VulScanner

Technical Documentation

Comprehensive Web Application Vulnerability Scanner
Built with Flask, Redis, and Advanced Security Testing Modules

Key Features	
OWASP Top 10 Coverage	Complete vulnerability detection
Multi-threaded Scanning	Parallel processing with Redis
RESTful API	Integration-ready endpoints
Multiple Databases	SQLite, PostgreSQL, MySQL support
VS Code Integration	Full development environment
Real-time Progress	Live scan status tracking

Generated: September 24, 2025
Platform: Windows
Python Version: 3.12.5

Table of Contents

Table of Contents

Overview

VulScanner is a comprehensive web application vulnerability scanner built with Flask. It performs automated security assessments on web applications, identifying common vulnerabilities following OWASP guidelines and security best practices.

Key Features

- **Multi-threaded scanning** with parallel processing
- **Real-time progress tracking** via Redis
- **Comprehensive vulnerability detection** (OWASP Top 10)
- **RESTful API** for integration
- **Web-based dashboard** with reporting
- **Database persistence** (SQLite/PostgreSQL)
- **Customizable scan configurations**
- **Security headers validation**

Technology Stack

Backend Framework

- **Flask** (v1.0.1) - Core web framework
- **Flask-RESTful** - REST API implementation
- **Flask-HTTPAuth** - Authentication handling
- **Werkzeug** (v1.0.1) - WSGI utilities

Database & Caching

- **SQLite3** - Default database (development)
- **PostgreSQL** - Production database support
- **Redis** (v3.5.3) - Session management and caching
- **SQLAlchemy** - Database ORM (via database.py)

Security & Scanning

- **python-nmap** (v0.6.1) - Network port scanning
- **requests** (v2.24.0) - HTTP client for web scanning
- **BeautifulSoup4** (v4.9.1) - HTML parsing
- **cryptography** (v3.0) - Encryption utilities
- **bcrypt** (v3.1.7) - Password hashing

System & Utilities

- **psutil** (v5.7.2) - System monitoring
- **paramiko** (v2.7.1) - SSH functionality
- **dnspython** (v2.0.0) - DNS resolution
- **validators** - Input validation

Reporting & Documentation

- **ReportLab** (v3.5.46) - PDF report generation
- **Jinja2** (v2.11.2) - Template engine
- **Sphinx** (v3.1.2) - Documentation generation

Architecture

Application Structure

```
vuln-scanner-flask/ ■■■ core/ # Core business logic ■ ■■■ database.py # Database
models & operations ■ ■■■ redis.py # Redis operations ■ ■■■ workers.py #
```

```
Background task workers ■ ■■■ url_scanner.py # URL scanning engine ■ ■■■
port_scanner.py # Network port scanning ■ ■■■ security.py # Security utilities ■
■■■ reports.py # Report generation ■ ■■■ user_manager.py # User authentication
■■■ scanner/ # Vulnerability detection modules ■ ■■■ scanner/ ■ ■■■ core/ #
Scanner engine ■ ■■■ checks/ # Vulnerability checks ■■■ views/ # Web interface
blueprints ■■■ views_api/ # REST API endpoints ■■■ templates/ # HTML templates
■■■ static/ # Static assets (CSS, JS, images) ■■■ main.py # Application entry
point
```

Request Flow

1. **Client Request** → Flask Application
2. **Authentication** → User Manager
3. **Route Handling** → Blueprint Views
4. **Business Logic** → Core Modules
5. **Data Storage** → Database/Redis
6. **Response** → Templates/JSON API

Dependencies

Production Dependencies

```
# Web Framework flask==1.0.1 flask_restful flask_httpauth Werkzeug==1.0.1 # Database
& Storage redis==3.5.3 psycopg2-binary==2.8.5 mysql-connector==2.2.9
pymongo==3.11.0 # Security & Scanning requests==2.24.0 python-nmap==0.6.1
beautifulsoup4==4.9.1 cryptography==3.0 bcrypt==3.1.7 validators # System Utilities
psutil==5.7.2 paramiko==2.7.1 dnspython==2.0.0 # Reporting reportlab==3.5.46
Pillow==7.2.0 PyPDF2==1.26.0 # Parsing & Templates Jinja2==2.11.2 html5lib==1.1 bs4
```

Development Dependencies

```
# Documentation Sphinx==3.1.2 sphinx-rtd-theme==0.5.0 # Utilities simplejson==3.17.2
itsdangerous==1.1.0
```

Core Components

1. Scanner Engine (`scanner/core/scanner_engine.py`)

- **Orchestrates** vulnerability scanning process
- **Manages** concurrent scanning tasks
- **Coordinates** with individual check modules
- **Handles** rate limiting and error recovery

2. HTTP Client (`scanner/core/http_client.py`)

- **Custom HTTP client** for security testing
- **Request/Response handling** with timeout management
- **Cookie and session management**
- **Proxy support** for testing environments

3. Redis Operations (`core/redis.py`)

- **Session state management**
- **Scan progress tracking**
- **Real-time status updates**
- **Task queue management**
- **Fallback** to mock implementation when Redis unavailable

4. Database Layer (`core/database.py`)

- **SQLAlchemy models** for data persistence
- **User management** and authentication
- **Scan results** and vulnerability storage
- **Report generation** data

5. Worker Processes (`core/workers.py`)

- **Background task processing**
- **Scan orchestration**
- **Parallel vulnerability checking**
- **Resource management**

6. URL Scanner (`core/url_scanner.py`)

- **Web application discovery**
- **URL enumeration** and crawling
- **Content analysis** and classification

- **Form detection** and parameter extraction

7. Port Scanner (`core/port_scanner.py`)

- **Network service discovery**
- **Port state detection**
- **Service fingerprinting**
- **Integration** with nmap

Security Testing Modules

Base Check Framework (`scanner/checks/base.py`)

All security checks inherit from `BaseCheck` abstract class:

- **Standardized interface** for vulnerability detection
- **Logging integration** for debugging
- **Result formatting** with Finding objects
- **Error handling** and recovery

Individual Check Modules

1. SQL Injection (`sql_injection.py`)

- **Error-based detection** using database error patterns
- **Union-based testing** with unique markers
- **Time-based blind detection** (short delays)
- **Boolean-based testing** via response comparison
- **Multiple database support** (MySQL, PostgreSQL, SQLite, Oracle, MSSQL)

2. Cross-Site Scripting (`reflected_xss.py`)

- **Reflected XSS detection** in parameters
- **Context-aware analysis** (HTML, attribute, script)
- **Multiple payload types** (script tags, event handlers, URL schemes)
- **Form-based XSS testing**
- **Confidence scoring** based on payload and context

3. Security Headers (`security_headers.py`)

- **Missing header detection** (CSP, HSTS, X-Frame-Options, etc.)
- **Information disclosure** checks (Server, X-Powered-By)
- **CSP policy analysis** (unsafe-inline, unsafe-eval, wildcards)
- **HSTS configuration validation**

- **X-Frame-Options** security assessment

4. Server-Side Request Forgery (`ssrf.py`)

- **Internal network** access attempts
- **Cloud metadata** service testing
- **Port scanning** via SSRF
- **Protocol handler** testing (file://, gopher://)

5. Directory Traversal (`directory_traversal.py`)

- **Path traversal** payload testing
- **File inclusion** attempts
- **Operating system** specific payloads
- **Encoded payload** variants

6. Open Redirect (`open_redirect.py`)

- **URL redirection** testing
- **Multiple redirect** parameter names
- **Domain validation** bypass attempts
- **JavaScript-based** redirects

7. Broken Access Control (`broken_access_control.py`)

- **Horizontal privilege** escalation
- **Vertical privilege** escalation
- **Direct object** reference testing
- **Parameter manipulation**

8. Authentication Bypass (`authentication_bypass.py`)

- **SQL injection** in login forms
- **Default credentials** testing
- **Session manipulation**
- **Password brute forcing** (when enabled)

9. Information Disclosure (`information_disclosure.py`)

- **Sensitive file** exposure
- **Error message** analysis
- **Comment and metadata** extraction
- **Version information** disclosure

10. Security Misconfiguration (`security_misconfiguration.py`)

- **Default configurations** detection
- **Unnecessary services** identification
- **Verbose error** messages
- **Debug mode** detection

Attack Types & Vulnerabilities

OWASP Top 10 Coverage

A01: Broken Access Control

- **Horizontal/Vertical Privilege Escalation**
- **Direct Object References**
- **Force Browsing**
- **Parameter Manipulation**

A02: Cryptographic Failures

- **Weak SSL/TLS Configuration**
- **Missing HSTS Headers**
- **Insecure Data Transmission**

A03: Injection

- **SQL Injection** (Error, Union, Boolean, Time-based)
- **NoSQL Injection**
- **Command Injection**
- **LDAP Injection**

A04: Insecure Design

- **Missing Security Controls**
- **Weak Business Logic**
- **Trust Boundary Violations**

A05: Security Misconfiguration

- **Default Credentials**
- **Unnecessary Services**
- **Verbose Error Messages**
- **Missing Security Headers**

A06: Vulnerable Components

- **Outdated Libraries**
- **Known CVE Detection**
- **Dependency Analysis**

A07: Identification & Authentication Failures

- **Weak Password Policies**
- **Session Management Issues**
- **Authentication Bypass**
- **Brute Force Vulnerabilities**

A08: Software & Data Integrity Failures

- **Insecure Deserialization**

- **Supply Chain Attacks**
- **Code Integrity Issues**

A09: Security Logging & Monitoring Failures

- **Insufficient Logging**
- **Log Injection**
- **Missing Monitoring**

A10: Server-Side Request Forgery (SSRF)

- **Internal Network Access**
- **Cloud Metadata Access**
- **Port Scanning via SSRF**
- **Protocol Handler Abuse**

Additional Vulnerability Classes

Client-Side Vulnerabilities

- **Cross-Site Scripting (XSS)**
- Reflected XSS
- Stored XSS
- DOM-based XSS
- **Clickjacking**
- **Cross-Site Request Forgery (CSRF)**

Network & Infrastructure

- **Open Ports** and services
- **SSL/TLS Misconfigurations**
- **DNS Issues**
- **HTTP Security Headers**

Business Logic Flaws

- **Race Conditions**
- **Workflow Bypasses**
- **Price Manipulation**
- **Logic Flow Issues**

Integrations

1. Redis Integration (`core/redis.py`)

- **Connection Management**: Automatic connection with fallback
- **Session Storage**: Scan states and progress tracking

- **Task Queuing**: Background job management
- **Real-time Updates**: Progress broadcasting
- **Mock Fallback**: When Redis unavailable

```
# Redis Configuration RDS_HOST = '127.0.0.1' RDS_PORT = 6379 RDS_PASSW = None
```

2. Database Integration

- **SQLite** (Default): File-based database for development
- **PostgreSQL**: Production-ready with connection pooling
- **MySQL**: Alternative production database
- **MongoDB**: Document store for flexible data

```
# Database Configuration DB_URL = "sqlite:///vulnscanner.db" DB_PORT =
os.environ.get('DB_PORT', '5432') DB_NAME = os.environ.get('DB_NAME',
'vulnscanner')
```

3. OWASP Integration

- **ZAP API Integration** (planned)
- **OWASP Dependency Check**
- **CWE Classification** for vulnerabilities
- **CVSS Scoring** for risk assessment

4. External Services

- **SMTP Integration** for email reports
- **Webhook Support** for CI/CD integration
- **API Keys** for third-party security services
- **Cloud Storage** for report archiving

5. Development Tools

- **VS Code Integration** with custom tasks and debugging
- **Docker Support** for containerized deployment
- **GitHub Actions** for CI/CD pipeline
- **Monitoring Integration** (Prometheus, Grafana)

Database Schema

Core Tables

Users

```
CREATE TABLE users ( id INTEGER PRIMARY KEY, username VARCHAR(80) UNIQUE NOT NULL, password_hash VARCHAR(128) NOT NULL, email VARCHAR(120), created_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP, last_login TIMESTAMP );
```

Scans

```
CREATE TABLE scans ( id INTEGER PRIMARY KEY, name VARCHAR(200) NOT NULL, target_url VARCHAR(500) NOT NULL, status VARCHAR(50) DEFAULT 'pending', start_time TIMESTAMP, end_time TIMESTAMP, user_id INTEGER REFERENCES users(id), config_json TEXT );
```

Findings

```
CREATE TABLE findings ( id INTEGER PRIMARY KEY, scan_id INTEGER REFERENCES scans(id), url VARCHAR(500) NOT NULL, title VARCHAR(200) NOT NULL, severity VARCHAR(20) NOT NULL, confidence INTEGER DEFAULT 0, cwe INTEGER, description TEXT, evidence TEXT, payload TEXT, remediation TEXT, created_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP );
```

Reports

```
CREATE TABLE reports ( id INTEGER PRIMARY KEY, scan_id INTEGER REFERENCES scans(id), format VARCHAR(20) DEFAULT 'html', file_path VARCHAR(500), generated_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP );
```

API Endpoints

REST API (`views_api/`)

Health Check

```
GET /health Response: {"status": "healthy", "version": "x.x.x"}
```

Scan Management

```
POST /api/scan Body: { "target": "https://example.com", "scan_type": "quick|full", "config": {...} } GET /api/scan/<scan_id> Response: { "id": "scan_id", "status": "running|completed|failed", "progress": 75, "findings": [...] } DELETE /api/scan/<scan_id>
```

System Updates

```
POST /api/update POST /api/update/<component> Body: {"component": "scanner|rules|signatures"}
```

Exclusion Management

```
GET /api/exclusion POST /api/exclusion Body: { "type": "url|param|header",
"pattern": "regex_pattern", "scope": "global|scan" }
```

Web Interface Routes (`views/`)

- `/login` - User authentication
- `/dashboard` - Main control panel
- `/qs` - Quick scan interface
- `/reports` - Scan reports and history
- `/settings` - Configuration management
- `/assessment` - Detailed vulnerability assessment
- `/topology` - Network topology visualization

Configuration

Application Configuration (`config.py`)

Web Server Settings

```
WEB_HOST = '0.0.0.0' WEB_PORT = 8080 WEB_DEBUG = False WEB_USER = 'admin' WEB_PASSW
= 'admin'
```

Security Headers

```
WEB_SECURITY = True WEB_SEC_HEADERS = { 'CSP': "default-src 'self' 'unsafe-inline';
object-src 'none'", 'CTO': 'nosniff', 'XSS': '1; mode=block', 'XFO': 'DENY', 'RP':
'no-referrer', 'Server': 'vulscanner' }
```

Scan Configuration

```
DEFAULT_SCAN = { 'targets': { 'networks': [], 'domains': [], 'urls': [] }, 'config':
{ 'allow_aggressive': 3, 'allow_dos': False, 'allow_bf': False, 'parallel_scan': 50,
'parallel_attack': 30, 'frequency': 'once' } }
```

Environment Variables

- `DB_PASSWORD` - Database password
- `DB_NAME` - Database name
- `DB_PORT` - Database port
- `WEB_USER` - Default admin username

- `WEB_PASSW` - Default admin password
- `FLASK_ENV` - Flask environment (development/production)

Development Workflow

Setup & Installation

```
# Clone repository git clone <repository-url> cd vuln-scanner-flask # Install dependencies pip install -r requirements.txt # Setup database python setup_sqlite.py # Initialize application python main.py
```

Development Commands

```
# Run application python main.py # Setup database python setup_sqlite.py # Test components python setup_simple.py # Verify branding python verify_vulscanner.py # Run specific tests python test_owasp_scanner.py
```

Code Structure Guidelines

- **Modular Design**: Separate concerns into distinct modules
- **Error Handling**: Comprehensive exception handling
- **Logging**: Structured logging throughout application
- **Testing**: Unit tests for critical components
- **Documentation**: Inline documentation for complex logic

Security Considerations

- **Input Validation**: All user inputs validated
- **SQL Injection Prevention**: Parameterized queries
- **XSS Prevention**: Output encoding
- **CSRF Protection**: Token-based protection
- **Session Security**: Secure session management

VS Code Integration

Configuration Files

Launch Configurations (`.vscode/launch.json`)

- **Start VulScanner**: Main application launcher
- **Setup Database**: Database initialization
- **Test Components**: Component testing
- **CSS Development**: Development with live CSS editing
- **Verify Branding**: Branding consistency check

Tasks (`.vscode/tasks.json`)

- **Build Tasks**: Start application, setup database
- **Test Tasks**: Component testing, Python path checking
- **Development Tasks**: CSS validation, branding verification
- **Utility Tasks**: Browser opening, file watching

Settings (`.vscode/settings.json`)

- **Python Configuration**: Interpreter, linting, formatting
- **CSS Enhancements**: Validation, completion, syntax highlighting
- **File Associations**: Proper syntax highlighting
- **Debugging**: Integrated terminal, problem matchers

Code Snippets (`.vscode/vulscanner.code-snippets`)

- **CSS Snippets**: Brand text, logo text, responsive design
- **HTML Snippets**: VulScanner branding elements
- **Development Shortcuts**: Common patterns and structures

Development Features

- **Integrated Debugging**: Step-through debugging for Python
- **CSS Live Editing**: Real-time CSS development
- **Task Integration**: One-click application lifecycle management
- **Code Completion**: IntelliSense for VulScanner-specific patterns
- **Error Detection**: Real-time syntax and logic error detection

---

Deployment

Development Deployment

```
# Single command start python main.py # With specific configuration
FLASK_ENV=development python main.py # Using VS Code Ctrl+Shift+P → "Tasks: Run
Task" → "Start VulScanner"
```

Production Deployment

Using Gunicorn

```
pip install gunicorn gunicorn -w 4 -b 0.0.0.0:8080 main:app
```

Using Docker

```
FROM python:3.9 WORKDIR /app COPY requirements.txt . RUN pip install -r
requirements.txt COPY . . EXPOSE 8080 CMD ["python", "main.py"]
```

Environment Setup

```
# Production environment variables export FLASK_ENV=production export
DB_PASSWORD=secure_password export DB_NAME=vulnscanner_prod export WEB_DEBUG=False
```

Security Hardening

- **HTTPS Enforcement**: SSL/TLS configuration
- **Database Security**: Encrypted connections, access controls
- **Network Security**: Firewall rules, network segmentation
- **Authentication**: Strong password policies, MFA
- **Monitoring**: Logging, alerting, audit trails

Scaling Considerations

- **Load Balancing**: Multiple application instances
- **Database Scaling**: Read replicas, connection pooling
- **Redis Clustering**: High availability caching
- **Resource Monitoring**: CPU, memory, disk usage
- **Performance Optimization**: Code profiling, query optimization

---

Monitoring & Maintenance

Application Monitoring

- **Health Endpoints**: `/health` for load balancer checks
- **Metrics Collection**: Scan completion rates, error rates
- **Performance Monitoring**: Response times, resource usage
- **Log Aggregation**: Centralized logging for troubleshooting

Security Monitoring

- **Authentication Logs**: Failed login attempts, unusual access patterns

- **Scan Activity**: Unusual scan patterns, potential abuse
- **Error Monitoring**: Security-related errors and exceptions
- **Vulnerability Updates**: New security checks and signatures

Maintenance Tasks

- **Database Cleanup**: Old scan results, report archiving
- **Log Rotation**: Prevent disk space issues
- **Security Updates**: Regular dependency updates
- **Performance Tuning**: Query optimization, caching strategies
- **Backup & Recovery**: Regular backups, disaster recovery testing

---

This documentation provides a comprehensive overview of the VulScanner project's technical architecture, capabilities, and development workflow. For specific implementation details, refer to the individual module documentation within the codebase.