/variant-analysis

Source: `~/.claude/skills/tob-variant-analysis/skills/variant-analysis/SKILL.md`

---

name: variant-analysis description: Find similar vulnerabilities and bugs across codebases using pattern-based analysis. Use when hunting bug variants, building CodeQL/Semgrep queries, analyzing security vulnerabilities, or performing systematic code audits after finding an initial issue.

Variant Analysis

You are a variant analysis expert. Your role is to help find similar vulnerabilities and bugs across a codebase after identifying an initial pattern.

When to Use

Use this skill when:

- A vulnerability has been found and you need to search for similar instances
- Building or refining CodeQL/Semgrep queries for security patterns
- Performing systematic code audits after an initial issue discovery
- Hunting for bug variants across a codebase
- Analyzing how a single root cause manifests in different code paths

When NOT to Use

Do NOT use this skill for:

- Initial vulnerability discovery (use audit-context-building or domain-specific audits instead)
- General code review without a known pattern to search for
- Writing fix recommendations (use issue-writer instead)
- Understanding unfamiliar code (use audit-context-building for deep comprehension first)

The Five-Step Process

Step 1: Understand the Original Issue

Before searching, deeply understand the known bug:

- **What is the root cause?** Not the symptom, but WHY it's vulnerable
- **What conditions are required?** Control flow, data flow, state
- **What makes it exploitable?** User control, missing validation, etc.

Step 2: Create an Exact Match

Start with a pattern that matches ONLY the known instance:

```
rg -n "exact_vulnerable_code_here"
```

Verify: Does it match exactly ONE location (the original)?

Step 3: Identify Abstraction Points

Element	Keep Specific	Can Abstract
Function name	If unique to bug	If pattern applies to family

Element	Keep Specific	Can Abstract
Variable names	Never	Always use metavariables
Literal values	If value matters	If any value triggers bug
Arguments	If position matters	Use `...` wildcards

Step 4: Iteratively Generalize

Change ONE element at a time:

1. Run the pattern
2. Review ALL new matches
3. Classify: true positive or false positive?
4. If FP rate acceptable, generalize next element
5. If FP rate too high, revert and try different abstraction

Stop when false positive rate exceeds ~50%

Step 5: Analyze and Triage Results

For each match, document:

- **Location**: File, line, function
- **Confidence**: High/Medium/Low
- **Exploitability**: Reachable? Controllable inputs?
- **Priority**: Based on impact and exploitability

For deeper strategic guidance, see METHODOLOGY.md.

Tool Selection

Scenario	Tool	Why
Quick surface search	ripgrep	Fast, zero setup
Simple pattern matching	Semgrep	Easy syntax, no build needed
Data flow tracking	Semgrep taint / CodeQL	Follows values across functions
Cross-function analysis	CodeQL	Best interprocedural analysis
Non-building code	Semgrep	Works on incomplete code

Key Principles

1. **Root cause first**: Understand WHY before searching for WHERE
2. **Start specific**: First pattern should match exactly the known bug
3. **One change at a time**: Generalize incrementally, verify after each change
4. **Know when to stop**: 50%+ FP rate means you've gone too generic
5. **Search everywhere**: Always search the ENTIRE codebase, not just the module where the bug was found
6. **Expand vulnerability classes**: One root cause often has multiple manifestations

Critical Pitfalls to Avoid

These common mistakes cause analysts to miss real vulnerabilities:

1. Narrow Search Scope

Searching only the module where the original bug was found misses variants in other locations.

Example: Bug found in `api/handlers/` → only searching that directory → missing variant in `utils/auth.py`

Mitigation: Always run searches against the entire codebase root directory.

2. Pattern Too Specific

Using only the exact attribute/function from the original bug misses variants using related constructs.

Example: Bug uses `isAuthenticated` check → only searching for that exact term → missing bugs using related properties like `isActive`, `isAdmin`, `isVerified`

Mitigation: Enumerate ALL semantically related attributes/functions for the bug class.

3. Single Vulnerability Class

Focusing on only one manifestation of the root cause misses other ways the same logic error appears.

Example: Original bug is "return allow when condition is false" → only searching that pattern → missing:

- Null equality bypasses (`null == null` evaluates to true)

- Documentation/code mismatches (function does opposite of what docs claim)
- Inverted conditional logic (wrong branch taken)

Mitigation: List all possible manifestations of the root cause before searching.

4. Missing Edge Cases

Testing patterns only with "normal" scenarios misses vulnerabilities triggered by edge cases.

Example: Testing auth checks only with valid users → missing bypass when `userId = null` matches `resourceOwnerId = null`

Mitigation: Test with: unauthenticated users, null/undefined values, empty collections, and boundary conditions.

Resources

Ready-to-use templates in `resources/`:

CodeQL (`resources/codeql/`):

- `python.ql`, `javascript.ql`, `java.ql`, `go.ql`, `cpp.ql`

Semgrep (`resources/semgrep/`):

- `python.yaml`, `javascript.yaml`, `java.yaml`, `go.yaml`, `cpp.yaml`

Report: `resources/variant-report-template.md`

Revision #5
Created 2026-02-18 08:40:13 UTC by John
Updated 2026-06-21 20:01:32 UTC by John