/sarif-parsing

Source: `~/.claude/skills/tob-static-analysis/skills/sarif-parsing/SKILL.md`

---

name: sarif-parsing description: Parse, analyze, and process SARIF (Static Analysis Results Interchange Format) files. Use when reading security scan results, aggregating findings from multiple tools, deduplicating alerts, extracting specific vulnerabilities, or integrating SARIF data into CI/CD pipelines. allowed-tools:

- Bash
- Read
- Glob
- Grep

SARIF Parsing Best Practices

You are a SARIF parsing expert. Your role is to help users effectively read, analyze, and process SARIF files from static analysis tools.

When to Use

Use this skill when:

- Reading or interpreting static analysis scan results in SARIF format
- Aggregating findings from multiple security tools
- Deduplicating or filtering security alerts
- Extracting specific vulnerabilities from SARIF files
- Integrating SARIF data into CI/CD pipelines
- Converting SARIF output to other formats

When NOT to Use

Do NOT use this skill for:

- Running static analysis scans (use CodeQL or Semgrep skills instead)
- Writing CodeQL or Semgrep rules (use their respective skills)
- Analyzing source code directly (SARIF is for processing existing scan results)
- Triaging findings without SARIF input (use variant-analysis or audit skills)

SARIF Structure Overview

SARIF 2.1.0 is the current OASIS standard. Every SARIF file has this hierarchical structure:

```
sarifLog
├── version: "2.1.0"
├── $schema: (optional, enables IDE validation)
└── runs[] (array of analysis runs)
    ├── tool
    │   ├── driver
    │   │   ├── name (required)
    │   │   ├── version
    │   │   └── rules[] (rule definitions)
    │   └── extensions[] (plugins)
    ├── results[] (findings)
    │   ├── ruleId
    │   ├── level (error/warning/note)
    │   ├── message.text
    │   ├── locations[]
    │   │   └── physicalLocation
    │   │       ├── artifactLocation.uri
    │   │       └── region (startLine, startColumn, etc.)
    │   ├── fingerprints{}
    │   └── partialFingerprints{}
    └── artifacts[] (scanned files metadata)
```

Why Fingerprinting Matters

Without stable fingerprints, you can't track findings across runs:

- **Baseline comparison**: "Is this a new finding or did we see it before?"
- **Regression detection**: "Did this PR introduce new vulnerabilities?"
- **Suppression**: "Ignore this known false positive in future runs"

Tools report different paths (`/path/to/project/` vs `/github/workspace/`), so path-based matching fails. Fingerprints hash the *content* (code snippet, rule ID, relative location) to create stable identifiers regardless of environment.

Tool Selection Guide

Use Case	Tool	Installation
Quick CLI queries	jq	`brew install jq` / `apt install jq`
Python scripting (simple)	pysarif	`pip install pysarif`
Python scripting (advanced)	sarif-tools	`pip install sarif-tools`
.NET applications	SARIF SDK	NuGet package
JavaScript/Node.js	sarif-js	npm package
Go applications	garif	`go get github.com/chavacava/garif`
Validation	SARIF Validator	sarifweb.azurewebsites.net

Strategy 1: Quick Analysis with jq

For rapid exploration and one-off queries:

```
# Pretty print the file
jq '.' results.sarif

# Count total findings
jq '[.runs[].results[]] | length' results.sarif

# List all rule IDs triggered
jq '[.runs[].results[].ruleId] | unique' results.sarif

# Extract errors only
jq '.runs[].results[] | select(.level == "error")' results.sarif

# Get findings with file locations
```

```
jq '.runs[].results[] | {
  rule: .ruleId,
  message: .message.text,
  file: .locations[0].physicalLocation.artifactLocation.uri,
  line: .locations[0].physicalLocation.region.startLine
}' results.sarif

# Filter by severity and get count per rule
jq '[.runs[].results[] | select(.level == "error")] | group_by(.ruleId) | map({rule:
.[0].ruleId, count: length})' results.sarif

# Extract findings for a specific file
jq --arg file "src/auth.py" '.runs[].results[] |
select(.locations[].physicalLocation.artifactLocation.uri | contains($file))' results.sarif
```

Strategy 2: Python with pysarif

For programmatic access with full object model:

```
from pysarif import load_from_file, save_to_file

# Load SARIF file
sarif = load_from_file("results.sarif")

# Iterate through runs and results
for run in sarif.runs:
    tool_name = run.tool.driver.name
    print(f"Tool: {tool_name}")

    for result in run.results:
        print(f"  [{result.level}] {result.rule_id}: {result.message.text}")

        if result.locations:
            loc = result.locations[0].physical_location
            if loc and loc.artifact_location:
                print(f"    File: {loc.artifact_location.uri}")
                if loc.region:
                    print(f"    Line: {loc.region.start_line}")
```

```
# Save modified SARIF
save_to_file(sarif, "modified.sarif")
```

Strategy 3: Python with sarif-tools

For aggregation, reporting, and CI/CD integration:

```
from sarif import loader

# Load single file
sarif_data = loader.load_sarif_file("results.sarif")

# Or load multiple files
sarif_set = loader.load_sarif_files(["tool1.sarif", "tool2.sarif"])

# Get summary report
report = sarif_data.get_report()

# Get histogram by severity
errors = report.get_issue_type_histogram_for_severity("error")
warnings = report.get_issue_type_histogram_for_severity("warning")

# Filter results
high_severity = [r for r in sarif_data.get_results()
                 if r.get("level") == "error"]
```

sarif-tools CLI commands:

```
# Summary of findings
sarif summary results.sarif

# List all results with details
sarif ls results.sarif

# Get results by severity
sarif ls --level error results.sarif
```

```
# Diff two SARIF files (find new/fixed issues)
sarif diff baseline.sarif current.sarif

# Convert to other formats
sarif csv results.sarif > results.csv
sarif html results.sarif > report.html
```

Strategy 4: Aggregating Multiple SARIF Files

When combining results from multiple tools:

```
import json
from pathlib import Path

def aggregate_sarif_files(sarif_paths: list[str]) -> dict:
    """Combine multiple SARIF files into one."""
    aggregated = {
        "version": "2.1.0",
        "$schema": "https://json.schemastore.org/sarif-2.1.0.json",
        "runs": []
    }

    for path in sarif_paths:
        with open(path) as f:
            sarif = json.load(f)
            aggregated["runs"].extend(sarif.get("runs", []))

    return aggregated

def deduplicate_results(sarif: dict) -> dict:
    """Remove duplicate findings based on fingerprints."""
    seen_fingerprints = set()

    for run in sarif["runs"]:
        unique_results = []
        for result in run.get("results", []):
```

```
            # Use partialFingerprints or create key from location
            fp = None
            if result.get("partialFingerprints"):
                fp = tuple(sorted(result["partialFingerprints"].items()))
            elif result.get("fingerprints"):
                fp = tuple(sorted(result["fingerprints"].items()))
            else:
                # Fallback: create fingerprint from rule + location
                loc = result.get("locations", [{}])[0]
                phys = loc.get("physicalLocation", {})
                fp = (
                    result.get("ruleId"),
                    phys.get("artifactLocation", {}).get("uri"),
                    phys.get("region", {}).get("startLine")
                )

            if fp not in seen_fingerprints:
                seen_fingerprints.add(fp)
                unique_results.append(result)

        run["results"] = unique_results

    return sarif
```

Strategy 5: Extracting Actionable Data

```
import json
from dataclasses import dataclass
from typing import Optional

@dataclass
class Finding:
    rule_id: str
    level: str
    message: str
    file_path: Optional[str]
    start_line: Optional[int]
    end_line: Optional[int]
```

```
    fingerprint: Optional[str]

def extract_findings(sarif_path: str) -> list[Finding]:
    """Extract structured findings from SARIF file."""
    with open(sarif_path) as f:
        sarif = json.load(f)

    findings = []
    for run in sarif.get("runs", []):
        for result in run.get("results", []):
            loc = result.get("locations", [{}])[0]
            phys = loc.get("physicalLocation", {})
            region = phys.get("region", {})

            findings.append(Finding(
                rule_id=result.get("ruleId", "unknown"),
                level=result.get("level", "warning"),
                message=result.get("message", {}).get("text", ""),
                file_path=phys.get("artifactLocation", {}).get("uri"),
                start_line=region.get("startLine"),
                end_line=region.get("endLine"),
                fingerprint=next(iter(result.get("partialFingerprints", {}).values()), None)
            ))

    return findings

# Filter and prioritize
def prioritize_findings(findings: list[Finding]) -> list[Finding]:
    """Sort findings by severity."""
    severity_order = {"error": 0, "warning": 1, "note": 2, "none": 3}
    return sorted(findings, key=lambda f: severity_order.get(f.level, 99))
```

Common Pitfalls and Solutions

1. Path Normalization Issues

Different tools report paths differently (absolute, relative, URI-encoded):

```
from urllib.parse import unquote
from pathlib import Path

def normalize_path(uri: str, base_path: str = "") -> str:
    """Normalize SARIF artifact URI to consistent path."""
    # Remove file:// prefix if present
    if uri.startswith("file://"):
        uri = uri[7:]

    # URL decode
    uri = unquote(uri)

    # Handle relative paths
    if not Path(uri).is_absolute() and base_path:
        uri = str(Path(base_path) / uri)

    # Normalize separators
    return str(Path(uri))
```

2. Fingerprint Mismatch Across Runs

Fingerprints may not match if:

- File paths differ between environments
- Tool versions changed fingerprinting algorithm
- Code was reformatted (changing line numbers)

Solution: Use multiple fingerprint strategies:

```
def compute_stable_fingerprint(result: dict, file_content: str = None) -> str:
    """Compute environment-independent fingerprint."""
    import hashlib

    components = [
        result.get("ruleId", ""),
        result.get("message", {}).get("text", "")[:100],  # First 100 chars
    ]

    # Add code snippet if available
    if file_content and result.get("locations"):
```

```
        region = result["locations"][0].get("physicalLocation", {}).get("region", {})
        if region.get("startLine"):
            lines = file_content.split("\n")
            line_idx = region["startLine"] - 1
            if 0 <= line_idx < len(lines):
                # Normalize whitespace
                components.append(lines[line_idx].strip())

    return hashlib.sha256("".join(components).encode()).hexdigest()[:16]
```

3. Missing or Incomplete Data

SARIF allows many optional fields. Always use defensive access:

```
def safe_get_location(result: dict) -> tuple[str, int]:
    """Safely extract file and line from result."""
    try:
        loc = result.get("locations", [{}])[0]
        phys = loc.get("physicalLocation", {})
        file_path = phys.get("artifactLocation", {}).get("uri", "unknown")
        line = phys.get("region", {}).get("startLine", 0)
        return file_path, line
    except (IndexError, KeyError, TypeError):
        return "unknown", 0
```

4. Large File Performance

For very large SARIF files (100MB+):

```
import ijson  # pip install ijson


def stream_results(sarif_path: str):
    """Stream results without loading entire file."""
    with open(sarif_path, "rb") as f:
        # Stream through results arrays
        for result in ijson.items(f, "runs.item.results.item"):
            yield result
```

5. Schema Validation

Validate before processing to catch malformed files:

```
# Using ajv-cli
npm install -g ajv-cli
ajv validate -s sarif-schema-2.1.0.json -d results.sarif

# Using Python jsonschema
pip install jsonschema
```

```
from jsonschema import validate, ValidationError
import json

def validate_sarif(sarif_path: str, schema_path: str) -> bool:
    """Validate SARIF file against schema."""
    with open(sarif_path) as f:
        sarif = json.load(f)
    with open(schema_path) as f:
        schema = json.load(f)

    try:
        validate(sarif, schema)
        return True
    except ValidationError as e:
        print(f"Validation error: {e.message}")
        return False
```

CI/CD Integration Patterns

GitHub Actions

```
- name: Upload SARIF
  uses: github/codeql-action/upload-sarif@v3
  with:
    sarif_file: results.sarif

- name: Check for high severity
  run: |
    HIGH_COUNT=$(jq '[.runs[].results[] | select(.level == "error")] | length' results.sarif)
```

```
    if [ "$HIGH_COUNT" -gt 0 ]; then
      echo "Found $HIGH_COUNT high severity issues"
      exit 1
    fi
```

Fail on New Issues

```
from sarif import loader

def check_for_regressions(baseline: str, current: str) -> int:
    """Return count of new issues not in baseline."""
    baseline_data = loader.load_sarif_file(baseline)
    current_data = loader.load_sarif_file(current)

    baseline_fps = {get_fingerprint(r) for r in baseline_data.get_results()}
    new_issues = [r for r in current_data.get_results()
                  if get_fingerprint(r) not in baseline_fps]

    return len(new_issues)
```

Key Principles

1. **Validate first**: Check SARIF structure before processing
2. **Handle optionals**: Many fields are optional; use defensive access
3. **Normalize paths**: Tools report paths differently; normalize early
4. **Fingerprint wisely**: Combine multiple strategies for stable deduplication
5. **Stream large files**: Use ijson or similar for 100MB+ files
6. **Aggregate thoughtfully**: Preserve tool metadata when combining files

Skill Resources

For ready-to-use query templates, see {baseDir}/resources/jq-queries.md:

- 40+ jq queries for common SARIF operations
- Severity filtering, rule extraction, aggregation patterns

For Python utilities, see {baseDir}/resources/sarif_helpers.py:

- `normalize_path()` - Handle tool-specific path formats
- `compute_fingerprint()` - Stable fingerprinting ignoring paths
- `deduplicate_results()` - Remove duplicates across runs

Reference Links

- OASIS SARIF 2.1.0 Specification
- Microsoft SARIF Tutorials
- SARIF SDK (.NET)
- sarif-tools (Python)
- pysarif (Python)
- GitHub SARIF Support
- SARIF Validator

Revision #5
Created 2026-02-18 08:40:09 UTC by John
Updated 2026-06-21 20:01:17 UTC by John