Exorcising the SAST Demons

How Ghost is Replacing Rule-Based Scanning with AI-Powered Triage and Detection.

A Report By Ghost Security

ghost

1. Executive Summary

TRADITIONAL SAST IS FAILING SECURITY AND ENGINEERING TEAMS, WASTING HUNDREDS OF HOURS FOR MINIMAL SECURITY VALUE.

Ghost Security scanned nearly 3,000 real-world open-source repositories across Go, Python, and PHP to test how well traditional static application security testing (SAST) tools hold up at scale.

The results were staggering:

- 2,116 potential security findings were flagged.
- Over 91% were false positives—alerts with no real risk.
- Manual triage would've taken more than 350 hours for just 180 true positives.

This isn't just inefficient. It's unsustainable.

Security teams are drowning in alerts. As development velocity accelerates, legacy SAST tools flood teams with low-signal noise, most of it false positives. AppSec and engineering teams spend hours chasing dead ends instead of delivering value to their customers. Meanwhile, real risks slip through the cracks.

Our research shows that AI-powered validation, when context-aware, can eliminate this toil, accelerate triage, and help teams focus on real risks. By combining pattern detection with reasoning about exploitability, we reduce noise and spotlight what matters.

This study scanned over 2,800 repos across Go (Gin), Python (Flask), and PHP (Laravel), comparing manual triage with AI-assisted validation.

What we uncovered reveals a critical insight: **the future of application security isn't just about identifying patterns. It's about understanding context.** Traditional SAST tools operate on syntax; effective AI-powered validation operates on semantics, intent, and risk.

Consider this: in Python/Flask projects, nearly **99.5%** of flagged command injection issues were false positives. That's thousands of misleading alerts—each demanding analyst review with little to no value in return. By contrast, our AI-assisted triage slashed review time while preserving detection quality, saving more than 350 hours across just three vulnerability classes.

And that's just the beginning. The opportunity isn't only in reducing toil—it's in rethinking detection from the ground up. Pattern-matching alone can't detect the vulnerabilities that matter most. The future belongs to systems that understand code, behavior, and context. That's what Ghost is building with **Contextual Application Security Testing (CAST).**

Why AI Triage Isn't Plug-and-Play

AI isn't a silver bullet for SAST triage. You can't just throw raw findings at a language model and expect useful results. Precision requires more:

- Vulnerability-specific prompts
- Framework-aware context
- Reasoning that mirrors expert judgment
- Signal enrichment and static/runtime cues
- Examples, examples, examples

It's not about using AI; it's about how you *structure, tune, and apply* it to the problem. Ghost's approach layers all of the above into specialized AI agents engineered to analyze findings, call supporting tools, and make informed, context-aware decisions autonomously.

2. Background

Modern development moves fast. CI/CD pipelines and AI coding assistants let teams ship multiple times a day. But this velocity comes at a cost: keeping code secure at scale is harder than ever.

To keep up, most AppSec teams rely on static application security testing (SAST) and software composition analysis (SCA) tools to catch vulnerabilities before code hits production. In theory, these tools prevent risky code from reaching production. In practice, they flood teams with noisy alerts and endless manual triage.

At Ghost, we wanted to quantify just how painful that process can be. We ran traditional SAST scans across more than 100 of our internal repositories and the results were worse than we expected:

- Nearly 5,000 findings were flagged—an overwhelming volume.
- It typically took 10–60 minutes to validate a single finding.
- Most were rated High or Critical, but due to mitigating controls or unreachable code, nearly all were non-issues.
- Tuning the rulesets was time-consuming, risky, and unreliable.
- In the end, fewer than a dozen findings were worth fixing.

The experience delivered very little security value despite hours of effort. And it was profoundly exhausting.

And it raised a bigger question:

If this is what SAST results look like in a security focused organization, how bad is it for everyone else?

Recognizing that Ghost's code repositories are unique and likely not fully representative, the research team set out to try to better understand the problem on a larger scale.

Symptoms of a Broken SAST Workflow

Here are the most common signs that your SAST workflow is broken and burning time on problems that don't matter.

- Thousands of "potential" findings—fewer than 10% worth fixing
- Triage takes 30–60 minutes per finding
- Most findings are rated as High/Critical, representing the worst-case scenario, but are not actually exploitable
- Ignore anything below "High" severity just to stay afloat
- Findings in test files or folders that pose no material risk
- 10x as many SCA findings as actual vulnerabilities in 1st-party code
- Manual rule-tuning that's hard to get right
- Security and engineering teams burned out by alert fatigue

3. Methodology

To evaluate how effective traditional SAST tools are and how much toil they generate, we designed an experiment focused on three variables:

- **Accuracy:** How many flagged vulnerabilities were exploitable (True Positives)?
- **Effort:** How much time would manual triage require to validate each finding?
- **AI Impact:** How much toil could be reduced through AI-assisted validation?

We selected three common vulnerability classes across three popular programming languages and frameworks:

- **Go + Gin:** SQL Injection (via GORM `db.First()` misuse)
- **Python + Flask:** Command Injection (via `subprocess` with `shell=True`)
- **PHP + Laravel:** Arbitrary File Upload (via `file()→storeAs()`)

For each language/framework pair, we scanned up to 1,000 actively maintained GitHub repositories—filtered by language, size (>500KB), recent commits, and at least one GitHub Star

Step-by-Step Process

1. Repository Selection

We used the GitHub repository search API to search for relevant repositories and applied filters to select actively maintained, non-template codebases.

- Queried by framework, language, and recent activity
- Filtered by repo size (>500KB), push date, and star count
- Cloned selected repos locally for scanning

2. SAST Scanning

Each repository was scanned using an open-source SAST engine.

- Applied a single rule per test targeting one vulnerability class
- Created custom detection rules for flaws not covered by default (e.g., GORM misuse in Go)

3. AI-Powered Validation

Each flagged finding was evaluated by a modern LLM.

- We evaluated multiple models and techniques before arriving at a combination of deterministic code evaluation and LLM analysis using state-of-the-art AI models

Human-in-the-Loop Validation

While AI helped drastically reduce triage time, every finding was reviewed by a human analyst before inclusion in the final results. This hybrid approach ensured that:

- Accurate classification of True and False Positives
- Correction of edge cases missed by AI
- A validated dataset that reflects real-world risk

By combining speed with expert oversight, we modeled how modern AppSec teams can realistically scale triage without compromising on quality.

AI did the heavy lifting. **Analysts made the final call.**

- Prompts were engineered with vulnerability-specific context and example-driven reasoning to maximize LLM precision
- Custom prompts and examples guided the model to classify findings as True or False Positives
- The LLM considered exploitability, attacker control, and the presence of mitigations
- For each True Positive, we generated an adjusted severity score and contextual risk assessment

4. Human Review

A human analyst verified all AI-generated judgments.

- Analysts confirmed whether or not each finding represented a real, exploitable risk
- Final results included both raw findings and validated security issues

This methodology allowed us to measure the noise-to-signal ratio for each language, estimate the manual triage burden, and quantify the time saved through AI-assisted analysis.

What We Tested

- **SQL Injection:** Dangerous user input executed in DB queries
- **Command Injection:** User input executed in OS-level shell commands
- **Arbitrary File Upload:** User input controls file names/paths during upload, risking overwrite or RCE

4. Findings & Analysis

Across the three language/framework pairs, we scanned nearly 3,000 open-source repositories and identified over 2,000 potential vulnerabilities. Each flagged finding was reviewed through our AI + human-in-the-loop triage pipeline.

In the following sections, we break down our results by language and vulnerability class, showing how traditional SAST performed and where AI added meaningful value.

4.1 Go + Gin: SQL Injection

Our first test case focused on Go applications using the Gin framework, targeting a known edge case in the `db.First()` function of the GORM ORM library.

When developers pass a string (rather than an integer) as the second parameter to `db.First()`, GORM skips query parameterization and directly interpolates the input into the SQL query, making it vulnerable to injection.

For example, this route handler passes unvalidated user input directly into a SQL query:

```
func GetUser(ctx *gin.Context) r.ResponseResult {
  id := ctx.Param("id")
  var user model.User

  if err := model.db.First(&user, id).Error; err != nil {
    ...
  }
}
```

An attacker could provide a string like `"1=1;DROP TABLE users;"`, resulting in a classic SQL injection.

The safe version validates input before use:

```
func GetUser(ctx *gin.Context) r.ResponseResult {
  idStr := ctx.Param("id")
  id, err := strconv.Atoi(idStr)
  if err != nil {
    return r.SetResponseFailure("Invalid User Id")
  }

  if err := model.db.First(&user, id).Error; err != nil {
    ...
  }
}
```

This provided the necessary knowledge to create a source code rule matching all instances of calls to `db.First( ... )` containing 2 or more parameters. This rule also included logic to match "chained" calls, such as `db.Where().First( ... )` again with two or more parameters passed.

Repository Sample and Detection Logic

We scanned **856 public Golang/Gin repositories**, filtered by size, recency, and popularity (see Appendix A). Using a custom rule, we identified instances where `db.First()` or `db.Where().First()` received unvalidated user input.

This scan returned **805 potential findings**, or nearly one finding per repo.

AI Validation Criteria

We ran each finding through a modern LLM using tailored prompts and language-specific augmented content. To be verified as a **True Positive**, a finding had to meet all three of the following criteria:

- The code is part of a real, reachable execution path
- The flaw is exploitable by external input
- No mitigations are present that would block the attack

Of the 805 findings:

- **159** were validated as True Positives (**19.75%**)
- **646** were False Positives (**80.25%**)

Based on an optimistic average of 10 minutes per manual triage, AI validation saved over **134 analyst hours** just for this single flaw.

Fun Fact

The vulnerable condition created by misusing GORM's `db.First()` is documented but rarely detected by default SAST tools, requiring custom rules to find reliably.

4.2 Python + Flask: Command Injection

Our second test case focused on **Python applications using the Flask framework**, specifically targeting command injection vulnerabilities introduced through misuse of the `subprocess` module.

The vulnerability arises when untrusted input is passed directly into a shell command using `subprocess` functions like `check_output()` or `Popen()` with `shell=True`. When this happens, attackers can inject arbitrary shell commands, potentially leading to full system compromise.

A vulnerable Flask route might look like this:

```
@app.route("/dns")
def page():
    hostname = request.values.get('hostname')
    cmd = 'nslookup ' + hostname
    return subprocess.check_output(cmd, shell=True)
```

If a user supplies a value like `example.com; rm -rf /`, both commands would be executed.

A secure implementation avoids `shell=True` and uses a list of arguments instead of a string:

```
@app.route("/dns")
def page():
    hostname = request.values.get('hostname')
    return subprocess.check_output(["nslookup"], hostname)
```

The source code matching rule looked for instances where calls to subprocess class methods such as `Popen()`, `run()`, and `check_output()` were present.

Repository Sample and Detection Logic

We scanned the top **1,000 Python/Flask repositories on GitHub**, filtered for activity, size, and popularity (see Appendix A). Using an existing rule in our SAST engine, we identified usages of `subprocess` functions with unvalidated input and `shell=True`.

This scan returned **1,166 potential findings**, or approximately **1.17 findings per repository**.

AI Validation Criteria

Each finding was passed to a modern LLM for triage using a tailored prompt and example-based reasoning. To be verified as a **True Positive**, the following conditions had to be met:

- Attacker-controlled input flows into a shell command
- The code path is exploitable in a real-world request
- No sanitization, validation, or access controls mitigate the risk

Of the 1,166 flagged issues:

- **6** were validated as True Positives (**0.51%**)
- **1,160** were False Positives (**99.49%**)

At an estimated 10 minutes per manual triage, AI validation saved over **194 analyst hours** with only 6 results requiring further review.

Why So Many False Positives?

Many alerts were triggered by benign uses of `subprocess`, or by tools calling internal scripts during CI/CD workflows. Static rules can't tell whether input is attacker-controlled or mitigated by context. AI can.

4.3 PHP + Laravel: Arbitrary File Upload

Our final test case focused on **PHP applications using the Laravel framework**, targeting a common file upload vulnerability present via an insecure use of the `storeAs()` method in Laravel's file handling system named Flysystem (version 2.x and older).

The issue arises when developers allow user-supplied input, either for the file name or file path, to flow directly into the `storeAs()` method. This opens the door to arbitrary file upload attacks, potentially allowing overwrites of critical config files or placing malicious files in sensitive directories.

Here's a simplified example of a vulnerable route handler:

```
public function upload(BackupUploadRequest $request)
{
    $file = $request->file('backup_file');
    $file->storeAs('backup/db', $file->getClientOriginalName());
}
```

An attacker could manipulate the original filename (e.g., "`../../.env`") to overwrite sensitive files.

A secure implementation sanitizes the input before storage:

```
public function upload(BackupUploadRequest $request)
{
    $file = $request->file('backup_file');
    $safeFileName = basename($file->getClientOriginalName());
    $file->storeAs('backup/db', $safeFileName);
}
```

The source code matching rule looked for instances where calls to `file()->storeAs()` were present in Laravel codebases.

Repository Sample and Detection Logic

We scanned the top **1,000 PHP/Laravel repositories** on GitHub, using the same size, activity, and popularity filters applied in previous tests. We developed a custom static analysis rule to detect potentially unsafe use of the `file()->storeAs()` method.

This scan surfaced **145 potential findings**, or roughly **0.14 findings per repository**.

AI Validation Criteria

Each finding was evaluated by a modern LLM using a structured prompt. To be classified as a **True Positive**, the following conditions had to be met:

- User input directly influences file name or path
- No input sanitization or validation is present
- The application lacks controls (e.g. auth, content-type checks) that prevent exploitation

Of the 145 findings:

- **15** were validated as True Positives **(10.34%)**
- **130** were False Positives **(89.66%)**

Assuming 10 minutes per manual review, AI-assisted triage saved approximately **24 analyst hours** on this vulnerability class alone.

Why This Flaw Matters

Arbitrary file upload vulnerabilities are often misunderstood and overlooked but can be devastating. User-supplied inputs that get sent unsanitized to file storage methods can lead to unauthorized access to environment configs or even remote code execution in certain cases.

4.4 Time Savings Summary

Across all three language and framework combinations, traditional SAST tools generated 2,116 potential security findings. After AI-assisted triage and human review, only 206 were actual vulnerabilities—a true positive rate of less than 10%.

That means over 91% of items surfaced were false positives, demanding manual effort but delivering no security value.

To estimate the burden this imposes on security teams, we applied a conservative estimate of 10 minutes per finding for manual review. The result: AI-assisted validation saved over 350 analyst hours across just three vulnerability classes.

The table below summarizes these results:

Language/ Framework	Vulnerability Class	Repos Scanned	Potential Findings	False Positives	FP Rate	True Positives	TP Rate	Hours Saved by AI Triage
Go/Gin	SQL Injection	856	805	646	80.25%	159	19.75%	134.17
Python / Flask	Command Injection	1000	1166	1160	99.49%	6	0.51%	194.33
PHP / Laravel	Arbitrary File Upload	1000	145	130	89.66%	15	10.34%	24.17
Totals		**2856**	**2116**	**1936**	**91.49%**	**180**	**8.51%**	**352.67**

Reality Check

We used a 10-minute-per-finding estimate, but the actual time varies. If triage takes 30 minutes, that's a whopping 1,058 hours saved. **If it takes 1 minute, it's still over 30 hours saved across just 3 vulnerability types.**

Bottom Line

AI didn't just save time. It transformed the triage process from a reactive, manual burden into a scalable, high-precision workflow.

5. Key Insights

The results of our experiment were clear. But the implications go far beyond false positive rates and time savings. Here's what our findings really tell us about the state of application security today:

1. The SAST Noise Problem Is Worse Than Most Teams Realize

Even with narrowly scoped scans and hand-tuned rules, more than 91% of findings were false positives. In one test, **99.5% of flagged issues were invalid**. This isn't just inefficient. It's unsustainable.

2. AI Dramatically Reduces Triage Toil Without Sacrificing Accuracy

By using AI to assess exploitability and context, we eliminated hundreds of hours of manual review. Importantly, **AI didn't miss valid issues**. It simply filtered out the noise so human analysts could focus on what matters.

3. Context Is the Missing Piece in Legacy Detection

Static pattern-matching can't distinguish real, exploitable risk from hypothetical flaws. Whether it's sanitized input, unreachable code, or internal-only paths, legacy SAST can't see the bigger picture. **Effective triage requires understanding the full context in which code runs.**

4. Teams Are Building Workarounds, Not Solutions

Ignoring "medium" findings, tuning rules aggressively, and filtering by file path aren't sustainable strategies—they're coping mechanisms. The current model forces AppSec teams into a tradeoff between accuracy and velocity. That has to change.

5. There's a Better Path Forward—And It's Contextual

This research didn't just measure inefficiency. It pointed to a new direction. **AI-powered validation**, when grounded in context and reinforced with human review, delivers scalable triage with real precision.

*This is the foundation for **CAST** (Contextual Application Security Testing) and it's where modern AppSec is headed.*

6. CAST: A New Approach

Our research confirmed the value of AI-assisted triage for traditional static analysis findings. But it also revealed a deeper insight: **many of the most dangerous vulnerabilities are fundamentally undetectable by pattern-matching tools.**

These include vulnerability classes OWASP flags as critical, such as:

- Broken Access Control – OWASP Web App Top 10 #1
- Broken Object Level Authorization (BOLA) — OWASP API Top 10 #1
- Broken Authentication – OWASP API Top 10 #2
- Broken Property Level Authorization (BPLA) – OWASP API Top 10 #3

These flaws are rarely detectable with syntactic pattern-matching alone. Identifying them requires what legacy tools lack: **context**. Specifically, understanding how application logic, user identity, and data flow interact across execution paths.

These issues often arise from a misuse of business logic, implicit authorization assumptions, or flawed transactional design. Two examples from our research illustrate this clearly.

Broken Object Level Authorization (BOLA)

In the `MakeTransfer()` function of a banking app, the source account is queried without verifying that it belongs to the authenticated user:

```
func MakeTransfer(c *gin.Context) {
	currentUser, err := helpers.GetCurrentUser(c)
	// the lack of a currentUser check on the UserId here is
	what exposes
	// these transfers to BOLA
	res := db.Where(&models.Account{
		Token:     input.AccountFrom,
		Partition: currentUser.Partition,
	}).First(&source)
```

Because there's no check that the account's `UserId` matches the authenticated user, an attacker can supply a valid account ID from another user in the same partition and transfer funds without authorization.

This isn't just a logic oversight—it's a **Broken Access Control** vulnerability (OWASP API #1) that allows unauthorized transactions and potential fraud. Traditional SAST tools completely miss this because the logic fault spans multiple objects, identity assumptions, and database queries.

Here's how our BOLA agent summarized the flaw:

> The `MakeTransfer` endpoint does not correctly verify that the authenticated user is the owner of the source account. The query that retrieves the source account checks for a matching account token and that the account's partition equals the authenticated user's partition, but it does not enforce that the account's UserId matches the authenticated user's ID.
>
> An attacker can supply an account token in the account_from field that does not belong to them but matches the partition value of the authenticated user. **Since the lookup of the source account does not verify the ownership (i.e., the UserId), the attacker can transfer funds from another user's account.** This leads to unauthorized transactions, which may result in data loss or fraud.

Race Condition in Financial Logic

In the same function, another critical flaw exists: the balance update logic runs outside of a database transaction.

```
// update source & dest account balances outside a database transaction
source.Balance = source.Balance - input.Amount
source.UpdatedAt = time.Now()
dest.Balance = dest.Balance + input.Amount
dest.UpdatedAt = time.Now()
// Save the updated balances in separate calls allowing for the race
condition
db.Save(&source)
db.Save(&dest)
```

This opens the door to concurrent requests. An attacker can rapidly issue multiple transfer requests that pass the balance check before updates are committed, resulting in unauthorized overdrafts or the synthetic creation of funds.

- This constitutes a **Broken Access Control** flaw (OWASP Web App A01:2021), as state changes can occur without consistency guarantees.
- It's also an **Insecure Design** issue (OWASP Web App A04:2021), due to missing transactional integrity in business-critical operations.

As our race condition agent put it:

> The `MakeTransfer` endpoint performs multi-step database modifications (balance deduction, addition, and transaction logging) without using database transactions or locking mechanisms.
>
> An attacker can rapidly issue multiple concurrent transfer requests using the same source account. Because the balance check and subsequent updates are performed outside of an atomic transaction, two or more requests could simultaneously pass the balance check and update the accounts. **This race condition can lead to overdrafts or creation of extra funds, bypassing intended business logic.**

These vulnerabilities don't show up as regex-matching "bad patterns." They only emerge when you understand how the app is supposed to behave and how that behavior breaks down under abuse.

What Makes CAST Different

Traditional SAST doesn't even attempt to detect these types of issues. A seasoned analyst might catch them during manual code review, but that level of effort is not scalable or cost-effective.

That's why we built CAST—**Contextual** Application Security Testing. It takes a fundamentally different approach to detection:

- Understands execution context and runtime exposure
- Uses AI agents trained to reason about logic, identity, and control flow
- Pre-indexes codebases to expose semantic relationships across files and services
- Finds OWASP-classified logic flaws missed by pattern-matching tools
- Avoids brittle, regex-driven detection models altogether

CAST doesn't just look for bad syntax. It builds a contextual model of how the app is supposed to behave—then identifies where reality breaks down.

It reasons about business logic from multiple vantage points, such as:

- From route handler inward
- From database model outward
- From CLI or API input into application behavior

Ghost's CAST platform uses a team of AI agents, each focused on specific vulnerability classes and armed with structured prompts, source code context, variable types, auth middleware logic, and data model definitions. These agents don't just "spot bad lines". They model how risks propagate through the system.

It's now possible to answer questions such as 'Are all my endpoints that need authentication properly enforcing it?' and 'For endpoints that enforce user specific authentication, does the authorization logic exist and properly restrict access to the data for just that user?'

It's a system that doesn't just scan code. It *understands* it.

7. Conclusion

Our research set out to measure the inefficiencies of traditional SAST, and the results speak volumes. Across nearly 3,000 real-world code repositories and 2,100 flagged issues, **over 91% were false positives**. In Python/Flask repos, **99.5%** of the potential findings related to command injection were ultimately irrelevant noise.

More concerning: **SAST missed entire classes of critical vulnerabilities**, including the very issues that dominate the OWASP Top 10 for web apps and APIs. Broken access controls, flawed business logic, and insecure workflows went completely undetected.

AI-assisted triage helped cut through the noise. But it's CAST that redefines the detection model.

By combining semantic indexing, runtime correlation, and a team of AI agents trained to think like security engineers, CAST:

- Filters out noise with high precision
- Surfaces true risk based on application and exposure context
- Detects OWASP-critical issues that traditional tools consistently miss
- Produces evidence-backed, developer-friendly findings that are ready for action

CAST doesn't just fix SAST. It replaces a broken model with one built for how modern software actually works.

As development velocity increases and APIs become the dominant surface area, risk shifts from simple bugs to **abuse of logic, intent, and system design**. Addressing those risks requires tools that model code semantically, not just syntactically.

That's the future Ghost is building: *One where application security is smarter, more precise, and finally aligned with how real-world systems behave.*

8. Resources and Further Reading

Static Application Security Testing (SAST)

Gartner Glossary: Static Application Security Testing (SAST)
https://www.gartner.com/en/information-technology/glossary/static-application-security-testing-sast
Overview of SAST methodology, use cases, and industry definition.

False Positives in Static Code Analysis – Parasoft
https://www.parasoft.com/blog/false-positives-in-static-code-analysis/
Analysis of common pitfalls in legacy SAST and strategies to reduce noise.

How to Reduce False Positives in SAST – Corgea
https://corgea.com/Learn/how-to-reduce-false-positives-in-sast
Practical guidance for improving signal-to-noise ratio in traditional scanning workflows.

Open Source SAST Tools

SonarQube – https://github.com/SonarSource/sonarqube
Popular open-source platform for continuous code quality and security analysis.

Semgrep - https://github.com/semgrep/semgrep
Lightweight, rule-based SAST tool with customizable pattern-matching.

CodeQL (GitHub) - https://github.com/github/codeql
Semantic code analysis engine that powers GitHub's security alerts.

Brakeman - https://github.com/presidentbeef/brakeman
Static analysis tool for Ruby on Rails applications.

Bandit - https://github.com/PyCQA/bandit
Security linter for Python codebases.

Open Web Application Security Project (OWASP)

OWASP Top 10 – Web Application Security Risks (2021)
https://owasp.org/www-project-top-ten/
The most critical web app security risks, including Broken Access Control and Insecure Design.

OWASP API Security Top 10 (2023)
https://owasp.org/API-Security/editions/2023/en/0x11-t10/
The definitive guide to the top risks facing modern API architectures.

Appendix A: GitHub Search Queries

To identify real-world, actively maintained repositories representative of typical development practices, we used the GitHub Repository Search API with targeted queries. Each query was crafted to match repositories using a specific language and framework, while filtering out inactive, trivial, or template repos.

The filters included:

- **Language**: to scope the search to relevant ecosystems
- **Size >500KB:** to exclude trivial/demo projects
- **Pushed after 2023-01-31**: to ensure recent activity
- **Stars ≥1:** to focus on public code with some adoption
- **Archived = false / Template = false:** to eliminate non-active projects
- **Sorted by stars descending:** to prioritize more popular and active repos

Go + Gin (SQL Injection)

```
gin language:Go size:>500 pushed:>2023-01-31 stars:⩾1 template:false
archived:false sort:stars-desc
```

Python + Flask (Command Injection)

```
flask language:Python size:>500 pushed:>2023-01-31 stars:⩾1
template:false archived:false sort:stars-desc
```

PHP + Laravel (Arbitrary File Upload)

```
laravel language:PHP size:>500 pushed:>2023-01-31 stars:⩾1
template:false archived:false sort:stars-desc
```

Note: These queries were executed using the GitHub API at the time of research (early 2025). Due to ongoing code activity on GitHub, exact result counts may vary if repeated at a later date.

END OF REPORT