J-Sentinel Rule Engine Architecture

Executive Summary

The J-Sentinel rule engine, a critical component of the Analysis & Processing Layer, detects OWASP Top 10 vulnerabilities and custom logic flaws in Java applications by querying pre-stored code graphs and taint flows in Neo4j. It processes codegraph.json (code structure) and taint_analysis.json (tainted paths) using a Python-based rule engine with a YAML-based Domain-Specific Language (DSL), translated into Cypher queries.

---

1. Introduction

Purpose: The rule engine evaluates rules to identify vulnerabilities (e.g., log injection, SQL injection) in Java code, supporting J-Sentinel's secure DevSecOps-integrated static analysis. It queries pre-stored codegraph.json and taint_analysis.json data in Neo4j, enabling deep path analysis.

Objectives:

- Define the rule engine's architecture and components.
- Describe how it queries Neo4j-stored data.
- Plan for future CFG integration.
- Provide recommendations for implementation and testing.

Scope: The development team's responsibility is to build and test the rule engine, assuming codegraph.json and taint_analysis.json are already stored in Neo4j by another team.

---

2. Neo4j Data Model

The rule engine queries codegraph.json and taint_analysis.json data stored in Neo4j, structured as a graph with nodes and relationships.

2.1 Graph Schema

- **Nodes**:
  - :FILE: Source file (e.g., test.java; id: 1, name: 'test.java').
  - :CLASS: Class (e.g., SimpleTest; id: 2).

- :METHOD: Method (e.g., run; id: 3, name: 'run', parameters: 1).
- :METHOD_CALL: Method call (e.g., logger.info; id: 5, scope: 'logger', name: 'info').
- :BINARY_EXPRESSION: Expression (e.g., "Processing input: " + userInput; id: 11, operator: '+').
- :SCAN: Scan metadata (e.g., scanId: 'fd01a841-ff0e-4f1e-9c9c-8e01fe973ed8').
- Others: :PARAMETER, :LOCAL_VARIABLE, :IF_STATEMENT, :STRING_LITERAL, etc.

- **Relationships**:
  - :CONTAINS: Links containers (e.g., :FILE → :CLASS).
  - :INVOKES: Method calls (e.g., :METHOD → :METHOD_CALL).
  - :DATA_FLOW: Taint paths (e.g., :METHOD_CALL {id: 6} → :METHOD_CALL {id: 5}, with severity: 'HIGH').
  - :PART_OF_SCAN: Links nodes to :SCAN for scan-specific queries.
- **Properties**:
  - Nodes: id, name, type, scope, etc., from JSON.
  - :DATA_FLOW: severity, vulnerability, pathNodes (list of IDs).

2.2 Storage Format

- **codegraph.json**:
  - Nodes: Each nodes entry is a Neo4j node (e.g., :METHOD_CALL {id: 5, name: 'info'}).
  - Relationships: edges create relationships (e.g., (:METHOD {id: 3})-[:INVOKES]->(:METHOD_CALL {id: 5})).
  - Issues (e.g., potentialLogInjections) are node properties (e.g., potentialLogInjection: 'Potential log injection').
- **taint_analysis.json**:
  - Nodes: Sources/sinks are :METHOD_CALL nodes (e.g., id: 6, name: 'readLine').
  - Relationships: taintedPaths create :DATA_FLOW (e.g., (:METHOD_CALL {id: 6})-[:DATA_FLOW {severity: 'HIGH'}]->(:METHOD_CALL {id: 5})).
  - Scan: Nodes link to :SCAN via :PART_OF_SCAN.

2.3 Querying Data

- **codegraph.json**:
  - Detects log injections (e.g., methodCallId: 5) by querying :BINARY_EXPRESSION with operator: '+' linked to :METHOD_CALL {scope: 'logger'}.

Example Cypher:

MATCH (method:METHOD)-[:CONTAINS_EXPRESSION]->(expr:BINARY_EXPRESSION {operator: '+'})

MATCH (method)-[:INVOKES]->(call:METHOD_CALL {scope: 'logger', name: 'info'})

RETURN call, expr, call.potentialLogInjection

- **taint_analysis.json**:
  - Validates taint paths (e.g., readLine → logger.info; id: 6 → 5).

Example Cypher:
MATCH (source:METHOD_CALL {name: 'readLine'})-[:DATA_FLOW]->(sink:METHOD_CALL {scope: 'logger'})

RETURN source, sink, properties(r).severity

---

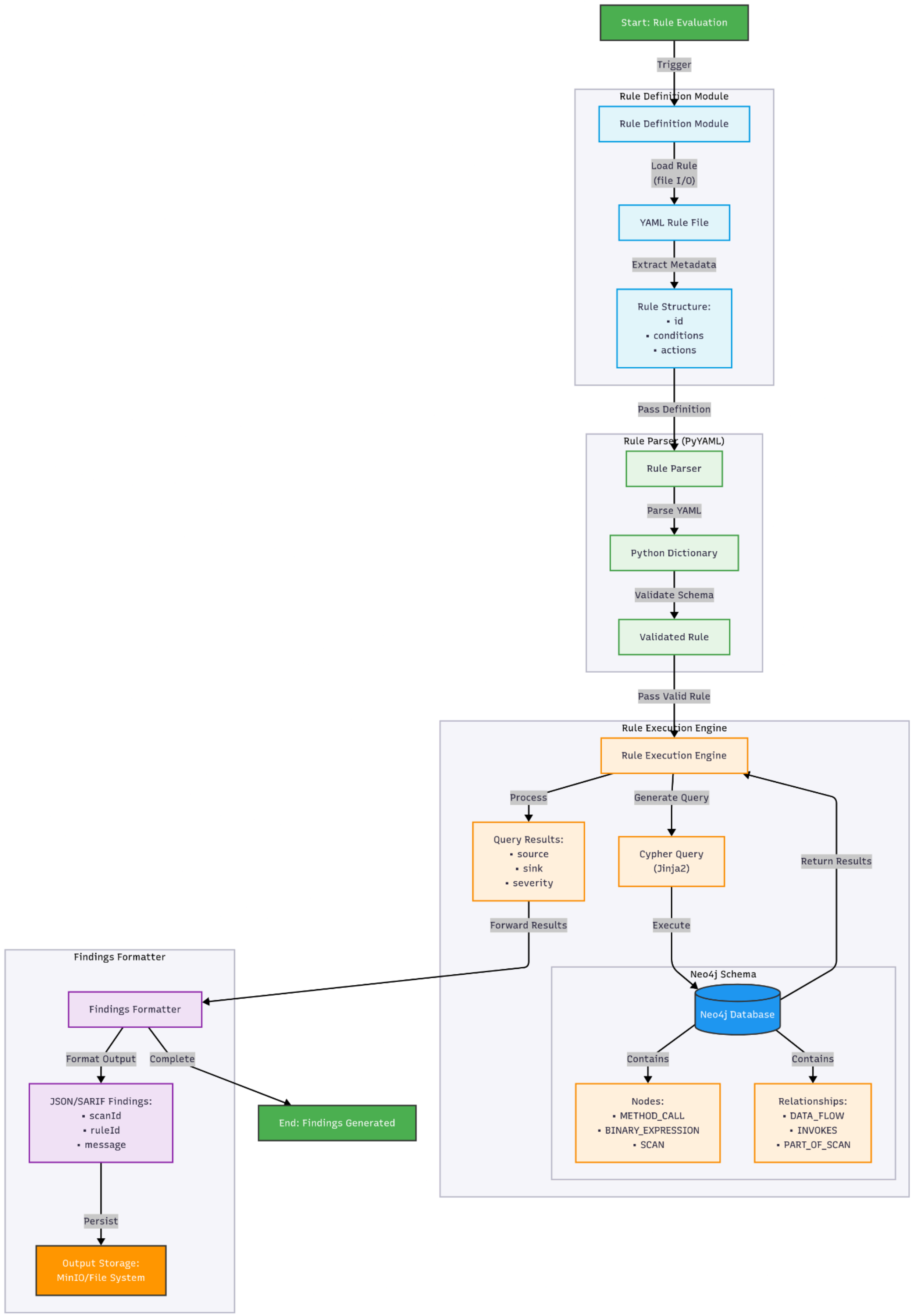
3. Rule Engine Architecture

The rule engine is a Python-based module that parses YAML rules, generates Cypher queries, and executes them against Neo4j. It integrates with J-Sentinel's Java-based CLI (scanner.java, analyse.java) via shared Neo4j data.

3.1 Tech Stack

- **Python 3.11**: Core language for rule parsing, query generation, and execution.
- **PyYAML 6.6**: Parses YAML rules.
- **Neo4j 5.0 with Cypher**: Queries stored data.
- **neo4j 5.16 (Python driver)**: Executes Cypher queries.
- **Jinja2 3.1.4**: Generates Cypher queries.
- **pytest 8.3.3**: Tests rule engine components.

This diagram illustrates the end-to-end flow of the J-Sentinel rule engine, from YAML rule definition to vulnerability findings. It shows how rules are parsed, executed as Cypher queries against Neo4j, and formatted into JSON/SARIF reports. Arrows explicitly label each data transformation and handoff between components.



Start: Rule Evaluation
Trigger
Rule Definition Module
Rule Definition Module
Load Rule (file I/O)
YAML Rule File
Extract Metadata
Rule Structure:
• id
• conditions
• actions
Pass Definition
Rule Parser (PyYAML)
Rule Parser
Parse YAML
Python Dictionary
Validate Schema
Validated Rule
Pass Valid Rule
Rule Execution Engine
Rule Execution Engine
Process
Generate Query
Query Results:
• source
• sink
• severity
Cypher Query (Jinja2)
Return Results
Forward Results
Execute
Neo4j Schema
Neo4j Database
Contains
Contains
Nodes:
• METHOD_CALL
• BINARY_EXPRESSION
• SCAN
Relationships:
• DATA_FLOW
• INVOKES
• PART_OF_SCAN
Findings Formatter
Findings Formatter
Format Output
Complete
JSON/SARIF Findings:
• scanId
• ruleId
• message
End: Findings Generated
Persist
Output Storage: MinIO/File System

3.2 Components

3.2.1 Rule Definition Module

- **Function**: Defines rules using a YAML-based DSL.
- **Details**:
    - Rules specify vulnerability conditions (e.g., taint from readLine to logger.info).

Example rule (log injection):

```
rule:

  id: LOG_INJECTION_001

  name: Detect Log Injection

  severity: HIGH

  conditions:

    - type: TAINT_PATH

      source: { type: "METHOD_CALL", name: "readLine" }

      sink: { type: "METHOD_CALL", scope: "logger", name: ["info", "severe"] }

  actions:

    - report:

        message: "Potential log injection: {sink.name}"

        suggested_fix: "Sanitize input with replaceAll('[\\n\\r]', '')"
```

-
    - Stored in a file system or MinIO, accessed by the rule engine.

3.2.2 Rule Parser

- **Function**: Parses YAML rules into Python objects for query generation.
- **Details**:
    - Uses PyYAML to load YAML files.
    - Validates rule structure (e.g., presence of id, conditions).

Example:

```
import yaml
```

```python
def parse_rule(yaml_file: str) -> dict:

    with open(yaml_file, 'r') as f:

        rule = yaml.safe_load(f)

    if not rule.get('id') or not rule.get('conditions'):

        raise ValueError("Invalid rule")

    return rule
```

3.2.3 Rule Execution Engine

- **Function**: Generates and executes Cypher queries against Neo4j.
- **Details**:
  - Uses Jinja2 to generate Cypher queries from rules.
  - Executes queries via neo4j Python driver.
  - Workflow:
    1. Parse YAML rule.
    2. Generate Cypher query (e.g., match `:DATA_FLOW` from `readLine` to `logger.info`).
    3. Query Neo4j for scan-specific data (via `:SCAN`).
    4. Collect findings (node IDs, paths, severity).

Example Cypher (generated):

```
MATCH path = (source:METHOD_CALL {name:
'readLine'})-[:DATA_FLOW*]->(sink:METHOD_CALL)

WHERE sink.scope = 'logger' AND sink.name IN ['info', 'severe']

RETURN source, sink, path, 'Potential log injection' AS message
```

3.2.4 Findings Formatter

- **Function**: Formats query results as JSON/SARIF.
- **Details**:
  - Outputs `scanId`, rule details, and findings.

Example:

```
{
```

"scanId": "fd01a841-ff0e-4f1e-9c9c-8e01fe973ed8",

"findings": [

{

"ruleId": "LOG_INJECTION_001",

"severity": "HIGH",

"message": "Potential log injection: info",

"source": { "id": 6, "name": "readLine" },

"sink": { "id": 5, "name": "info" },

"suggestedFix": "Sanitize input with replaceAll('[\\n\\r]', '')"

}

]

}

- Findings are saved to MinIO or a file system for downstream use.

---

4. Future CFG Integration

- **Control Flow Graph (CFG)**:
  - **Nodes**: Statements (e.g., conditionals; :CFG_NODE).
  - **Relationships**: :NEXT, :BRANCH.
  - **Use**: Verify sanitization before sinks (e.g., check replaceAll before logger.info).
- **Implementation**:
  - Add :CFG_NODE labels to Neo4j.
  - Link to :METHOD via :HAS_CFG.

Update rules to traverse :NEXT paths:

```
MATCH (sanitize:CFG_NODE {name: 'replaceAll'})-[:NEXT*]->(sink:CFG_NODE {name: 'logger.info'})

RETURN sanitize, sink
```

---

5. Recommendations

1. **Implement Core Components**:
   - Develop the rule parser using PyYAML.
   - Build the query generator with Jinja2.
   - Integrate neo4j driver for query execution.
2. **Test Thoroughly**:
   - Write pytest tests for rule parsing, query generation, and execution.
   - Validate against taint_analysis.json findings (e.g., log injection, id: 6 → 5).

Example test:

```
def test_parse_rule():

   rule = parse_rule("tests/log_injection.yaml")

   assert rule["id"] == "LOG_INJECTION_001"
```

   -

3. **Prepare for CFG**:
   - Design rules to query :CFG_NODE and :NEXT relationships.
   - Test with sample CFG data when available.
4. **Integration**:
   - Ensure the rule engine reads YAML rules from a shared directory or MinIO.
   - Coordinate with the data loading team to verify Neo4j schema compatibility.

---

6. Conclusion

The J-Sentinel rule engine, implemented in Python, leverages Neo4j to detect vulnerabilities by querying pre-stored codegraph.json and taint_analysis.json data. Its modular architecture (parser, execution engine, formatter) and YAML-based DSL enable flexible rule definition and execution. Future CFG integration will enhance precision. The development team should focus on implementing the parser and query generator, testing against Neo4j data, and preparing for CFG support.