Plugin Development Guide

This guide explains how to create custom plugins for the Web3Sec Framework.

Plugin Types

The framework supports several types of plugins:

1. **Built-in Scanners**: Pattern-based vulnerability detection
2. **External Tool Plugins**: Integration with external security tools
3. **Template Plugins**: YAML-based vulnerability templates
4. **Custom Plugins**: User-defined scanning logic

Creating a Built-in Scanner Plugin

1. Basic Structure

```
from web3sec_framework.plugins.base_plugin import BasePlugin
from web3sec_framework.core.scanner_base import Finding, Severity
from typing import List

class MyCustomScanner(BasePlugin):
    def __init__(self):
        super().__init__()
        self.name = "my_custom_scanner"
        self.plugin_type = "builtin"
        self.version = "1.0.0"
        self.description = "My custom vulnerability scanner"
        self.supported_extensions = ['.sol', '.js']

    def get_name(self) -> str:
        return self.name

    def supports_file(self, file_path: str) -> bool:
        return any(file_path.lower().endswith(ext) for ext in self.suppor-
ted_extensions)

    def scan_file(self, file_path: str, content: str) -> List[Finding]:
        findings = []
        # Your scanning logic here
        return findings
```

2. Pattern-Based Detection

```
import re

def scan_file(self, file_path: str, content: str) -> List[Finding]:
    findings = []
    lines = content.split('\n')

    # Define vulnerability patterns
    patterns = {
        "dangerous_function": {
            "regex": r"dangerousFunction\s*\(",
            "severity": Severity.HIGH,
            "description": "Usage of dangerous function detected"
        }
    }

    for vuln_name, pattern_data in patterns.items():
        for line_num, line in enumerate(lines, 1):
            if re.search(pattern_data["regex"], line):
                finding = Finding(
                    filename=file_path,
                    line=line_num,
                    vuln_type=vuln_name,
                    severity=pattern_data["severity"].value,
                    description=pattern_data["description"],
                    code_snippet=self._extract_snippet(content, line_num),
                    recommendation="Avoid using dangerous functions",
                    category="Security"
                )
                findings.append(finding)

    return findings
```

Creating an External Tool Plugin

1. Inherit from ExternalToolPlugin

```python
from web3sec_framework.plugins.external_tool_plugin import ExternalToolPlugin
import subprocess
import json

class MyToolPlugin(ExternalToolPlugin):
    def __init__(self, config: dict):
        super().__init__()
        self.name = "mytool"
        self.tool_path = config.get('path', 'mytool')
        self.timeout = config.get('timeout', 60)

    def is_available(self) -> bool:
        try:
            result = subprocess.run([self.tool_path, '--version'],
                                    capture_output=True, timeout=10)
            return result.returncode == 0
        except:
            return False

    def scan_file(self, file_path: str, content: str) -> List[Finding]:
        # Create temporary file
        with tempfile.NamedTemporaryFile(mode='w', suffix='.sol', delete=False) as f:
            f.write(content)
            temp_path = f.name

        try:
            # Run external tool
            result = self.run_tool([self.tool_path, '--json', temp_path])

            # Parse results
            if result.stdout:
                data = json.loads(result.stdout)
                return self._parse_results(data, file_path)
        finally:
            os.unlink(temp_path)

        return []
```

2. Result Parsing

```
def _parse_results(self, data: dict, file_path: str) -> List[Finding]:
    findings = []

    for issue in data.get('issues', []):
        finding = Finding(
            filename=file_path,
            line=issue.get('line', 1),
            vuln_type=issue.get('type', 'Unknown'),
            severity=self._map_severity(issue.get('severity')),
            description=issue.get('message', ''),
            code_snippet=issue.get('code', ''),
            recommendation=self._get_recommendation(issue.get('type'))
        )
        findings.append(finding)

    return findings

def _map_severity(self, tool_severity: str) -> str:
    mapping = {
        'critical': 'critical',
        'high': 'high',
        'medium': 'medium',
        'low': 'low'
    }
    return mapping.get(tool_severity.lower(), 'medium')
```

Creating Template Plugins

1. YAML Template Structure

```yaml
id: my-custom-check
info:
  name: "My Custom Vulnerability Check"
  author: "Security Team"
  severity: high
  description: "Detects my custom vulnerability pattern"
  tags: ["custom", "security"]

file:
  extensions:
    - "sol"
    - "js"

matchers-condition: and
matchers:
  - type: regex
    regex:
      - "vulnerablePattern\\s*\\("
      - "anotherPattern\\s*="
    condition: or

  - type: word
    words:
      - "dangerous"
      - "unsafe"
    condition: or

extractors:
  - type: regex
    regex:
      - "function\\s+(\\w+)\\s*\\("
    group: 1
```

2. Template Plugin Implementation

Templates are automatically loaded by the `TemplatePlugin` class. Place your YAML files in:
- `templates/vulnerabilities/` (built-in)
- Custom directories specified in configuration

Plugin Registration

1. Built-in Plugins

Place your plugin file in:

```
plugins/builtin/my_scanner.py
```

Update `plugins/builtin/__init__.py` :

```
from .my_scanner import MyCustomScanner

__all__ = [
    # ... existing plugins
    "MyCustomScanner"
]
```

2. Custom Plugin Directories

Configure custom plugin directories in `.web3scanrc`:

```
{
  "plugins": {
    "custom_plugin_dirs": [
      "/path/to/my/plugins",
      "./custom_plugins"
    ]
  }
}
```

Best Practices

1. Error Handling

```
def scan_file(self, file_path: str, content: str) -> List[Finding]:
    findings = []

    try:
        # Your scanning logic
        pass
    except Exception as e:
        self.logger.error(f"Error scanning {file_path}: {e}")
        # Don't raise - return empty list to continue scanning

    return findings
```

2. Performance Optimization

```
def scan_file(self, file_path: str, content: str) -> List[Finding]:
    # Pre-compile regex patterns
    if not hasattr(self, '_compiled_patterns'):
        self._compiled_patterns = {
            name: re.compile(pattern['regex'])
            for name, pattern in self.patterns.items()
        }

    # Use compiled patterns for better performance
    for name, compiled_pattern in self._compiled_patterns.items():
        # ... scanning logic
```

3. Configuration Support

```python
class MyScanner(BasePlugin):
    def __init__(self, config: dict = None):
        super().__init__()
        self.config = config or {}

        # Load configuration
        self.enabled_checks = self.config.get('enabled_checks', [])
        self.severity_threshold = self.config.get('severity_threshold', 'medium')

    def get_config(self) -> dict:
        return {
            'enabled_checks': self.enabled_checks,
            'severity_threshold': self.severity_threshold,
            'supported_extensions': self.supported_extensions
        }
```

4. Testing

```python
import unittest
from pathlib import Path

class TestMyScanner(unittest.TestCase):
    def setUp(self):
        self.scanner = MyCustomScanner()

    def test_vulnerability_detection(self):
        test_code = """
        function vulnerable() {
            dangerousFunction();
        }
        """

        findings = self.scanner.scan_file("test.sol", test_code)
        self.assertEqual(len(findings), 1)
        self.assertEqual(findings[0].vuln_type, "dangerous_function")

    def test_no_false_positives(self):
        safe_code = """
        function safe() {
            safeFunction();
        }
        """

        findings = self.scanner.scan_file("test.sol", safe_code)
        self.assertEqual(len(findings), 0)
```

Plugin Metadata

Required Methods

Every plugin must implement:
- `get_name()` : Return plugin name
- `supports_file(file_path)` : Check if plugin supports file type
- `scan_file(file_path, content)` : Perform the actual scanning

Optional Methods

- `get_config()`: Return plugin configuration
- `validate()`: Validate plugin setup
- `get_metadata()`: Return plugin metadata

Plugin Information

```python
def get_metadata(self) -> dict:
    return {
        'name': self.name,
        'version': self.version,
        'description': self.description,
        'author': 'Your Name',
        'supported_extensions': self.supported_extensions,
        'plugin_type': self.plugin_type,
        'requires_external_tool': False,
        'configuration_schema': {
            'enabled_checks': 'list',
            'severity_threshold': 'string'
        }
    }
```

Advanced Features

1. Multi-file Analysis

```python
def scan_project(self, project_path: str) -> List[Finding]:
    """Analyze entire project for cross-file vulnerabilities."""
    findings = []

    # Collect all files
    files = self._discover_files(project_path)

    # Build project context
    context = self._build_project_context(files)

    # Analyze with context
    for file_path in files:
        content = file_path.read_text()
        file_findings = self._scan_with_context(file_path, content, context)
        findings.extend(file_findings)

    return findings
```

2. Incremental Scanning

```python
def scan_incremental(self, file_path: str, content: str, cache: dict) -> List[Finding]:
    """Scan with caching support."""
    content_hash = hashlib.md5(content.encode()).hexdigest()

    if cache.get(file_path) == content_hash:
        return []  # No changes, skip scan

    findings = self.scan_file(file_path, content)
    cache[file_path] = content_hash

    return findings
```

3. Custom Output Formatting

```python
def format_findings(self, findings: List[Finding]) -> dict:
    """Custom formatting for plugin-specific output."""
    return {
        'plugin': self.name,
        'version': self.version,
        'findings': [f.to_dict() for f in findings],
        'statistics': {
            'total': len(findings),
            'by_severity': self._count_by_severity(findings)
        }
    }
```

Debugging Plugins

1. Enable Debug Logging

```python
import logging

class MyScanner(BasePlugin):
    def __init__(self):
        super().__init__()
        self.logger = logging.getLogger(__name__)

    def scan_file(self, file_path: str, content: str) -> List[Finding]:
        self.logger.debug(f"Scanning {file_path}")

        findings = []
        # ... scanning logic

        self.logger.debug(f"Found {len(findings)} issues in {file_path}")
        return findings
```

2. Test with Sample Files

Create test files in `examples/` directory and run:

```
web3sec --target examples/test.sol --plugins my_custom_scanner --debug
```

3. Validate Plugin

```
# Test plugin loading
web3sec --list-plugins

# Get plugin information
web3sec --plugin-info my_custom_scanner

# Validate configuration
web3sec --validate-config
```

This guide provides a comprehensive overview of plugin development for the Web3Sec Framework. For more examples, see the built-in plugins in the `plugins/` directory.