OWASP Dependency-Check Integration Plan

Overview

This document outlines the comprehensive plan for integrating OWASP Dependency-Check into the existing security scanner framework. OWASP Dependency-Check is a Software Composition Analysis (SCA) tool that attempts to detect publicly disclosed vulnerabilities contained within a project's dependencies.

1. OWASP Dependency-Check Overview

What is OWASP Dependency-Check?

- **Purpose**: Identifies project dependencies and checks if there are any known, publicly disclosed vulnerabilities
- **Coverage**: Java, .NET, JavaScript, Python, Ruby, PHP, and more
- **Database**: Uses National Vulnerability Database (NVD) and other sources
- **Output**: Supports HTML, XML, JSON, CSV, and SARIF formats
- **License**: Apache License 2.0 (Open Source)

Key Features

- **Multi-language Support**: Supports 15+ programming languages and package managers
- **CVE Detection**: Identifies Common Vulnerabilities and Exposures (CVEs)
- **CVSS Scoring**: Provides Common Vulnerability Scoring System scores
- **False Positive Suppression**: Allows suppression of false positives
- **CI/CD Integration**: Command-line interface suitable for automation
- **Extensive Reporting**: Multiple output formats with detailed vulnerability information

2. Integration Architecture

PROF

2.1 Scanner Class Structure

Following the existing pattern established by other scanners in the framework:

```
class DependencyCheckScanner(BaseScanner):
    """OWASP Dependency-Check scanner implementation."""

    @property
    def name(self) -> str:
        return "dependency-check"

    @property
    def supported_targets(self) -> List[str]:
        return [
            "git_repository",
            "filesystem"
```

```
        ]

    @property
    def required_tools(self) -> List[str]:
        return ["dependency-check"]
```

2.2 Target Type Support

- **git_repository**: Scan entire Git repositories for dependency vulnerabilities
- **filesystem**: Scan specific filesystem paths containing project files

2.3 Language and Package Manager Support

The scanner will automatically detect and scan:

- **Java**: Maven (pom.xml), Gradle (build.gradle), Ivy
- **JavaScript/Node.js**: npm (package.json, package-lock.json), Yarn
- **Python**: pip (requirements.txt, setup.py), Pipenv, Poetry
- **Ruby**: Bundler (Gemfile, Gemfile.lock)
- **PHP**: Composer (composer.json, composer.lock)
- **C#/.NET**: NuGet (packages.config, *.csproj)
- **Go**: go.mod files
- **Rust**: Cargo.toml
- **Swift**: Package.swift
- **Scala**: SBT build files

3. Implementation Details

3.1 Command Construction

The scanner will build commands based on target type and detected project files:

PROF

```
# Basic scan command
dependency-check --project "ProjectName" --scan /path/to/project --
format JSON --out /output/path

# Advanced options
dependency-check \
  --project "ProjectName" \
  --scan /path/to/project \
  --format JSON \
  --format HTML \
  --out /output/path \
  --suppression /path/to/suppression.xml \
  --enableRetired \
  --enableExperimental \
  --nvdApiKey YOUR_API_KEY
```

3.2 Output Parsing

The scanner will parse JSON output to extract:

- **Vulnerability Information**: CVE IDs, descriptions, severity scores
- **Dependency Details**: Component names, versions, file paths
- **Evidence**: File evidence that led to dependency identification
- **Remediation**: Available fixes and version recommendations

3.3 Finding Mapping

Map Dependency-Check findings to the framework's Finding model:

```
Finding(
    id=vulnerability.get("name", "UNKNOWN"),  # CVE ID
    title=f"Vulnerable dependency: {component_name}",
    description=vulnerability.get("description", ""),
    severity=self._map_cvss_to_severity(cvss_score),
    scanner=self.name,
    target=target.path,
    location=f"{file_path}:{component_name}",
    cve_id=vulnerability.get("name") if vulnerability.get("name",
"").startswith("CVE-") else None,
    cvss_score=cvss_score,
    references=vulnerability.get("references", []),
    remediation=self._build_remediation_advice(vulnerability,
component),
    metadata={
        "component_name": component_name,
        "component_version": component_version,
        "file_path": file_path,
        "confidence": evidence.get("confidence", "UNKNOWN"),
        "evidence_type": evidence.get("type", ""),
        "vulnerable_software": vulnerability.get("vulnerableSoftware",
[])
    }
)
```


4. Configuration Options

4.1 Scanner Configuration

```
dependency-check:
  enabled: true
  timeout: 1800  # 30 minutes (dependency scanning can be slow)
  severity_threshold: "MEDIUM"
  additional_args:
    - "--enableRetired"
    - "--enableExperimental"
```

```
    - "--nvdApiKey"
    - "YOUR_NVD_API_KEY"
  suppression_file: "dependency-check-suppressions.xml"
  update_database: true
  database_directory: "/opt/dependency-check/data"
```

4.2 Advanced Configuration Options

- **Database Updates**: Control when to update the vulnerability database
- **Suppression Files**: Support for false positive suppression
- **API Keys**: NVD API key for faster database updates
- **Analyzers**: Enable/disable specific analyzers (e.g., retired, experimental)
- **Proxy Settings**: Support for corporate proxy environments

5. Installation and Dependencies

5.1 Docker Integration

Add to the existing Dockerfile:

```
# Install OWASP Dependency-Check
RUN wget -O /tmp/dependency-check.zip \

https://github.com/jeremylong/DependencyCheck/releases/download/v8.4.3/d
ependency-check-8.4.3-release.zip \
    && unzip /tmp/dependency-check.zip -d /opt/ \
    && ln -s /opt/dependency-check/bin/dependency-check.sh
/usr/local/bin/dependency-check \
    && chmod +x /usr/local/bin/dependency-check \
    && rm /tmp/dependency-check.zip

# Create data directory for vulnerability database
RUN mkdir -p /opt/dependency-check/data \
    && chown -R scanner:scanner /opt/dependency-check
```

PROF

5.2 System Requirements

- **Java Runtime**: OpenJDK 8 or higher (already included in base image)
- **Memory**: Minimum 4GB RAM recommended for large projects
- **Storage**: ~2GB for vulnerability database
- **Network**: Internet access for database updates

6. Performance Considerations

6.1 Optimization Strategies

- **Database Caching**: Cache vulnerability database between scans

- **Incremental Scanning**: Only scan changed dependencies when possible
- **Parallel Processing**: Utilize multiple CPU cores for large projects
- **Memory Management**: Configure JVM heap size based on project size

6.2 Timeout Configuration

- **Default Timeout**: 30 minutes (1800 seconds)
- **Large Projects**: May require 60+ minutes
- **CI/CD Considerations**: Balance thoroughness vs. pipeline speed

7. Integration Points

7.1 Scanner Registry

Update `src/scanners/__init__.py`:

```
from .dependency_check import DependencyCheckScanner

AVAILABLE_SCANNERS = {
    # ... existing scanners
    'dependency-check': DependencyCheckScanner
}
```

7.2 Configuration Templates

Add to scan request templates:

```
# examples/templates/scan-requests/dependency-scan.yaml
scanners:
  dependency-check:
    enabled: true
    timeout: 1800
    severity_threshold: "MEDIUM"
    additional_args:
      - "--enableRetired"
```

PROF

8. Testing Strategy

8.1 Unit Tests

- Test command construction for different target types
- Test output parsing with sample JSON responses
- Test finding mapping and severity conversion
- Test error handling and timeout scenarios

8.2 Integration Tests

- Test with real projects containing known vulnerabilities
- Test with different programming languages and package managers
- Test database update functionality
- Test suppression file handling

8.3 Test Projects

Create test projects with known vulnerabilities:

- **Java**: Maven project with vulnerable dependencies
- **JavaScript**: Node.js project with vulnerable npm packages
- **Python**: Python project with vulnerable pip packages

9. Documentation Updates

9.1 README Updates

- Add Dependency-Check to the list of supported scanners
- Update the scanner coverage matrix
- Add example usage for dependency scanning

9.2 Configuration Documentation

- Document all configuration options
- Provide examples for different use cases
- Document suppression file format and usage

10. Future Enhancements

10.1 Advanced Features

- **SBOM Integration**: Generate Software Bill of Materials
- **License Analysis**: Analyze dependency licenses for compliance
- **Dependency Graph**: Visualize dependency relationships
- **Risk Scoring**: Advanced risk assessment based on usage patterns

PROF

10.2 Performance Improvements

- **Incremental Scanning**: Only scan changed dependencies
- **Distributed Scanning**: Split large projects across multiple workers
- **Smart Caching**: Intelligent caching based on dependency versions

11. Success Metrics

11.1 Functional Metrics

- **Language Coverage**: Support for 10+ programming languages
- **Accuracy**: <5% false positive rate
- **Performance**: Complete scans within configured timeout

- **Integration**: Seamless integration with existing CI/CD pipelines

11.2 Adoption Metrics

- **Usage**: Dependency-Check enabled in 80%+ of scan configurations
- **Findings**: Consistent vulnerability detection across projects
- **Remediation**: Clear, actionable remediation advice

12. Implementation Timeline

Phase 1 (Week 1): Core Implementation

- [ ] Implement DependencyCheckScanner class
- [ ] Add command construction and output parsing
- [ ] Update scanner registry and configuration
- [ ] Basic unit tests

Phase 2 (Week 2): Integration and Testing

- [ ] Docker image updates
- [ ] Integration tests with sample projects
- [ ] Configuration template updates
- [ ] Documentation updates

Phase 3 (Week 3): Advanced Features and Optimization

- [ ] Suppression file support
- [ ] Performance optimizations
- [ ] Advanced configuration options
- [ ] Comprehensive testing

This integration plan ensures that OWASP Dependency-Check is seamlessly integrated into the existing security scanner framework while maintaining consistency with established patterns and providing comprehensive dependency vulnerability scanning capabilities.