One Thousand and One AI-Prevented CVEs

Vibe Coding a Whole New Supply Chain Defense

Brandon Wu

OWASP LA 2026

The Scale of Supply Chain Attacks

Software supply chain attacks have become impossible to ignore.

- Nearly **50,000** estimated supply chain vulnerabilities in 2025
- Up from **40,000** in 2024 — an increase of **20%**
- High-profile incidents: **Shai-Hulud**, **React2shell**, **Langflow**

For companies building application security tools, this raises a practical question:

What does it take to build a mature supply chain security product?

Detection Is Not Enough

At the core of any vulnerability detection product is the detection engine.

But detection capability alone is not enough.

More than anything else, a mature software supply chain solution must be able to **keep up with the incoming flood** of vulnerabilities.

The Status Quo: Meet Max

Max is a security researcher. His morning routine:

1. Check the queue — tickets auto-filed overnight for the latest CVEs
2. For each CVE, write a Semgrep rule that detects uses of the vulnerable functions, constrained to the vulnerable versions
3. Read the vulnerability description and the commit that fixed the issue
4. Open the repository, click through the source code

For Semgrep Supply Chain, *each CVE* necessitates writing a rule.

Max's Problem

Max has never seen this package before.

- He searches GitHub to see how people actually use the library
- The CVE concerns a single **private utility function**
- That function is consumed widely across the package
- He's not sure if *all* consumers of the package are vulnerable
- After ∼30 minutes of clicking through source, he makes his best guess

He sighs in satisfaction. Then checks his queue.

Twenty more vulnerabilities to go.

Why This Doesn't Scale

This workflow becomes more untenable as:

- The number of CVEs being filed **grows**
- The surface of languages and frameworks that must be supported **expands**
- Each rule costs expensive human resources

We need an approach that:

- **Scales** with the growing volume of supply chain vulnerabilities
- **Costs less** in terms of researcher time

Like any good programmers, we've identified a workflow, and we're itching to automate it.

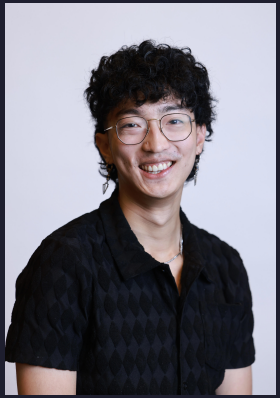
Me!

My name is Brandon Wu, and I am a program analysis engineer working at Semgrep.

I have been working at Semgrep for four years now, and I was educated in computer science at Carnegie Mellon University, where I previously lectured on the subject of functional programming.

Semgrep is a software security startup and application security platform that helps developers find and fix security vulnerabilities in their code, at minimal friction to their workflow.

Mission To profoundly improve software security.



@onefiftyman

LinkedIn

The Lifecycle of a Vulnerability

A vulnerability starts and ends its life in the same place — **in the code**.

Introducing Commit	The commit that introduces insecure behavior — either as originally written, or when some functionality is later modified
Disclosure	The vulnerability is found and divulged; the CVE system assigns a number
Patch Commit	Changes counter to the introducing commit fix the issue
Vulnerable Range	All versions released between the introducing and patch commits

SCA's Job

When a package with a CVE is used by a consumer, it is the job of Semgrep Supply Chain and other SCA tools to determine if the consumer is **potentially affected**.

Depending on the nature of the library and the vulnerability, this can be complex.

An important distinction: *how* does the vulnerability present itself?

Reachable vs. Upgrade-Only

Reachable

- We can write a rule isolating only the affected functions
- E.g., an object's `save()` method has a path traversal vuln
- A library's `foo()` is vulnerable if called with sensitive inputs
- Usually a *subset* of the API is vulnerable

Upgrade-Only

- The only realistic remediation is to upgrade the package entirely
- Could take significant effort for the user
- No discrete vulnerable entry point to match

```
rules:
  - id: my_sample_rule
    pattern: |
      library.save(...)
```

Design Goals

We want a better method for writing Semgrep Supply Chain rules that cuts out the manual work. These things should be true:

1. **Autonomous** — largely or completely free of human intervention
2. **Discriminating** — accurately distinguish when an upgrade-only or reachable rule is required
3. **Auditable** — easily verifiable for correctness
4. **Precise** — accurately determine which parts of a library's API are vulnerable to a CVE

Determinism vs. Agentic Workflows

We have a problem statement: *automatically writing Semgrep rules from CVEs*.

The easiest solution? Throw Claude Code at it.

But our tool was developed in 2024, before agentic coding assistants existed. Three useful lessons for 2026:

1. **Non-agentic code is deliberate code.** Shelling out to an agent is easy, but results in a less transparent structure that may accomplish tasks suboptimally.
2. **Agentic workflows are unpredictable.** Giving an agent freedom to figure out a solution opens you to inconsistent results.
3. **Structured code is easier to test, reason about, and improve upon.** When the entire rule-writing process is delegated to a black-box, it becomes difficult to improve specific parts of the system.

Let's talk about a workflow that could very easily be agentic today, and how we brought it down to earth as **(mostly) deterministic code**.

A Difference in Zeitgeist

But critically, one other thing aside from agentic coding assistants has changed since 2024.

We had Chappell Roan, Sabrina Carpenter, and the Eras Tour. Real ones will remember, however, the defining moment of the summer:

BRAT: Better RAT

The idea of a tool for helping researchers write rules is not new.

RAT (Rule Authorship Tool) — automated the boilerplate of creating a rule's schema, but not the semantic content (what it matches). Could save minutes per rule, but left room for automation.

BRAT (Better RAT) — also automatically determines the patterns the rule should match. Three main steps:

Step 1: Research

The "research" phase uses LLMs and traditional scraping to determine all we can about a given package or CVE:

- Scrape the **README** of the library
- Scrape the **online API documentation** of the package
- Search **GitHub** for real-world usage patterns
- Build a **codegraph**: determine which functions are transitively affected by the vulnerability's patch commit

Step 2: Test Generation

For the functions identified during research:

- Generate a "test file" of usages of the afflicted functions
- Each function is treated **independently**
- Uses the other artifacts from the research phase (README, API docs, real-world examples) for context
- Composable: if a subcomponent fails, we can rerun it by itself

Step 3: Rule Generation

Using the test file from the previous step:

- Generate Semgrep patterns that can match the test code
- Because we've done bookkeeping on where the function invocations are, we can **mechanically verify** whether rule generation succeeds
- Track coverage: e.g., matched 3 out of 5 target functions
- Stitch all successful patterns into a single rule

The philosophy: pure, simple, verifiable parts overall make a stronger, more stable product. Actions that can be done deterministically *are*, with only small sprinklings of nondeterminism for tasks that would be annoying to implement.

Standalone or Library?

Not all CVEs are made the same.

Libraries Imported and called in first-party code. E.g., `left-pad` — a CVE would affect any code that imports and uses that solitary function.

Standalone Installed as CLI tools, not imported in code. E.g., `ffmpeg` — invoked totally differently than a library.

Frameworks E.g., Django, Flask — don't have discrete entry points like library code, but affect overall system security.

In the latter two cases, SCA rules cannot typically be written. These tend to become **upgrade-only** rules.

BRAT Strategies

An important part of automating rule writing is accurately determining whether a reachable rule is even possible.

BRAT employs **"strategies"**: conditionally branching behavior early on based on whether a package is classified as:

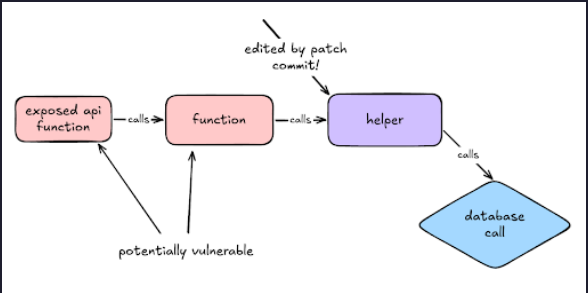
- A **standalone application** $\rightarrow$ upgrade-only path
- A **library** used in first-party code $\rightarrow$ full codegraph analysis $+$ reachable rule generation

What's in a Codegraph?

Key insight: Vulnerable code is made vulnerable by *calling* vulnerable code.

Remember Max's CVE? A deeply nested **private utility function** was changed by the patching commit. This implies:

- The vulnerable behavior has something to do with that function
- Any *caller* of that function may potentially be compromised
- By traversing the call graph **backwards**, we can find outward-facing functions that users actually consume



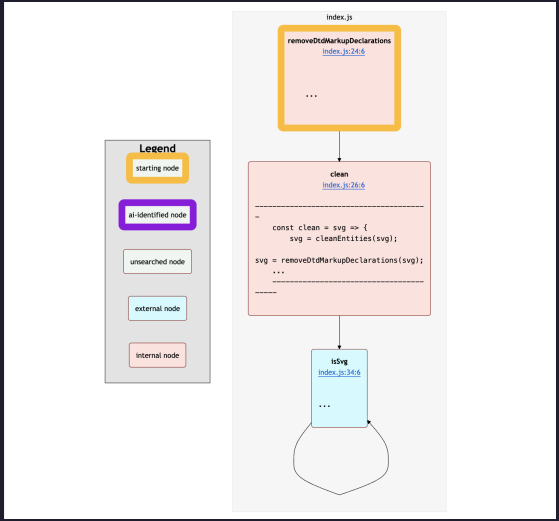
Building the Codegraph

1. Model the library's call graph as a **directed graph**: functions as nodes, caller–callee relationships as edges
2. Populate using **Semgrep program analysis**, which understands semantic caller–callee relationships in code
3. Mark functions changed by the patch commit as "vulnerable"
4. Transitively propagate: every node that eventually points to an initially vulnerable function is also marked

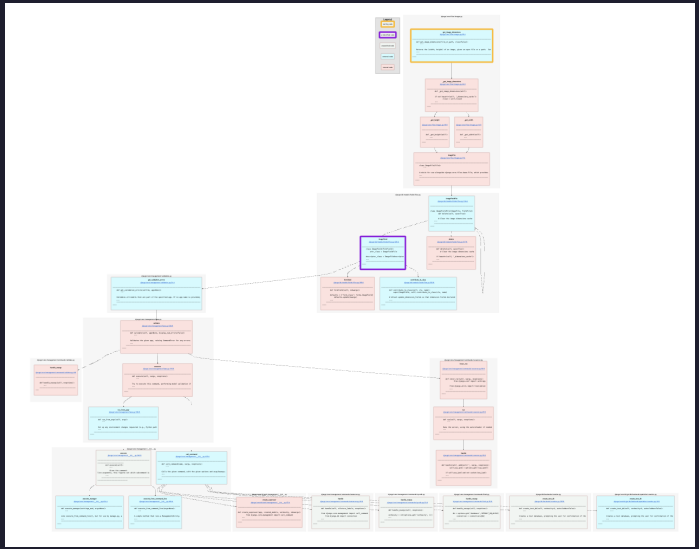
This makes codegraph generation deterministic — built on fundamental program analysis methods, not LLMs.

AI is used only in small subroutines: scraping API docs, parsing READMEs for additional context.

Codegraph Example: Simple



Codegraph Example: Complex



Filtering to Public Nodes

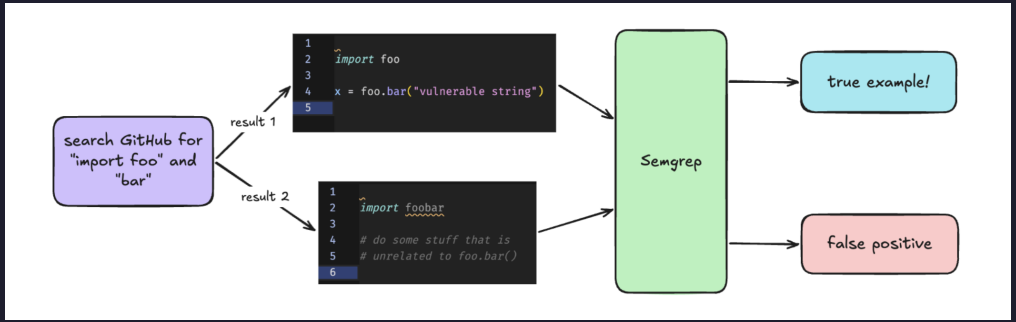
The transitive closure of a project could be hundreds of nodes — we can't realistically write a rule for all of them.

BRAT has a **filtering pass** to identify the "public" nodes: the ways a library is typically invoked in real code.

1. Use language-specific heuristics to coarse-grainedly search GitHub for code that *looks like* it uses each symbol
2. Use **Semgrep queries** to filter search results to only actual uses of the symbols

Example: Is `foo.bar()` used in real code?
Search GitHub for `"bar()"` + `"import foo"` → may get false positives (e.g., `"bar"` in a string) → Semgrep verifies which results are true usages of `foo.bar()`.

Filtering to Public Nodes (cont.)



Why Split Test Gen and Rule Gen?

Naïve approach: spawn an agent, give it the afflicted functions, call it a day.

Early attempts yielded less-than-ideal results:

- Sometimes an **invalid** Semgrep rule
- Sometimes a symbol **missed entirely**
- Sometimes a rule that didn't properly **match** the targets

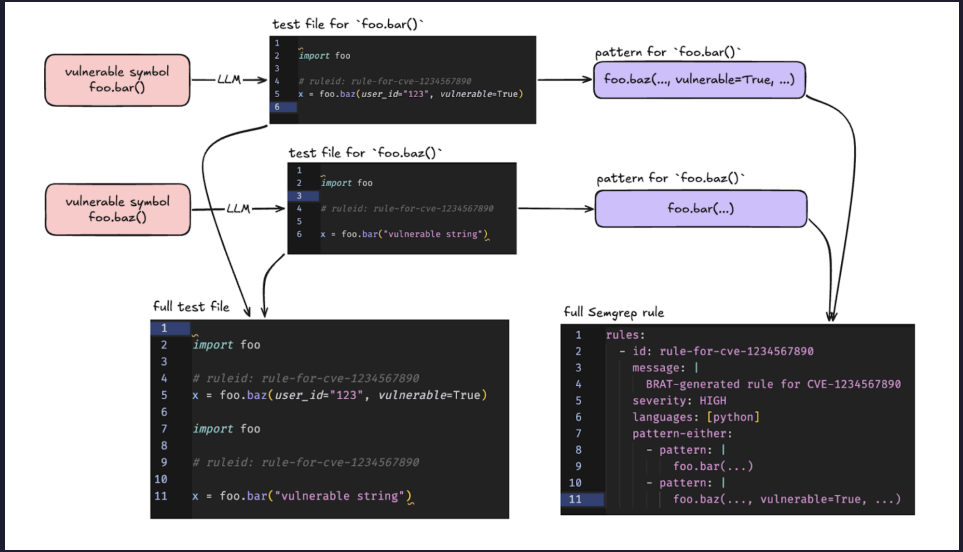
The key insight: independent functions in a package *are independent*. We can treat test file generation and rule generation on a **per-function basis**, decomposing the problem into simpler subtasks.

Per-Function Decomposition

To flag uses of `foo.bar` and `foo.baz`:

1. Ask an LLM *independently* to generate test code for `foo.bar` usage and `foo.baz` usage
2. Ask an LLM to generate a pattern matching `foo.bar` and one for `foo.baz`
3. Test each pattern **separately** — mechanically verifiable
4. If a subcomponent fails, rerun it by itself
5. Stitch all test files and patterns into a single rule

Per-Function Decomposition (cont.)



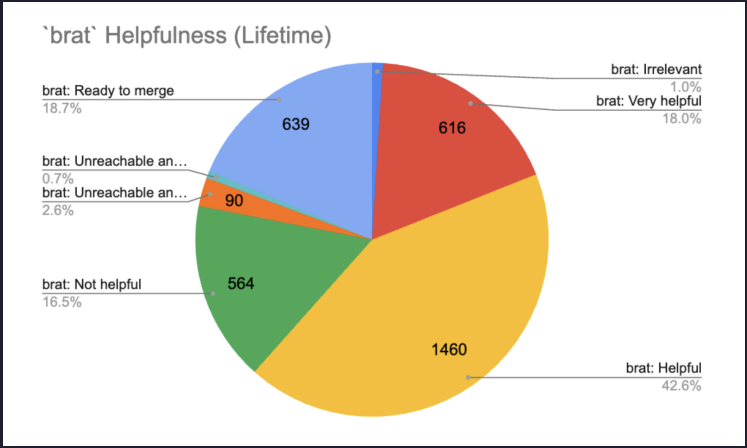
Composable Verification

The advantages are numerous:

- We **can't accidentally drop a symbol** — each is tracked independently
- We can **audit** the component of each rule responsible for each function independently
- We get a more **composable and understandable** system responsible for the final result
- Failures are **isolated** and retriable without rerunning the entire pipeline

Results: Two Years of BRAT

Over its two years of existence, BRAT has authored or co-authored **nearly 3,500 rules**.



Time Savings

Data from security researchers: researching a CVE and writing a rule takes on the order of **an hour**.

Conservative estimates for BRAT-assisted rules:

Ready to merge	∼10 min of researcher time
Very helpful	∼20 min of researcher time
Helpful	∼45 min of researcher time

Proportionally: BRAT has saved 40% of time spent writing rules overall — a tangible cost decrease in expensive human resources.

Scale Benefits

BRAT didn't replace the human workflow — it instilled a trusted, mostly-deterministic pipeline that **aided a human-in-the-loop workflow**.

- Runs at scale in parallel for arbitrarily high workloads
- On some nights, as many as **50 CVEs** filed overnight — BRAT responds to the entire backlog automatically
- Enables **backfilling** rules en masse: high/critical severity CVEs as far back as 2017 that the research team didn't have time to get to before BRAT

Case Study: CVE-2025-57820

Prototype pollution vulnerability in `devalue`, a relatively simple library.

An agent's thought process:

> *Let me get the full advisory with PoC details.*
> *Fetch(https://github.com/.../GHSA-vj54-72f3-p5jv)*
> *The vulnerability is in devalue.parse() --- the unflatten()*
> *function doesn't block __proto__ keys...*

The agent states the problem is in `unflatten`, but did not realize that `unflatten` is *also exposed by the library*.

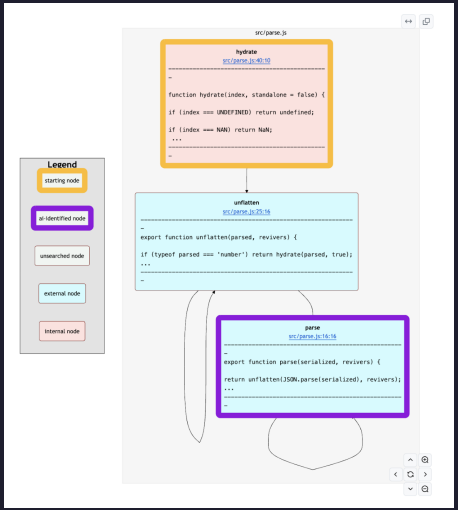
The Agent's Rule

```
rules:
  - id: CVE-2025-57820-devalue-prototype-pollution
    pattern-either:
      # require('devalue').parse(...)
      - pattern: require('devalue').parse(...)
      # const devalue = require('devalue'); devalue.parse(...)
      - patterns:
          - pattern: $MOD.parse(...)
          - pattern-inside: |
              $MOD = require('devalue');
              ...
```

All patterns look for `devalue.parse()`.

Missing entirely: `devalue.unflatten()` is also a valid insecure entry point!

BRAT's Codegraph Tells the Whole Story



- `hydrate` is the source — a "starting node" in the codegraph's traversal algorithm
- `hydrate` → called by `unflatten` → called by `parse`
- BRAT explicitly checks for public usages: identifies that `unflatten` *also* has real-world usages
- Both `parse` *and* `unflatten` are flagged as vulnerable entry points

Agent vs. BRAT

Agentic

- Fast for simple cases
- Skips steps, leaps in logic
- Not auditable or composable
- Misses non-obvious paths
- Primary improvement lever: prompt tuning

BRAT

- Thorough, subtask-oriented
- Each step independently verifiable
- Composable, retriable results
- Catches non-obvious call paths
- Improvable at each stage

Thorough, subtask-oriented algorithms are easier to trust and can lead to better results than letting an agent handle the entire problem from start to finish.

Conclusion

Shareholders and AI enthusiasts might purport that some problems are best solved by dispatching to an agent — and in many cases, they'd be correct.

But there still exist problems with decomposable, discrete parts and workflows that are best kept in Good Old-Fashioned Code (GOFC) land.

Deterministic workflows + nondeterministic LLMs
= an effect better than what either could accomplish alone.

A tasteful amount of AI ends up being the right amount.

Questions?