
        """Extract API schema information from response"""
        schema = {
            'fields': [],
            'data_types': {},
            'nested_objects': []
        }

        body = response_data.get('body')

        if isinstance(body, dict):
            for key, value in body.items():
                schema['fields'].append(key)
                schema['data_types'][key] = type(value).__name__

                if isinstance(value, dict):
                    schema['nested_objects'].append(key)

        return schema

    async def probe_endpoint_methods(self, session: aiohttp.ClientSession,
                                     url: str) -> List[Tuple[str, int]]:
        """Probe endpoint with different HTTP methods"""
        results = []

        for method in self.http_methods:
            response_data = await self.fetch_url(session, url, method,
                                                 allow_redirects=False,
                                                 timeout=aiohttp.ClientTimeout(total=10))

            if response_data:
                status_code = response_data['status_code']

                # Only record if not 404/405 (method not allowed)
                if status_code not in [404, 405]:
                    results.append((method, status_code))
                    logger.info(f"Discovered: {method} {url} -> {status_code}")

        return results

    def register_endpoint(self, url: str, method: str, response_data: Dict):
        """Register discovered endpoint"""
        parsed = urlparse(url)
        path = parsed.path
        normalized_path = self.normalizer.normalize_path(path)
```

```
        # Generate endpoint key
        key = self.generate_endpoint_key(method, normalized_path)

        # Extract query parameters
        params = list(parse_qs(parsed.query).keys())

        # Check if authentication required (401/403 status codes)
        auth_required = response_data['status_code'] in [401, 403]

        current_time = datetime.utcnow().isoformat()

        if key in self.endpoints:
            # Update existing endpoint
            endpoint = self.endpoints[key]
            endpoint.response_codes.add(response_data['status_code'])
            endpoint.last_seen = current_time
            endpoint.request_count += 1

            # Merge parameters
            for param in params:
                if param not in endpoint.parameters:
                    endpoint.parameters.append(param)
        else:
            # Create new endpoint
            endpoint = APIEndpoint(
                url=url,
                method=method,
                path=normalized_path,
                parameters=params,
                headers=response_data.get('headers', {}),
                response_codes={response_data['status_code']},
                content_type=response_data.get('content_type'),
                authentication_required=auth_required,
                discovered_at=current_time,
                last_seen=current_time,
                request_count=1
            )

            self.endpoints[key] = endpoint
            self.stats['endpoints_discovered'] += 1

            logger.info(f"Registered new endpoint: {method} {normalized_path}")

    async def crawl_url(self, session: aiohttp.ClientSession, url: str,
                        depth: int):
        """Crawl a single URL and discover endpoints"""
        if depth > self.max_depth or url in self.visited_urls:
            return

        self.visited_urls.add(url)

        # Check if this looks like an API endpoint
        if self.is_api_endpoint(url):
            # Probe with different HTTP methods
            method_results = await self.probe_endpoint_methods(session, url)
```

```
            for method, status_code in method_results:
                response_data = await self.fetch_url(session, url, method)
                if response_data:
                    self.register_endpoint(url, method, response_data)

        # Fetch page to find more links
        response_data = await self.fetch_url(session, url, 'GET')

        if response_data and response_data.get('body'):
            # Extract links for further crawling
            links = self.extract_links(response_data['body'], url)

            for link in links:
                # Only follow links within the same domain
                if urlparse(link).netloc == urlparse(self.base_url).netloc:
                    if link not in self.visited_urls:
                        await self.url_queue.put((link, depth + 1))

    async def worker(self, session: aiohttp.ClientSession):
        """Worker coroutine to process URL queue"""
        while True:
            try:
                url, depth = await asyncio.wait_for(
                    self.url_queue.get(),
                    timeout=1.0
                )

                await self.crawl_url(session, url, depth)
                self.url_queue.task_done()

            except asyncio.TimeoutError:
                break
            except Exception as e:
                logger.error(f"Worker error: {e}")

    async def discover_from_common_paths(self, session: aiohttp.ClientSession):
        """Try common API endpoint paths"""
        common_paths = [
            '/api',
            '/api/v1',
            '/api/v2',
            '/api/users',
            '/api/products',
            '/api/orders',
            '/api/auth',
            '/api/login',
            '/rest',
            '/graphql',
            '/swagger.json',
            '/openapi.json',
            '/api-docs',
        ]

        for path in common_paths:
            url = urljoin(self.base_url, path)
            await self.url_queue.put((url, 0))
```

```
    async def run(self):
        """Main crawl execution"""
        self.stats['start_time'] = datetime.utcnow().isoformat()

        logger.info(f"Starting API discovery for {self.base_url}")

        # Configure session
        timeout = aiohttp.ClientTimeout(total=30)
        connector = aiohttp.TCPConnector(limit=self.max_concurrent)

        async with aiohttp.ClientSession(timeout=timeout,
                                         connector=connector) as session:
            # Add initial URL
            await self.url_queue.put((self.base_url, 0))

            # Try common API paths
            await self.discover_from_common_paths(session)

            # Create worker tasks
            workers = [
                asyncio.create_task(self.worker(session))
                for _ in range(self.max_concurrent)
            ]

            # Wait for queue to be processed
            await self.url_queue.join()

            # Cancel workers
            for worker in workers:
                worker.cancel()

        self.stats['end_time'] = datetime.utcnow().isoformat()

        logger.info(f"Discovery complete. Found {len(self.endpoints)} unique endpoints")
        self.print_summary()

    def print_summary(self):
        """Print discovery summary"""
        print("\n" + "="*80)
        print("API ENDPOINT DISCOVERY SUMMARY")
        print("="*80)
        print(f"Base URL: {self.base_url}")
        print(f"Total Requests: {self.stats['total_requests']}")
        print(f"Endpoints Discovered: {self.stats['endpoints_discovered']}")
        print(f"Errors: {self.stats['errors']}")
        print(f"URLs Visited: {len(self.visited_urls)}")
        print("\nDiscovered Endpoints:")
        print("-"*80)

        # Group by path
        grouped = defaultdict(list)
        for endpoint in self.endpoints.values():
            grouped[endpoint.path].append(endpoint)

        for path in sorted(grouped.keys()):
```

```
            endpoints = grouped[path]
            methods = [e.method for e in endpoints]
            print(f"  {path}")
            print(f"    Methods: {', '.join(methods)}")
            print(f"    Auth Required: {endpoints[^1_0].authentication_required}")
            if endpoints[^1_0].parameters:
                print(f"    Parameters: {', '.join(endpoints[^1_0].parameters)}")
            print()

    def export_endpoints(self, filename: str):
        """Export discovered endpoints to JSON file"""
        data = {
            'base_url': self.base_url,
            'discovery_date': datetime.utcnow().isoformat(),
            'statistics': self.stats,
            'endpoints': [ep.to_dict() for ep in self.endpoints.values()]
        }

        with open(filename, 'w') as f:
            json.dump(data, f, indent=2)

        logger.info(f"Endpoints exported to {filename}")

    def load_endpoints(self, filename: str):
        """Load previously discovered endpoints"""
        with open(filename, 'r') as f:
            data = json.load(f)

        for ep_data in data['endpoints']:
            ep_data['response_codes'] = set(ep_data['response_codes'])
            endpoint = APIEndpoint(**ep_data)
            key = self.generate_endpoint_key(endpoint.method, endpoint.path)
            self.endpoints[key] = endpoint

        logger.info(f"Loaded {len(self.endpoints)} endpoints from {filename}")


class EndpointProtectionMapper:
    """
    Maps discovered endpoints to WAAP protection policies
    Integrates Module 1 and Module 2
    """

    def __init__(self, waap_module, spider: APISpider):
        self.waap = waap_module
        self.spider = spider
        self.protection_policies = {}

    def create_protection_policy(self, endpoint: APIEndpoint) -> Dict:
        """Create protection policy for endpoint"""
        policy = {
            'endpoint': f"{endpoint.method} {endpoint.path}",
            'authentication_required': endpoint.authentication_required,
            'rate_limit': True,
            'input_validation': True,
            'required_permission': 'read' if endpoint.method == 'GET' else 'write',
```

```
        }

        # Determine if it's a sensitive business flow
        sensitive_keywords = ['payment', 'purchase', 'order', 'transaction',
                              'password', 'account', 'admin']

        for keyword in sensitive_keywords:
            if keyword in endpoint.path.lower():
                policy['business_flow_protection'] = True
                policy['required_permission'] = 'admin'
                break

        return policy

    def map_all_endpoints(self):
        """Create protection policies for all discovered endpoints"""
        for key, endpoint in self.spider.endpoints.items():
            policy = self.create_protection_policy(endpoint)
            self.protection_policies[key] = policy
            logger.info(f"Created protection policy for {key}")

    def export_policies(self, filename: str):
        """Export protection policies"""
        with open(filename, 'w') as f:
            json.dump(self.protection_policies, f, indent=2)

        logger.info(f"Protection policies exported to {filename}")


# Example usage
async def main():
    # Initialize API Spider
    target_url = "https://api.example.com"  # Replace with your target
    spider = APISpider(target_url, max_depth=3, max_concurrent=5)

    # Run discovery
    await spider.run()

    # Export discovered endpoints
    spider.export_endpoints("discovered_endpoints.json")

    # Initialize WAAP and create protection policies
    from api_security_module import WAAPAPISecurityModule
    import secrets

    SECRET_KEY = secrets.token_urlsafe(32)
    waap = WAAPAPISecurityModule(SECRET_KEY)

    # Map endpoints to protection policies
    mapper = EndpointProtectionMapper(waap, spider)
    mapper.map_all_endpoints()
    mapper.export_policies("endpoint_protection_policies.json")


if __name__ == "__main__":
    asyncio.run(main())
```

Integration Example: Using Both Modules Together

Here's how to use both modules in a Flask application:

```
"""
Example: Flask application with WAAP protection
"""

from flask import Flask, request, jsonify
from api_security_module import WAAPAPISecurityModule, protect_endpoint
import secrets

app = Flask(__name__)

# Initialize WAAP
SECRET_KEY = secrets.token_urlsafe(32)
waap = WAAPAPISecurityModule(SECRET_KEY)

# Load discovered endpoints and protection policies
# (from Module 2 spider results)
import json
with open('endpoint_protection_policies.json', 'r') as f:
    protection_policies = json.load(f)


@app.route('/api/v1/users/<user_id>', methods=['GET'])
@protect_endpoint(waap, required_permission='read', object_type='user')
def get_user(user_id):
    """Protected endpoint - requires authentication and authorization"""
    return jsonify({
        'user_id': user_id,
        'username': 'john_doe',
        'email': 'john@example.com'
    }), 200


@app.route('/api/v1/users/<user_id>', methods=['PUT'])
@protect_endpoint(waap, required_permission='write', object_type='user')
def update_user(user_id):
    """Protected endpoint with input validation"""
    data = request.json

    # Additional mass assignment protection
    allowed_fields = ['username', 'email', 'bio']
    valid, msg = waap.input_validator.validate_mass_assignment(data, allowed_fields)

    if not valid:
        return jsonify({'error': msg}), 400

    return jsonify({'message': 'User updated', 'user_id': user_id}), 200


@app.route('/api/v1/purchase', methods=['POST'])
@protect_endpoint(waap, required_permission='write', business_flow='purchase')
def create_purchase():
    """Protected sensitive business flow"""
```

```
    data = request.json

    return jsonify({
        'message': 'Purchase created',
        'order_id': 'ORD-12345'
    }), 201


@app.route('/api/v1/admin/users', methods=['DELETE'])
@protect_endpoint(waap, required_permission='admin')
def delete_users():
    """Admin-only endpoint"""
    return jsonify({'message': 'Users deleted'}), 200


# Authentication endpoint (not protected)
@app.route('/api/v1/login', methods=['POST'])
def login():
    """Login endpoint - generates JWT token"""
    data = request.json

    # Validate credentials (simplified)
    username = data.get('username')
    password = data.get('password')

    # In production, verify against database
    if username and password:
        # Generate token
        token = waap.auth_manager.generate_token(
            user_id=username,
            role='user',  # or determine from database
            expires_in=3600
        )

        return jsonify({
            'token': token,
            'expires_in': 3600
        }), 200

    return jsonify({'error': 'Invalid credentials'}), 401


if __name__ == '__main__':
    app.run(debug=False, host='0.0.0.0', port=5000)
```

Key Features Summary

Module 1 - API Security:

- Protects against all OWASP API Top 10 2023 vulnerabilities[2] [1]
- JWT-based authentication with role-based access control
- Object-level and function-level authorization
- Rate limiting and business flow protection

- Input validation against SQL injection, XSS, command injection
- SSRF protection with URL validation
- Mass assignment prevention
- Security headers configuration
- Third-party API data validation

Module 2 - API Spider:

- Automatic endpoint discovery across any web application[3]
- Path normalization to identify unique endpoints
- Multi-method probing (GET, POST, PUT, DELETE, etc.)
- Concurrent crawling with configurable depth
- JSON export of discovered endpoints
- Integration with Module 1 for automatic protection policy generation

These modules work together seamlessly - the spider discovers all endpoints, and the security module automatically protects them against attacks using the same codebase regardless of the application.[5] [6] [7] [8]

⁂

for module 1 , for xss ,sqli and other such all the rule set, i will add more rules as different class file. so give a modular program, where such core rule sets, i can update in future if needed, every vulnerability should be in seperate class file.

Now I'll create a completely modular architecture with separate class files for each vulnerability type, following industry-standard WAF rule engine patterns.[34] [35] [36]

Modular WAAP Architecture - Refactored Module 1

Project Structure

```
waap_module1/
├── __init__.py
├── core/
│   ├── __init__.py
│   ├── base_rule_engine.py      # Base classes for rule engine
│   ├── rule_loader.py            # Dynamic rule loading system
│   └── waap_security.py          # Main WAAP coordinator
├── rules/
│   ├── __init__.py
│   ├── api01_bola.py            # Broken Object Level Authorization
```

```
│   ├── api02_authentication.py   # Broken Authentication
│   ├── api03_property_authorization.py  # Broken Object Property Level Auth
│   ├── api04_rate_limiting.py    # Unrestricted Resource Consumption
│   ├── api05_function_authorization.py  # Broken Function Level Authorization
│   ├── api06_business_flow.py    # Unrestricted Access to Sensitive Business Flows
│   ├── api07_ssrf.py             # Server Side Request Forgery
│   ├── api08_security_config.py  # Security Misconfiguration
│   ├── api09_inventory.py        # Improper Inventory Management
│   └── api10_unsafe_consumption.py  # Unsafe Consumption of APIs
├── validators/
│   ├── __init__.py
│   ├── sql_injection.py          # SQL Injection detection
│   ├── xss_detection.py          # XSS detection
│   ├── command_injection.py      # Command injection detection
│   ├── path_traversal.py         # Path traversal detection
│   ├── ldap_injection.py         # LDAP injection detection
│   └── xxe_detection.py          # XXE detection
├── config/
│   ├── __init__.py
│   ├── rule_config.yaml          # Rule configuration
│   └── thresholds.yaml           # Scoring thresholds
└── utils/
    ├── __init__.py
    ├── logger.py                 # Logging utilities
    └── decorators.py             # Protection decorators
```

1. Base Rule Engine (`core/base_rule_engine.py`)

```python
"""
Base classes for WAAP rule engine
Provides plugin architecture for modular rule development
"""

from abc import ABC, abstractmethod
from typing import Dict, List, Optional, Any, Tuple
from dataclasses import dataclass
from enum import Enum
import logging

logger = logging.getLogger(__name__)


class RuleSeverity(Enum):
    """Rule severity levels"""
    CRITICAL = 5
    HIGH = 4
    MEDIUM = 3
    LOW = 2
    INFO = 1


class RuleAction(Enum):
    """Actions to take when rule matches"""
    BLOCK = "block"
    ALERT = "alert"
```

```
    LOG = "log"
    ALLOW = "allow"


@dataclass
class RuleMatch:
    """Represents a rule match result"""
    rule_id: str
    rule_name: str
    severity: RuleSeverity
    action: RuleAction
    score: int
    message: str
    matched_pattern: Optional[str] = None
    matched_field: Optional[str] = None
    metadata: Dict[str, Any] = None

    def __post_init__(self):
        if self.metadata is None:
            self.metadata = {}


@dataclass
class RuleContext:
    """Context passed to rules for evaluation"""
    request_id: str
    user_id: Optional[str] = None
    role: Optional[str] = None
    ip_address: Optional[str] = None
    endpoint: Optional[str] = None
    method: Optional[str] = None
    headers: Optional[Dict[str, str]] = None
    query_params: Optional[Dict[str, Any]] = None
    body: Optional[Dict[str, Any]] = None
    url: Optional[str] = None
    metadata: Optional[Dict[str, Any]] = None

    def __post_init__(self):
        if self.headers is None:
            self.headers = {}
        if self.query_params is None:
            self.query_params = {}
        if self.body is None:
            self.body = {}
        if self.metadata is None:
            self.metadata = {}


class BaseRule(ABC):
    """
    Abstract base class for all security rules
    All vulnerability-specific rules must inherit from this
    """

    def __init__(self, rule_id: str, name: str, description: str,
                 severity: RuleSeverity, enabled: bool = True):
```

```
        self.rule_id = rule_id
        self.name = name
        self.description = description
        self.severity = severity
        self.enabled = enabled
        self.match_count = 0
        self.false_positive_count = 0

        logger.info(f"Initialized rule: {rule_id} - {name}")

    @abstractmethod
    def evaluate(self, context: RuleContext) -> List[RuleMatch]:
        """
        Evaluate the rule against the given context
        Returns list of matches (empty if no match)
        """
        pass

    def get_score(self) -> int:
        """Get anomaly score for this rule"""
        return self.severity.value

    def get_action(self) -> RuleAction:
        """Get default action for this rule"""
        if self.severity in [RuleSeverity.CRITICAL, RuleSeverity.HIGH]:
            return RuleAction.BLOCK
        elif self.severity == RuleSeverity.MEDIUM:
            return RuleAction.ALERT
        else:
            return RuleAction.LOG

    def is_enabled(self) -> bool:
        """Check if rule is enabled"""
        return self.enabled

    def disable(self):
        """Disable this rule"""
        self.enabled = False
        logger.info(f"Rule {self.rule_id} disabled")

    def enable(self):
        """Enable this rule"""
        self.enabled = True
        logger.info(f"Rule {self.rule_id} enabled")


class BaseValidator(ABC):
    """
    Abstract base class for input validators
    Used for specific attack pattern detection (SQLi, XSS, etc.)
    """

    def __init__(self, name: str, enabled: bool = True):
        self.name = name
        self.enabled = enabled
        self.patterns: List[str] = []
```

```
        self.load_patterns()

    @abstractmethod
    def load_patterns(self):
        """Load detection patterns"""
        pass

    @abstractmethod
    def validate(self, data: Any, field_name: str = "") -> Tuple[bool, Optional[str], Opt
        """
        Validate input data
        Returns: (is_valid, error_message, matched_pattern)
        """
        pass

    def is_enabled(self) -> bool:
        """Check if validator is enabled"""
        return self.enabled


class RuleEngine:
    """
    Main rule engine that orchestrates all rules
    """

    def __init__(self, anomaly_threshold: int = 5):
        self.rules: Dict[str, BaseRule] = {}
        self.validators: Dict[str, BaseValidator] = {}
        self.anomaly_threshold = anomaly_threshold
        self.total_requests = 0
        self.blocked_requests = 0

        logger.info(f"Rule engine initialized with threshold {anomaly_threshold}")

    def register_rule(self, rule: BaseRule):
        """Register a security rule"""
        self.rules[rule.rule_id] = rule
        logger.info(f"Registered rule: {rule.rule_id}")

    def register_validator(self, validator: BaseValidator):
        """Register an input validator"""
        self.validators[validator.name] = validator
        logger.info(f"Registered validator: {validator.name}")

    def evaluate_all(self, context: RuleContext) -> Tuple[bool, List[RuleMatch], int]:
        """
        Evaluate all enabled rules against context
        Returns: (should_block, matches, total_score)
        """
        self.total_requests += 1
        matches = []
        total_score = 0

        for rule in self.rules.values():
            if not rule.is_enabled():
                continue
```

```
            try:
                rule_matches = rule.evaluate(context)

                if rule_matches:
                    matches.extend(rule_matches)
                    rule.match_count += len(rule_matches)

                    # Calculate score
                    for match in rule_matches:
                        total_score += match.score

            except Exception as e:
                logger.error(f"Error evaluating rule {rule.rule_id}: {e}")

        # Determine if request should be blocked
        should_block = total_score >= self.anomaly_threshold

        if should_block:
            self.blocked_requests += 1
            logger.warning(
                f"Request {context.request_id} blocked. "
                f"Score: {total_score}, Threshold: {self.anomaly_threshold}"
            )

        return should_block, matches, total_score

    def get_statistics(self) -> Dict[str, Any]:
        """Get engine statistics"""
        return {
            'total_requests': self.total_requests,
            'blocked_requests': self.blocked_requests,
            'block_rate': self.blocked_requests / max(self.total_requests, 1),
            'rule_count': len(self.rules),
            'enabled_rules': sum(1 for r in self.rules.values() if r.is_enabled()),
            'validator_count': len(self.validators)
        }
```

2. Rule Loader (`core/rule_loader.py`)

```
"""
Dynamic rule loading system
Automatically discovers and loads rule modules
"""

import os
import importlib
import inspect
from typing import List, Type
from pathlib import Path
import logging

from .base_rule_engine import BaseRule, BaseValidator, RuleEngine

logger = logging.getLogger(__name__)
```

```
class RuleLoader:
    """
    Dynamically loads rules and validators from modules
    Supports hot-reloading for updates
    """

    def __init__(self, rules_dir: str = "rules", validators_dir: str = "validators"):
        self.rules_dir = rules_dir
        self.validators_dir = validators_dir
        self.loaded_modules = {}

    def discover_rules(self) -> List[Type[BaseRule]]:
        """Discover all rule classes in rules directory"""
        rule_classes = []

        rules_path = Path(__file__).parent.parent / self.rules_dir

        if not rules_path.exists():
            logger.warning(f"Rules directory not found: {rules_path}")
            return rule_classes

        for file_path in rules_path.glob("*.py"):
            if file_path.name.startswith("__"):
                continue

            module_name = f"waap_module1.{self.rules_dir}.{file_path.stem}"

            try:
                module = importlib.import_module(module_name)
                self.loaded_modules[module_name] = module

                # Find all BaseRule subclasses in module
                for name, obj in inspect.getmembers(module):
                    if (inspect.isclass(obj) and
                        issubclass(obj, BaseRule) and
                        obj is not BaseRule):
                        rule_classes.append(obj)
                        logger.info(f"Discovered rule class: {name} from {file_path.name}

            except Exception as e:
                logger.error(f"Error loading module {module_name}: {e}")

        return rule_classes

    def discover_validators(self) -> List[Type[BaseValidator]]:
        """Discover all validator classes in validators directory"""
        validator_classes = []

        validators_path = Path(__file__).parent.parent / self.validators_dir

        if not validators_path.exists():
            logger.warning(f"Validators directory not found: {validators_path}")
            return validator_classes
```

```
        for file_path in validators_path.glob("*.py"):
            if file_path.name.startswith("__"):
                continue

            module_name = f"waap_module1.{self.validators_dir}.{file_path.stem}"

            try:
                module = importlib.import_module(module_name)
                self.loaded_modules[module_name] = module

                # Find all BaseValidator subclasses in module
                for name, obj in inspect.getmembers(module):
                    if (inspect.isclass(obj) and
                        issubclass(obj, BaseValidator) and
                        obj is not BaseValidator):
                        validator_classes.append(obj)
                        logger.info(f"Discovered validator class: {name} from {file_path.

            except Exception as e:
                logger.error(f"Error loading module {module_name}: {e}")

        return validator_classes

    def load_all_rules(self, engine: RuleEngine):
        """Load and register all discovered rules"""
        rule_classes = self.discover_rules()

        for rule_class in rule_classes:
            try:
                # Instantiate rule with default config
                rule_instance = rule_class()
                engine.register_rule(rule_instance)
            except Exception as e:
                logger.error(f"Error instantiating rule {rule_class.__name__}: {e}")

    def load_all_validators(self, engine: RuleEngine):
        """Load and register all discovered validators"""
        validator_classes = self.discover_validators()

        for validator_class in validator_classes:
            try:
                validator_instance = validator_class()
                engine.register_validator(validator_instance)
            except Exception as e:
                logger.error(f"Error instantiating validator {validator_class.__name__}:

    def reload_module(self, module_name: str):
        """Hot-reload a specific module"""
        if module_name in self.loaded_modules:
            try:
                importlib.reload(self.loaded_modules[module_name])
                logger.info(f"Reloaded module: {module_name}")
            except Exception as e:
                logger.error(f"Error reloading module {module_name}: {e}")
```

3. SQL Injection Validator (`validators/sql_injection.py`)

```
"""
SQL Injection Detection Validator
Easily extensible with new patterns
"""

import re
from typing import Any, Tuple, Optional, List
from ..core.base_rule_engine import BaseValidator
import logging

logger = logging.getLogger(__name__)


class SQLInjectionValidator(BaseValidator):
    """
    Detects SQL injection attacks
    Patterns based on OWASP CRS and industry best practices
    """

    def __init__(self, enabled: bool = True, paranoia_level: int = 1):
        self.paranoia_level = paranoia_level
        super().__init__("sql_injection", enabled)

    def load_patterns(self):
        """
        Load SQL injection detection patterns
        Organized by paranoia level for granular control
        """
        # Level 1: High confidence patterns (low false positives)
        self.level1_patterns = [
            # Union-based SQLi
            (r"(?i)\bunion\b.{1,100}\bselect\b", "SQL: UNION SELECT"),
            (r"(?i)\bunion\b.{1,100}\ball\b.{1,100}\bselect\b", "SQL: UNION ALL SELECT"),

            # Boolean-based SQLi
            (r"(?i)'\s*or\s*'1'\s*=\s*'1", "SQL: OR '1'='1"),
            (r"(?i)'\s*or\s*1\s*=\s*1\s*--", "SQL: OR 1=1 --"),
            (r"(?i)'\s*or\s*true\s*--", "SQL: OR true --"),

            # Comment injection
            (r"(?i)(;|')\s*--", "SQL: Comment injection"),
            (r"(?i)/\*.*\*/", "SQL: Block comment"),
            (r"#.*$", "SQL: Hash comment"),

            # Stacked queries
            (r"(?i);\s*(drop|delete|insert|update|alter)\b", "SQL: Stacked query"),

            # SQL commands
            (r"(?i)\b(drop|delete)\b.{1,100}\b(table|database|schema)\b", "SQL: DROP/DELE
            (r"(?i)\bexec(\s|\()", "SQL: EXEC command"),
            (r"(?i)\bexecute\b.{1,50}\b(immediate|sp_)", "SQL: EXECUTE command"),

            # Information schema
            (r"(?i)\binformation_schema\b", "SQL: Information schema access"),
```

```
            (r"(?i)\bsys\.(tables|columns|databases)", "SQL: System catalog access"),
        ]

        # Level 2: Medium confidence (moderate false positives)
        self.level2_patterns = [
            (r"(?i)'\s*and\s*'.*?'\s*=\s*'", "SQL: AND condition"),
            (r"(?i)\bor\b\s+\d+\s*=\s*\d+", "SQL: OR numeric comparison"),
            (r"(?i)\bselect\b.{1,100}\bfrom\b", "SQL: SELECT FROM"),
            (r"(?i)\binsert\b.{1,100}\binto\b", "SQL: INSERT INTO"),
            (r"(?i)\bupdate\b.{1,100}\bset\b", "SQL: UPDATE SET"),
            (r"(?i)\bwhere\b.{1,50}\b(or|and)\b", "SQL: WHERE condition"),
            (r"(?i)\bhaving\b.{1,50}\b(or|and)\b", "SQL: HAVING condition"),
            (r"(?i)\bgroup\s+by\b", "SQL: GROUP BY"),
            (r"(?i)\border\s+by\b", "SQL: ORDER BY"),
        ]

        # Level 3: Lower confidence (higher false positives)
        self.level3_patterns = [
            (r"(?i)\bselect\b", "SQL: SELECT keyword"),
            (r"(?i)\bfrom\b\s+\w+", "SQL: FROM keyword"),
            (r"(?i)\bwhere\b", "SQL: WHERE keyword"),
            (r"(?i)\bdelete\b", "SQL: DELETE keyword"),
            (r"(?i)\bupdate\b", "SQL: UPDATE keyword"),
            (r"(?i)\binsert\b", "SQL: INSERT keyword"),
            (r"[\'\";]", "SQL: Quote character"),
        ]

        # Level 4: Very aggressive (many false positives)
        self.level4_patterns = [
            (r"=", "SQL: Equal sign"),
            (r"--", "SQL: Double dash"),
            (r"\bor\b", "SQL: OR keyword"),
            (r"\band\b", "SQL: AND keyword"),
        ]

        # Compile patterns based on paranoia level
        self.patterns = []

        if self.paranoia_level >= 1:
            self.patterns.extend(self.level1_patterns)
        if self.paranoia_level >= 2:
            self.patterns.extend(self.level2_patterns)
        if self.paranoia_level >= 3:
            self.patterns.extend(self.level3_patterns)
        if self.paranoia_level >= 4:
            self.patterns.extend(self.level4_patterns)

        # Compile regex patterns for performance
        self.compiled_patterns = [
            (re.compile(pattern), desc)
            for pattern, desc in self.patterns
        ]

        logger.info(f"Loaded {len(self.compiled_patterns)} SQL injection patterns "
                    f"(paranoia level {self.paranoia_level})")
```

```
    def validate(self, data: Any, field_name: str = "") -> Tuple[bool, Optional[str], Opt
        """
        Validate data for SQL injection attempts
        Returns: (is_valid, error_message, matched_pattern)
        """
        if not self.enabled:
            return True, None, None

        if isinstance(data, str):
            return self._check_string(data, field_name)

        elif isinstance(data, dict):
            for key, value in data.items():
                is_valid, msg, pattern = self.validate(value, f"{field_name}.{key}")
                if not is_valid:
                    return False, msg, pattern

        elif isinstance(data, list):
            for idx, item in enumerate(data):
                is_valid, msg, pattern = self.validate(item, f"{field_name}[{idx}]")
                if not is_valid:
                    return False, msg, pattern

        return True, None, None

    def _check_string(self, text: str, field_name: str) -> Tuple[bool, Optional[str], Opt
        """Check individual string for SQL injection"""
        for regex, description in self.compiled_patterns:
            match = regex.search(text)
            if match:
                logger.warning(
                    f"SQL injection detected in {field_name}: {description} - "
                    f"Matched: {match.group()[:50]}"
                )
                return (
                    False,
                    f"Potential SQL injection detected: {description}",
                    match.group()
                )

        return True, None, None

    def add_custom_pattern(self, pattern: str, description: str):
        """Add a custom SQL injection pattern at runtime"""
        try:
            compiled = re.compile(pattern)
            self.compiled_patterns.append((compiled, description))
            logger.info(f"Added custom SQL injection pattern: {description}")
        except re.error as e:
            logger.error(f"Invalid regex pattern: {e}")

    def set_paranoia_level(self, level: int):
        """Change paranoia level and reload patterns"""
        if 1 <= level <= 4:
            self.paranoia_level = level
```

```
            self.load_patterns()
            logger.info(f"SQL injection paranoia level set to {level}")
```

4. XSS Detection Validator (`validators/xss_detection.py`)

```
"""
Cross-Site Scripting (XSS) Detection Validator
"""

import re
from typing import Any, Tuple, Optional
from ..core.base_rule_engine import BaseValidator
import logging
import html

logger = logging.getLogger(__name__)


class XSSDetectionValidator(BaseValidator):
    """
    Detects XSS (Cross-Site Scripting) attacks
    Supports reflected, stored, and DOM-based XSS patterns
    """

    def __init__(self, enabled: bool = True, paranoia_level: int = 1):
        self.paranoia_level = paranoia_level
        super().__init__("xss_detection", enabled)

    def load_patterns(self):
        """Load XSS detection patterns"""

        # Level 1: High confidence XSS patterns
        self.level1_patterns = [
            # Script tags
            (r"(?i)<script[^>]*>.*?</script>", "XSS: Script tag"),
            (r"(?i)<script[^>]*>", "XSS: Script tag opening"),

            # JavaScript protocol
            (r"(?i)javascript\s*:", "XSS: JavaScript protocol"),
            (r"(?i)vbscript\s*:", "XSS: VBScript protocol"),

            # Event handlers
            (r"(?i)\bon\w+\s*=", "XSS: Event handler"),
            (r"(?i)onerror\s*=", "XSS: onerror handler"),
            (r"(?i)onload\s*=", "XSS: onload handler"),
            (r"(?i)onclick\s*=", "XSS: onclick handler"),
            (r"(?i)onmouseover\s*=", "XSS: onmouseover handler"),

            # Iframe injection
            (r"(?i)<iframe[^>]*>", "XSS: iframe tag"),
            (r"(?i)<embed[^>]*>", "XSS: embed tag"),
            (r"(?i)<object[^>]*>", "XSS: object tag"),

            # Data URI
            (r"(?i)data:text/html", "XSS: Data URI HTML"),
```

```
            (r"(?i)data:text/javascript", "XSS: Data URI JavaScript"),

            # Expression
            (r"(?i)expression\s*\(", "XSS: CSS expression"),
            (r"(?i)-moz-binding\s*:", "XSS: XBL binding"),
        ]

        # Level 2: Medium confidence patterns
        self.level2_patterns = [
            (r"(?i)<img[^>]*>", "XSS: img tag"),
            (r"(?i)<svg[^>]*>", "XSS: svg tag"),
            (r"(?i)<input[^>]*>", "XSS: input tag"),
            (r"(?i)<form[^>]*>", "XSS: form tag"),
            (r"(?i)<meta[^>]*>", "XSS: meta tag"),
            (r"(?i)<link[^>]*>", "XSS: link tag"),
            (r"(?i)<base[^>]*>", "XSS: base tag"),
            (r"(?i)alert\s*\(", "XSS: alert function"),
            (r"(?i)confirm\s*\(", "XSS: confirm function"),
            (r"(?i)prompt\s*\(", "XSS: prompt function"),
            (r"(?i)eval\s*\(", "XSS: eval function"),
        ]

        # Level 3: Lower confidence patterns
        self.level3_patterns = [
            (r"<[a-zA-Z]", "XSS: HTML tag start"),
            (r"&#\d+;", "XSS: Numeric entity"),
            (r"&#x[0-9a-fA-F]+;", "XSS: Hex entity"),
            (r"(?i)document\.", "XSS: document object"),
            (r"(?i)window\.", "XSS: window object"),
            (r"(?i)location\.", "XSS: location object"),
        ]

        # Compile patterns based on paranoia level
        self.patterns = []

        if self.paranoia_level >= 1:
            self.patterns.extend(self.level1_patterns)
        if self.paranoia_level >= 2:
            self.patterns.extend(self.level2_patterns)
        if self.paranoia_level >= 3:
            self.patterns.extend(self.level3_patterns)

        self.compiled_patterns = [
            (re.compile(pattern, re.DOTALL), desc)
            for pattern, desc in self.patterns
        ]

        logger.info(f"Loaded {len(self.compiled_patterns)} XSS patterns "
                    f"(paranoia level {self.paranoia_level})")

    def validate(self, data: Any, field_name: str = "") -> Tuple[bool, Optional[str], Opt
        """Validate data for XSS attempts"""
        if not self.enabled:
            return True, None, None

        if isinstance(data, str):
```

```
            return self._check_string(data, field_name)

        elif isinstance(data, dict):
            for key, value in data.items():
                is_valid, msg, pattern = self.validate(value, f"{field_name}.{key}")
                if not is_valid:
                    return False, msg, pattern

        elif isinstance(data, list):
            for idx, item in enumerate(data):
                is_valid, msg, pattern = self.validate(item, f"{field_name}[{idx}]")
                if not is_valid:
                    return False, msg, pattern

        return True, None, None

    def _check_string(self, text: str, field_name: str) -> Tuple[bool, Optional[str], Opt
        """Check individual string for XSS"""
        # Decode HTML entities first
        decoded = html.unescape(text)

        # Check both original and decoded versions
        for check_text in [text, decoded]:
            for regex, description in self.compiled_patterns:
                match = regex.search(check_text)
                if match:
                    logger.warning(
                        f"XSS detected in {field_name}: {description} - "
                        f"Matched: {match.group()[:50]}"
                    )
                    return (
                        False,
                        f"Potential XSS attack detected: {description}",
                        match.group()
                    )

        return True, None, None

    def sanitize(self, text: str) -> str:
        """Sanitize string by escaping HTML"""
        return html.escape(text)
```

5. Command Injection Validator (`validators/command_injection.py`)

```
"""
Command Injection Detection Validator
"""

import re
from typing import Any, Tuple, Optional
from ..core.base_rule_engine import BaseValidator
import logging

logger = logging.getLogger(__name__)
```

```
class CommandInjectionValidator(BaseValidator):
    """
    Detects command injection attacks
    Prevents OS command execution vulnerabilities
    """

    def __init__(self, enabled: bool = True):
        super().__init__("command_injection", enabled)

    def load_patterns(self):
        """Load command injection patterns"""

        self.patterns = [
            # Command separators
            (r"[;&|`]", "CMD: Command separator"),
            (r"\$\(.+\)", "CMD: Command substitution $()"),
            (r"`[^`]+`", "CMD: Backtick substitution"),

            # Redirects and pipes
            (r"[<>]\s*\w", "CMD: Redirect operator"),
            (r"\|\s*\w", "CMD: Pipe operator"),
            (r">>\s*\w", "CMD: Append redirect"),

            # Common dangerous commands
            (r"(?i)\b(cat|ls|pwd|whoami|id|uname|wget|curl|nc|netcat|bash|sh)\b",
             "CMD: Common Unix command"),
            (r"(?i)\b(cmd|powershell|wscript|cscript|rundll32)\b",
             "CMD: Windows command"),

            # Environment variables
            (r"\$\w+", "CMD: Environment variable"),
            (r"%\w+%", "CMD: Windows environment variable"),

            # Path traversal in commands
            (r"\.\./", "CMD: Path traversal"),
            (r"/etc/", "CMD: /etc/ access"),
            (r"/bin/", "CMD: /bin/ access"),

            # Remote code execution
            (r"(?i)eval\s*\(", "CMD: eval function"),
            (r"(?i)exec\s*\(", "CMD: exec function"),
            (r"(?i)system\s*\(", "CMD: system function"),
            (r"(?i)passthru\s*\(", "CMD: passthru function"),
        ]

        self.compiled_patterns = [
            (re.compile(pattern), desc)
            for pattern, desc in self.patterns
        ]

        logger.info(f"Loaded {len(self.compiled_patterns)} command injection patterns")

    def validate(self, data: Any, field_name: str = "") -> Tuple[bool, Optional[str], Opt
        """Validate data for command injection attempts"""
        if not self.enabled:
```

```
            return True, None, None

        if isinstance(data, str):
            return self._check_string(data, field_name)

        elif isinstance(data, dict):
            for key, value in data.items():
                is_valid, msg, pattern = self.validate(value, f"{field_name}.{key}")
                if not is_valid:
                    return False, msg, pattern

        elif isinstance(data, list):
            for idx, item in enumerate(data):
                is_valid, msg, pattern = self.validate(item, f"{field_name}[{idx}]")
                if not is_valid:
                    return False, msg, pattern

        return True, None, None

    def _check_string(self, text: str, field_name: str) -> Tuple[bool, Optional[str], Opt
        """Check individual string for command injection"""
        for regex, description in self.compiled_patterns:
            match = regex.search(text)
            if match:
                logger.warning(
                    f"Command injection detected in {field_name}: {description} - "
                    f"Matched: {match.group()[:50]}"
                )
                return (
                    False,
                    f"Potential command injection detected: {description}",
                    match.group()
                )

        return True, None, None
```

6. Input Validation Rule (`rules/api03_property_authorization.py`)

```
"""
API03:2023 - Broken Object Property Level Authorization
Includes comprehensive input validation using modular validators
"""

from typing import List
from ..core.base_rule_engine import (
    BaseRule, RuleContext, RuleMatch,
    RuleSeverity, RuleAction
)
from ..validators.sql_injection import SQLInjectionValidator
from ..validators.xss_detection import XSSDetectionValidator
from ..validators.command_injection import CommandInjectionValidator
import logging

logger = logging.getLogger(__name__)
```

```
class InputValidationRule(BaseRule):
    """
    Validates all input data against injection attacks
    Uses modular validators for different attack types
    """

    def __init__(self):
        super().__init__(
            rule_id="API03-001",
            name="Comprehensive Input Validation",
            description="Validates input against SQLi, XSS, command injection, etc.",
            severity=RuleSeverity.HIGH
        )

        # Initialize validators
        self.validators = {
            'sql_injection': SQLInjectionValidator(paranoia_level=2),
            'xss': XSSDetectionValidator(paranoia_level=2),
            'command_injection': CommandInjectionValidator()
        }

    def evaluate(self, context: RuleContext) -> List[RuleMatch]:
        """Evaluate input validation"""
        matches = []

        # Validate query parameters
        for validator_name, validator in self.validators.items():
            if context.query_params:
                is_valid, msg, pattern = validator.validate(
                    context.query_params,
                    "query_params"
                )

                if not is_valid:
                    matches.append(RuleMatch(
                        rule_id=f"{self.rule_id}-{validator_name}",
                        rule_name=f"{self.name} - {validator_name}",
                        severity=self.severity,
                        action=self.get_action(),
                        score=self.get_score(),
                        message=msg,
                        matched_pattern=pattern,
                        matched_field="query_params"
                    ))

            # Validate body
            if context.body:
                is_valid, msg, pattern = validator.validate(
                    context.body,
                    "body"
                )

                if not is_valid:
                    matches.append(RuleMatch(
                        rule_id=f"{self.rule_id}-{validator_name}",
```

```
                        rule_name=f"{self.name} - {validator_name}",
                        severity=self.severity,
                        action=self.get_action(),
                        score=self.get_score(),
                        message=msg,
                        matched_pattern=pattern,
                        matched_field="body"
                    ))

        return matches


class MassAssignmentProtectionRule(BaseRule):
    """
    Prevents mass assignment vulnerabilities
    """

    def __init__(self):
        super().__init__(
            rule_id="API03-002",
            name="Mass Assignment Protection",
            description="Prevents unauthorized field modifications",
            severity=RuleSeverity.MEDIUM
        )

        # Define protected fields that should never be modified by users
        self.protected_fields = [
            'id', 'user_id', 'created_at', 'updated_at',
            'is_admin', 'role', 'permissions', 'password_hash',
            'salt', 'token', 'api_key'
        ]

    def evaluate(self, context: RuleContext) -> List[RuleMatch]:
        """Check for mass assignment attempts"""
        matches = []

        if not context.body or not isinstance(context.body, dict):
            return matches

        for field in context.body.keys():
            if field.lower() in [f.lower() for f in self.protected_fields]:
                matches.append(RuleMatch(
                    rule_id=self.rule_id,
                    rule_name=self.name,
                    severity=self.severity,
                    action=self.get_action(),
                    score=self.get_score(),
                    message=f"Attempt to modify protected field: {field}",
                    matched_field=field
                ))

        return matches
```

7. Main WAAP Coordinator (`core/waap_security.py`)

```
"""
Main WAAP Security Module
Coordinates all rules and validators
"""

from typing import Dict, List, Optional, Tuple
from .base_rule_engine import RuleEngine, RuleContext, RuleMatch
from .rule_loader import RuleLoader
import logging
import uuid

logger = logging.getLogger(__name__)


class WAAPSecurityModule:
    """
    Main WAAP security coordinator
    Integrates all modular rules and validators
    """

    def __init__(self, anomaly_threshold: int = 5, auto_load_rules: bool = True):
        self.engine = RuleEngine(anomaly_threshold=anomaly_threshold)
        self.loader = RuleLoader()

        if auto_load_rules:
            self.load_all_rules()

        logger.info("WAAP Security Module initialized")

    def load_all_rules(self):
        """Load all rules and validators"""
        self.loader.load_all_validators(self.engine)
        self.loader.load_all_rules(self.engine)
        logger.info("All rules and validators loaded")

    def evaluate_request(self, **kwargs) -> Tuple[bool, List[RuleMatch], int]:
        """
        Evaluate a request against all rules

        Args:
            Various request components (method, url, headers, body, etc.)

        Returns:
            (should_block, matches, total_score)
        """
        # Create context
        context = RuleContext(
            request_id=kwargs.get('request_id', str(uuid.uuid4())),
            user_id=kwargs.get('user_id'),
            role=kwargs.get('role'),
            ip_address=kwargs.get('ip_address'),
            endpoint=kwargs.get('endpoint'),
            method=kwargs.get('method'),
            headers=kwargs.get('headers'),
```

```
            query_params=kwargs.get('query_params'),
            body=kwargs.get('body'),
            url=kwargs.get('url'),
            metadata=kwargs.get('metadata')
        )

        # Evaluate all rules
        return self.engine.evaluate_all(context)

    def get_statistics(self) -> Dict:
        """Get WAAP statistics"""
        return self.engine.get_statistics()

    def reload_rules(self):
        """Reload all rules (for updates)"""
        self.load_all_rules()
        logger.info("Rules reloaded")
```

8. Usage Example

```
"""
Example: Using the modular WAAP system
"""

from waap_module1.core.waap_security import WAAPSecurityModule
import logging

logging.basicConfig(level=logging.INFO)

# Initialize WAAP
waap = WAAPSecurityModule(anomaly_threshold=5)

# Example 1: Test malicious SQL injection
result = waap.evaluate_request(
    method="POST",
    endpoint="/api/users",
    body={
        "username": "admin' OR '1'='1",
        "email": "test@example.com"
    }
)

should_block, matches, score = result
print(f"Block: {should_block}, Score: {score}")
for match in matches:
    print(f"  - {match.rule_name}: {match.message}")

# Example 2: Test XSS
result = waap.evaluate_request(
    method="GET",
    endpoint="/api/search",
    query_params={
        "q": "<script>alert('XSS')</script>"
    }
)
```

```
should_block, matches, score = result
print(f"\nBlock: {should_block}, Score: {score}")
for match in matches:
    print(f"  - {match.rule_name}: {match.message}")

# Example 3: Safe request
result = waap.evaluate_request(
    method="GET",
    endpoint="/api/users/123",
    headers={"Authorization": "Bearer token123"}
)

should_block, matches, score = result
print(f"\nBlock: {should_block}, Score: {score}")

# Get statistics
stats = waap.get_statistics()
print(f"\nStatistics: {stats}")
```

Key Features of This Modular Architecture

1. **Separation of Concerns** :
   - Each vulnerability type in its own file
   - Easy to update individual rule sets
   - Validators separate from rules
2. **Plugin Architecture** :
   - Automatic rule discovery
   - Hot-reload capability
   - Easy to add new rules/validators
3. **Paranoia Levels** :
   - Configurable sensitivity
   - Balance between security and false positives
4. **Extensibility** :
   - Add custom patterns at runtime
   - Override base classes for custom logic
   - Configuration through YAML files
5. **Scoring System** :
   - Anomaly-based detection
   - Configurable thresholds
   - Multiple rule aggregation

This modular design allows you to easily add new validators or update existing ones without touching the core engine, following industry best practices from OWASP ModSecurity Core Rule

Set .

⁂

1. https://owasp.org/API-Security/editions/2023/en/0x11-t10/
2. https://apisecurity.io/owasp-api-security-top-10/
3. https://blog.cloudflare.com/ml-api-discovery-and-schema-learning/
4. https://www.cequence.ai/products/api-spyder/
5. https://cashmere.io/v/Gz6Qly
6. https://cashmere.io/v/1apjUd
7. https://cashmere.io/v/D1uS0y
8. https://cashmere.io/v/uzXYlO
9. https://www.techrxiv.org/articles/preprint/An_OWASP_Top_Ten_Driven_Survey_on_Web_Application_Protection_Methods/13265180/files/25545929.pdf
10. https://www.mdpi.com/2076-3417/12/8/4077/pdf?version=1650281818
11. https://arxiv.org/pdf/2405.01111.pdf
12. https://arxiv.org/pdf/2212.06606.pdf
13. https://arxiv.org/html/2306.08869v2
14. https://arxiv.org/pdf/2201.10833.pdf
15. https://www.techscience.com/cmc/v64n3/39444
16. https://dl.acm.org/doi/pdf/10.1145/3658644.3690294
17. https://owasp.org/www-project-api-security/
18. https://portswigger.net/web-security/api-testing/top-10-api-vulnerabilities
19. https://owasp.org/www-project-top-ten/
20. https://www.grapestechsolutions.com/blog/python-securityt-best-practices-for-apis-and-microservices/
21. https://www.cloudflare.com/learning/security/api/owasp-api-security-top-10/
22. https://www.browserstack.com/guide/rest-api-design-principles-and-best-practices
23. https://spider.cloud/guides/spider-api
24. https://www.apisecuniversity.com/courses/owasp-api-security-top-10-and-beyond
25. https://www.moesif.com/blog/technical/api-development/Mastering-Python-with-REST-API-Your-Essential-Guide-to-Building-Robust-Web-Services/
26. https://www.baeldung.com/java-webmagic-web-crawler
27. https://owasp.org/API-Security/editions/2023/en/0x00-header/
28. https://blog.vidocsecurity.com/blog/api-security-best-practices-for-developers
29. https://github.com/BruceDone/awesome-crawler
30. https://www.wiz.io/academy/owasp-api-security
31. https://dev.to/biswajitfsd/mastering-rest-api-best-practices-in-python-5bda
32. https://www.geeksforgeeks.org/blogs/api-security-best-practices/
33. https://cashmere.io/v/Krn5Yu

34. https://owasp.org/www-project-modsecurity-core-rule-set/
35. https://www.netnea.com/cms/apache-tutorial-7_including-modsecurity-core-rules/
36. https://www.radware.com/cyberpedia/application-security/waf-architecture/
37. https://ccsenet.org/journal/index.php/cis/article/download/4279/3726
38. https://www.mdpi.com/1424-8220/22/20/8024/pdf?version=1666268588
39. https://coreruleset.org
40. https://owasp.org/www-chapter-dorset/assets/presentations/2023-06/Introduction_to_the_OWASP_ModSecurity_Core_Rule_Set_Project.pdf
41. https://github.com/coreruleset/coreruleset
42. https://github.com/irods/irods_rule_engine_plugin_python
43. https://www.prosec-networks.com/en/blog/modsecurity-core-rule-sets-und-eigene-regeln/
44. https://papers.ssrn.com/sol3/Delivery.cfm/de573061-7335-4eab-a80d-266517517446-MECA.pdf?abstractid=5263014&mirid=1
45. https://www.nected.ai/blog/python-rule-engines-automate-and-enforce-with-python
46. https://docs.cpanel.net/knowledge-base/security/owasp-modsecurity-crs/
47. https://aws.amazon.com/blogs/security/defense-in-depth-using-aws-managed-rules-for-aws-waf-part-1/
48. https://stackoverflow.com/questions/467738/implementing-a-rules-engine-in-python
49. https://www.f5.com/company/events/webinars/modsecurity-and-nginx-tuning-the-owasp-core-rule-set
50. https://www.cloudflare.com/learning/ddos/glossary/web-application-firewall-waf/
51. https://zato.io/en/docs/4.1/rule-engine/tutorial.html
52. https://learn.microsoft.com/en-us/azure/web-application-firewall/ag/application-gateway-crs-rulegroups-rules
53. https://www.indusface.com/blog/how-web-application-firewall-works/
54. https://community.sonarsource.com/t/i-want-to-develop-some-plugins-for-python-scanning-but-the-documentation-is-too-simple-for-me/62937
55. https://www.cisoplatform.com/profiles/blogs/mastering-owasp-modsecurity-core-rule-set-4-sampling-mode-christi
56. http://arxiv.org/pdf/2406.13547.pdf
57. https://arxiv.org/pdf/1803.05529.pdf
58. http://arxiv.org/pdf/2308.04964.pdf
59. https://downloads.hindawi.com/journals/wcmc/2022/1657627.pdf
60. https://downloads.hindawi.com/journals/wcmc/2021/8028073.pdf
61. https://dl.acm.org/doi/pdf/10.1145/3658644.3690227