Latio

Actually Useful Product Guide

AI Auto-Fixing

Reporting Period
Q1 • 2025

Prepared by
James Berthoty

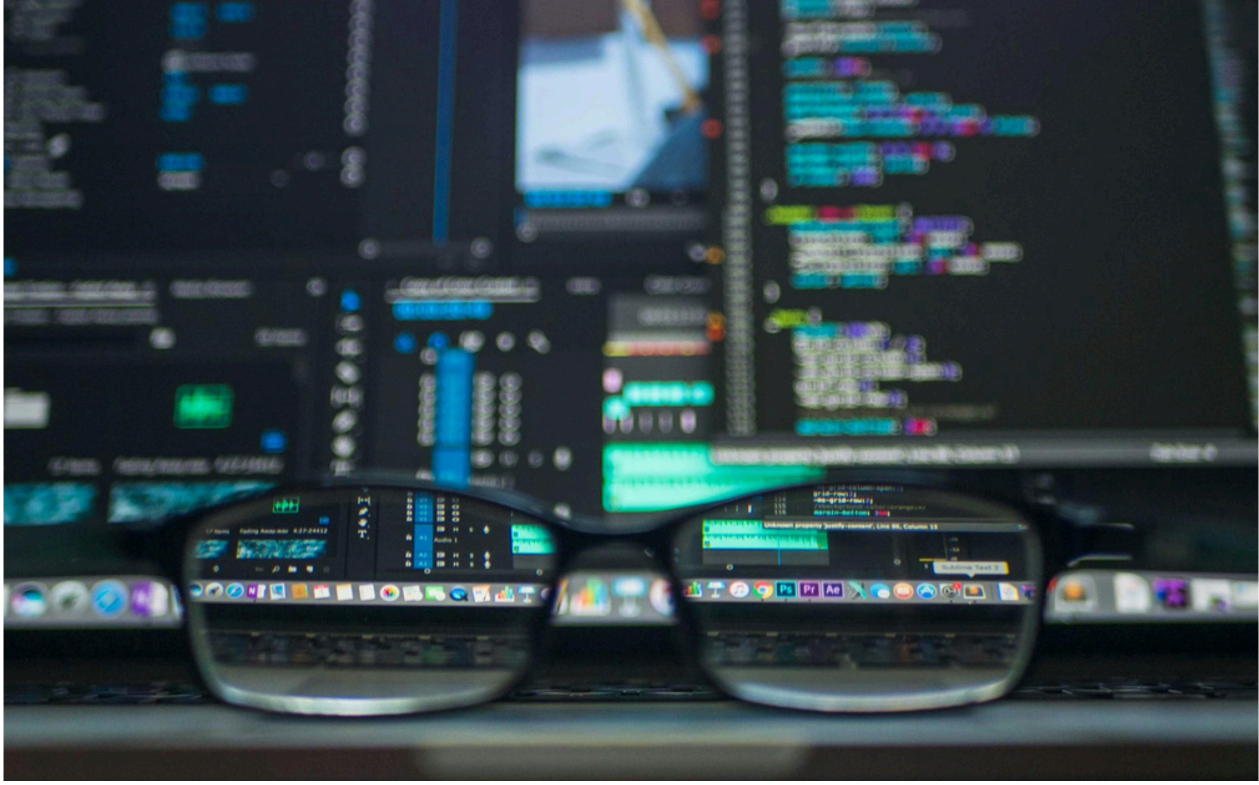


Table of Contents

The AI Code Security Landscape

As application security companies frantically re-brand into “your friendly neighborhood AI AppSec engineer,” their marketing teams would have you believe their product is the all-in-one AI engineer of your dreams. **Before you fire your entire AppSec team, this guide will help determine if AI is ready for the job.**

In this report, we’ll assess the different technical approaches taken by several vendors to see how close they get to the reality of deploying automatic code fixes to your applications and help you decide which one is the right investment for your security team.

There are two primary use cases in AI and code security:

1. Using AI to do static code analysis
2. Using AI to create fixes

In this report, we focus primarily on using AI to create the fixes for already discovered issues.

Prepared by

Latio

The Battle for the Future of Code Security: AI Upstarts vs. Established Platforms

The early days of ChatGPT led to the rapid launch of at least five dedicated AI code security companies: Amplify (2022), Corgea (2023), DryRun (2022), Pixee (2022), Mobb (2021), and Zeropath (2024).

As an analyst, I was fortunate to engage in early conversations with each of these founding teams. From our interactions, two things were immediately clear:

1. Each of these startups focused on providing developers with high-quality insights into their code by fixing the problems of traditional SAST with AI
    a. The false positive problem - most SAST findings are false positives
    b. The “Time-To-Fix” problem - it takes much longer to fix an issue than to discover it
2. Each company had a convincing and unique approach to the right way to use AI in security.

As the value of auto-fixing became clear, it didn’t take long for all major SAST providers to claim that “they, too,” do AI auto-fixing. This remains a lingering question - is AI auto-fixing a big enough moat to justify a standalone product? In this report, we’ll answer these questions:

1. Why does AI auto-fixing matter?
2. What approach to AI auto-fixing is the best?
3. What innovations are happening using AI for static code analysis?

Testing Methodology

Vendors were chosen based on their capability to detect or ingest SAST reports and create actual code fixes for those issues. This excluded vendors like Moderne and Grit, who provide ways to make large-scale code changes but do not use SAST as the middle ground. It also excluded vendors like Backslash Security, which offers AI fixing based on examples similar to your code.

Semgrep's scan results were used as a baseline for code fixing for this study. Semgrep was chosen because it was the vendors' most widely supported scan engine. This affected two vendors particularly negatively because they didn't share the same baseline detections: Snyk and Mobb. Due to the different baselines, both vendors were excluded from most of the visuals, but the raw results are still available. Mobb's Semgrep CE support is in beta. Snyk's SAST results didn't have the crossover with Semgrep necessary to generate enough coverage to have meaningful results.

Auto-fixes were generated using the platforms, with the final code output added to the shared Google Sheet's second tab. Scores were then subjectively assessed by the team at Latio based on the following factors:

1. Were unidentified issues also fixed?
2. Were false positives identified?
3. Was the fix presented in a way that integrated with the overall code base?
4. Was the fix logical to understand the suggestion?
5. Was an elegant solution provided, or at least guided towards?

```
require File.expand_path("../../[illegible]
# Prevent database truncation [illegible]
abort("The Rails environment [illegible]
require 'spec_helper'
require 'rspec/rails'

require 'capybara/rspec'
require 'capybara/rails'

Capybara.javascript_driver = [illegible]
Category.delete_all; Category [illegible]
Shoulda::Matchers.configure do [illegible]
  config.integrate do [illegible]
    with.test_framework [illegible]
    with.library :rails
  end
end

# Add additional requires [illegible]

# Requires supporting ruby [illegible]
# spec/support/ and its [illegible]
# run as spec files by default. [illegible]
# in _spec.rb will both be [illegible]
# run twice. It is recommended [illegible]
# end with _spec.rb. You can configure [illegible]
```

The final outputs are publicly available for you to assess if helpful. General scores were added for usability, detections, time taken, and triage. These scores factor only into the "total score" and are meant to get a feel for how the product works as a whole, as the rest of the report focuses on the fixes.

Testing also occurred across many different kinds of findings and coding languages. There were tests for Python, Java, Javascript, and Infrastructure-as-Code (Iac) findings. Two findings were known false positives, which most tools correctly hid from their dashboards. Third-party libraries were used to assess if the tool provided a fix in the proper context of that library.

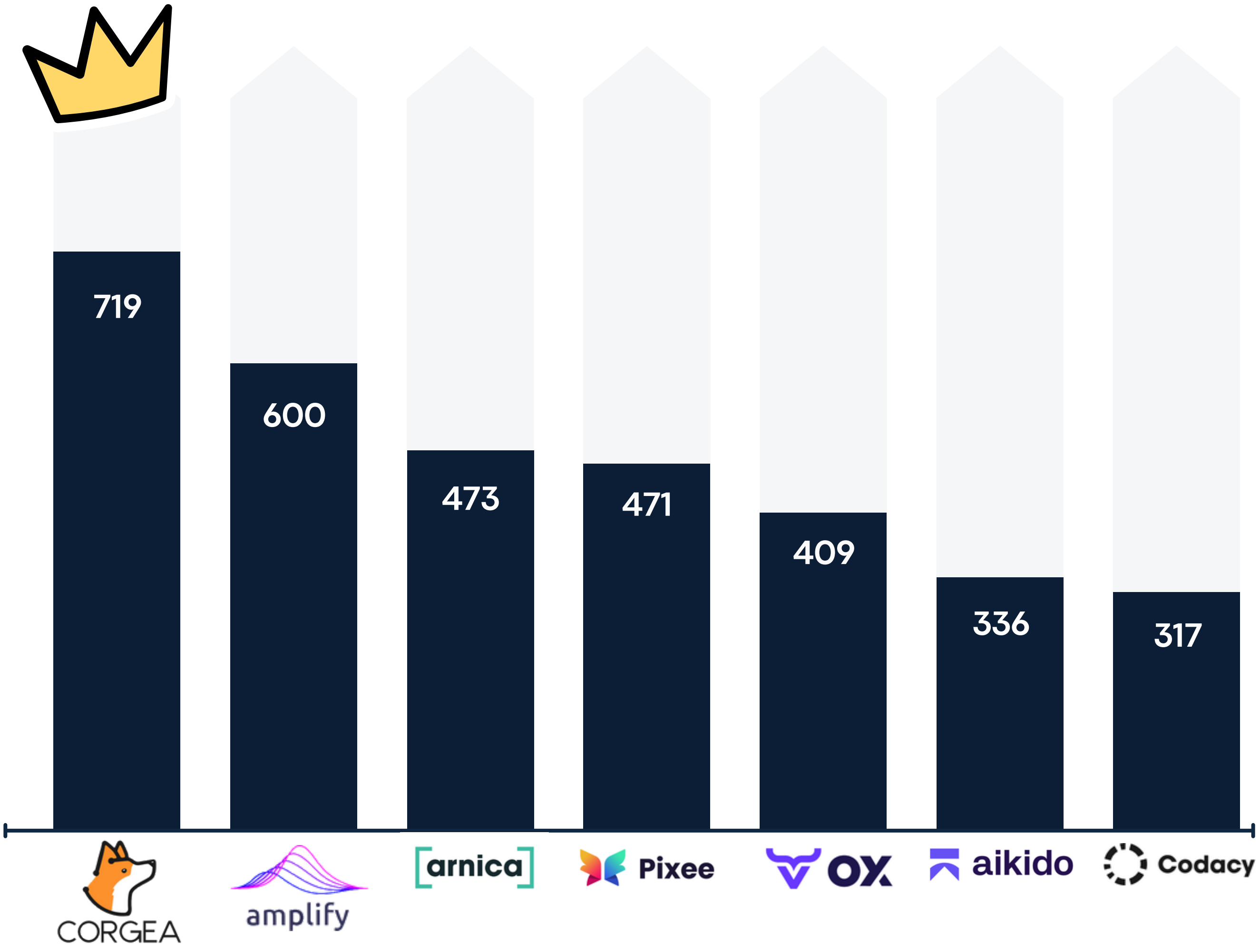
A final methodological issue is that this is in a test repo mostly built around very simple and intentionally vulnerable code. This creates weird patterns, such as when we take a user's SQL query and run it. These tests intentionally see how an LLM responds to seemingly insecure-by-design issues, but they skew the repo away from more realistic examples.

Overview of Results: Agentic AI Leads in Coverage and Quality

Raw Results and Scoring

Final Score

(Coverage X Quality)



Test results demonstrated a sizable advantage to vendors who took an agentic approach to creating auto-fixes. Most vendors had a clear trade-off between fix accuracy and fix coverage. Still, Corgea and Amplify provided high scan coverage without an accuracy tradeoff due to their approach of sending specific prompts to different AI agents to handle code fixes. This allowed their fixes to be accurate while supporting fixes for even obscure detections.

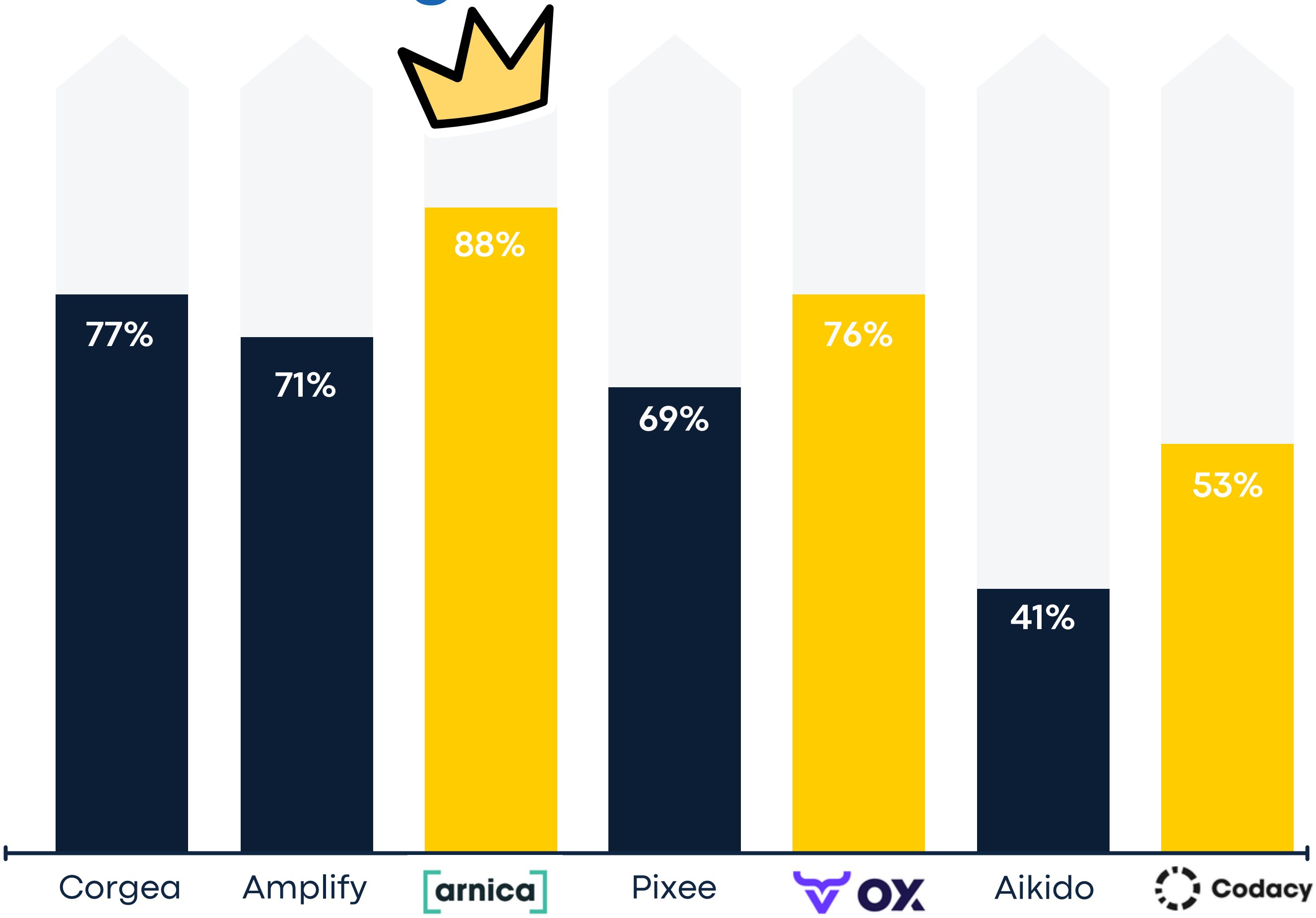
There are three general implementations of code fixing, and these approaches aligned well with the test results:

1. **Sending the Code and Finding to an LLM - tools in this first category had broad fix coverage but at the cost of fix quality.**
2. **Deterministic Solutions - tools in this category typically had a high average fix quality but at the cost of low fix coverage.**
3. **Multi-step Agentic Systems - these tools managed to have high-quality fixes and coverage.**

Other approaches led to a clear tradeoff between fix coverage and fix accuracy. Vendors who used a simple LLM integration typically had high fix coverage, but accuracy was all over the place and never quite perfect. Conversely, more deterministic vendors would have high fix accuracy, but only for findings where they had manually built rules.

Sending the Code and Finding to an LLM

Fix Coverage



Tools with simple LLM integrations had **broad fix coverage but at the cost of fix quality.** LLMs would generate a fix for any finding, and generally, those fixes were decent. However, it was easy to see how the quality would decrease the more contextual the code became to large code bases or any non-standard libraries.

LLMs typically struggled with having enough code context sent to them to generate accurate or creative fixes.

Overall, we found these tools to be helpful for finding the right direction to go with a fix, but dangerous to trust blindly. For example, they might parameterize a SQL query, but use the wrong syntax for the particular library you had chosen. While they'd make some logical changes, they would often break if you tried to copy and paste them directly and only fixed the exact issue that was called out, without regard for the application as a whole.

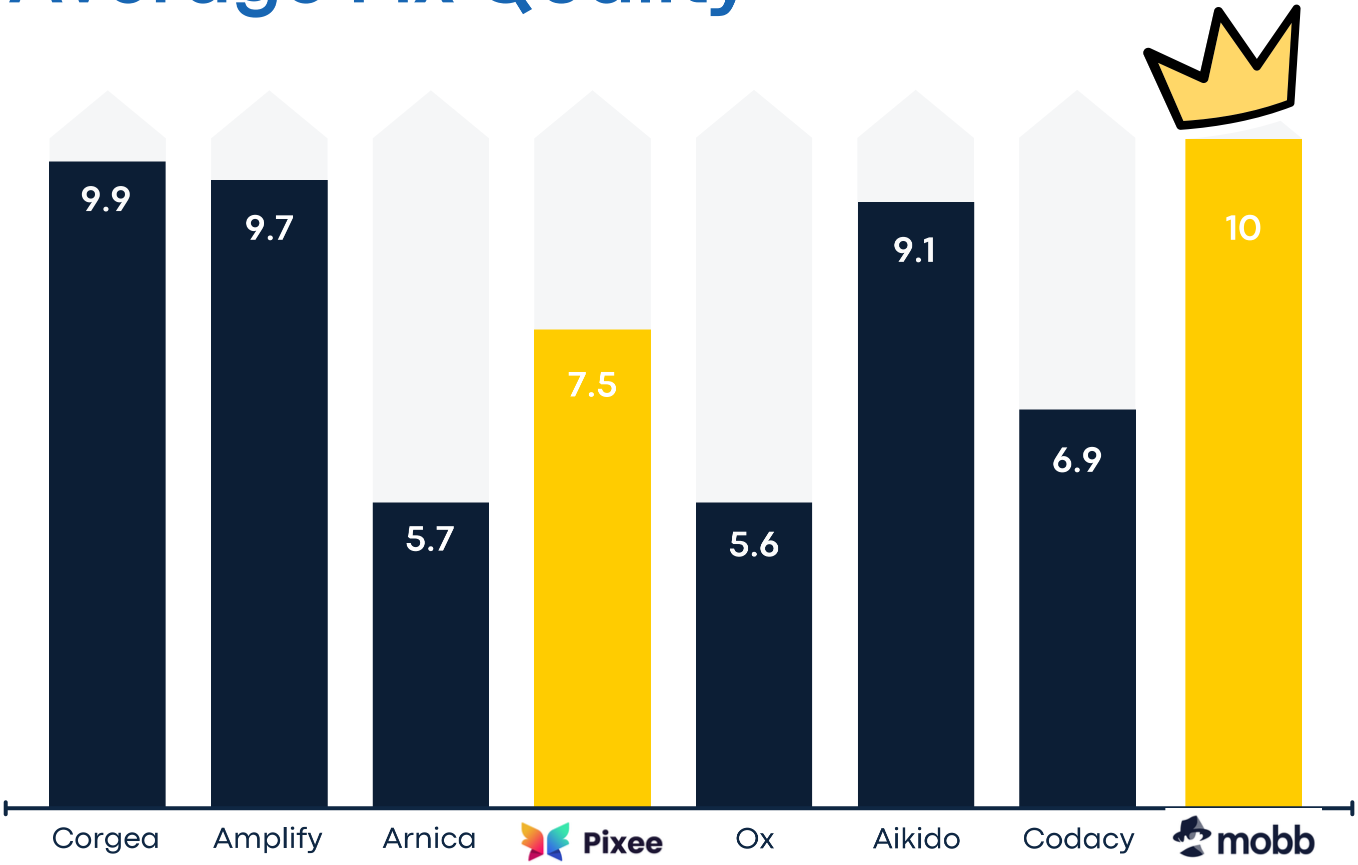
Deterministic Solutions

Tools built with deterministic fixes typically had a high average fix quality but at the cost of low fix coverage. These tools often began with deterministic fixing but have since begun adding LLM fix capabilities.

These tools were clearly more mature for specific languages, scanners, and finding types that likely reflect their early customers as they built capabilities. For example, it was easy to see where Pixee had a common pattern for fixing insecure usage of the request library in python. Critically, these vendors have been expanding into LLM, recognizing the importance of scaling fix coverage.

In general, the biggest benefit of these tools is **knowing what the pattern of the fix is going to be**, allowing you to have repeatable approaches to fixing common weaknesses - which is why they might appeal more to enterprise buyers. **The extremely limited sample size is why Mobb was excluded from elsewhere in this report.**

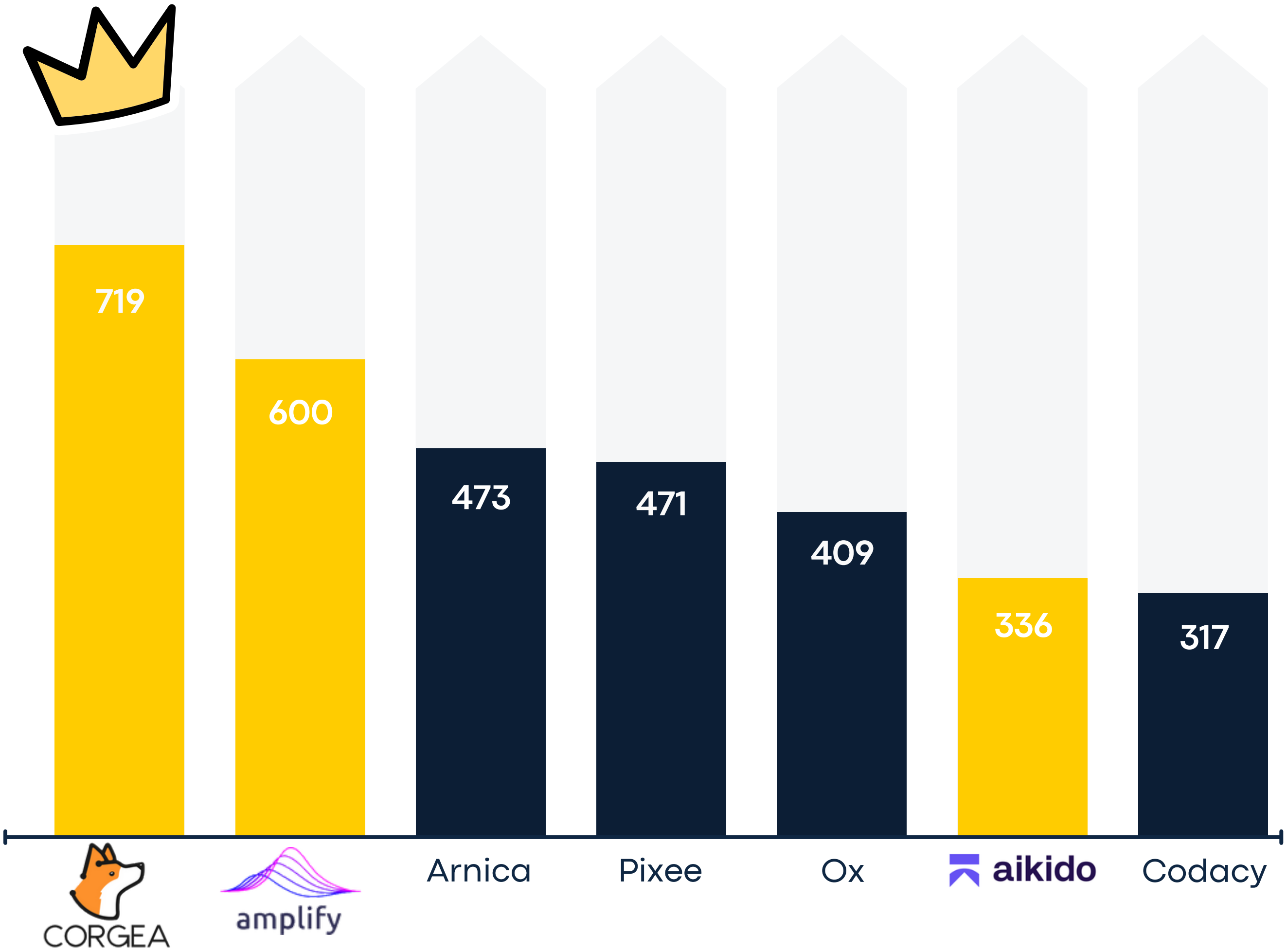
Average Fix Quality



Multi-step Agentic Systems

Final Score

(Coverage X Quality)



Finally, we're convinced that tools that dove into agentic architectures early on have benefited immensely from the approach. These fixes are often the massive improvements security teams look for. By that, we mean that these tools usually do much more than fix the immediate issue and have the rare "surprise and delight" feeling that comes with seeing unexpected issues get fixed, too.

Here are some of the surprising benefits of code security tools with agentic architectures:

1. **Fixing error messages to make them more contextual**
    a. For example, several code fixers might suggest providing a list of approved domains but typically won't also update the corresponding error message in the context.
2. **Removing other vulnerabilities that would come from a subsequent scan**
    a. In one example, we return the user's input back to them, but it's also a SQL injection. AI tools would see how this is also an XSS vulnerability and fix the return message as well.
3. **Properly identifying false positives or unfixable issues**
    a. One side effect of running agentic discussions is they dissect the murkiness that comes with something being a false positive or not in the larger application context.
4. **Being creative also increases code quality itself or refactoring functions of the app that don't make sense.**
    a. Most code has opportunities for refactoring where agentic models can improve quality alongside security if it's the right situation.

These agentic solutions were creative in finding additional code context and creating fixes that balanced developer ease of implementation with security concerns.

In general, these agentic solutions combined LLMs in conjunction with abstract syntax trees to find relevant data, and those data points were discussed in the context of different guidance from each agent. For example, relevant functions would be discovered across the code base and then discussed between different agents, prioritizing both the code and the security tradeoffs, especially considering the context of different libraries being used. This created holistic fixes that fit the code of the particular application.

Common Pros & Cons

Across all of the tools, here are some basic pros and cons you can use in your evaluations.

Great tools here would:

Implement creative solutions that address more than the immediate vulnerability

Fix not just the code itself but also contextualize error messages and outputs

Automatically generate fixes for all findings instead of requiring a ton of clicks

Provide fixes using the correct syntax for the application’s libraries

Common mistakes were:

Using the wrong syntax for a library

Providing a half-fix that was overly focused on the specific vulnerability

Be misled by the SAST finding itself into creating a fix that didn’t make sense

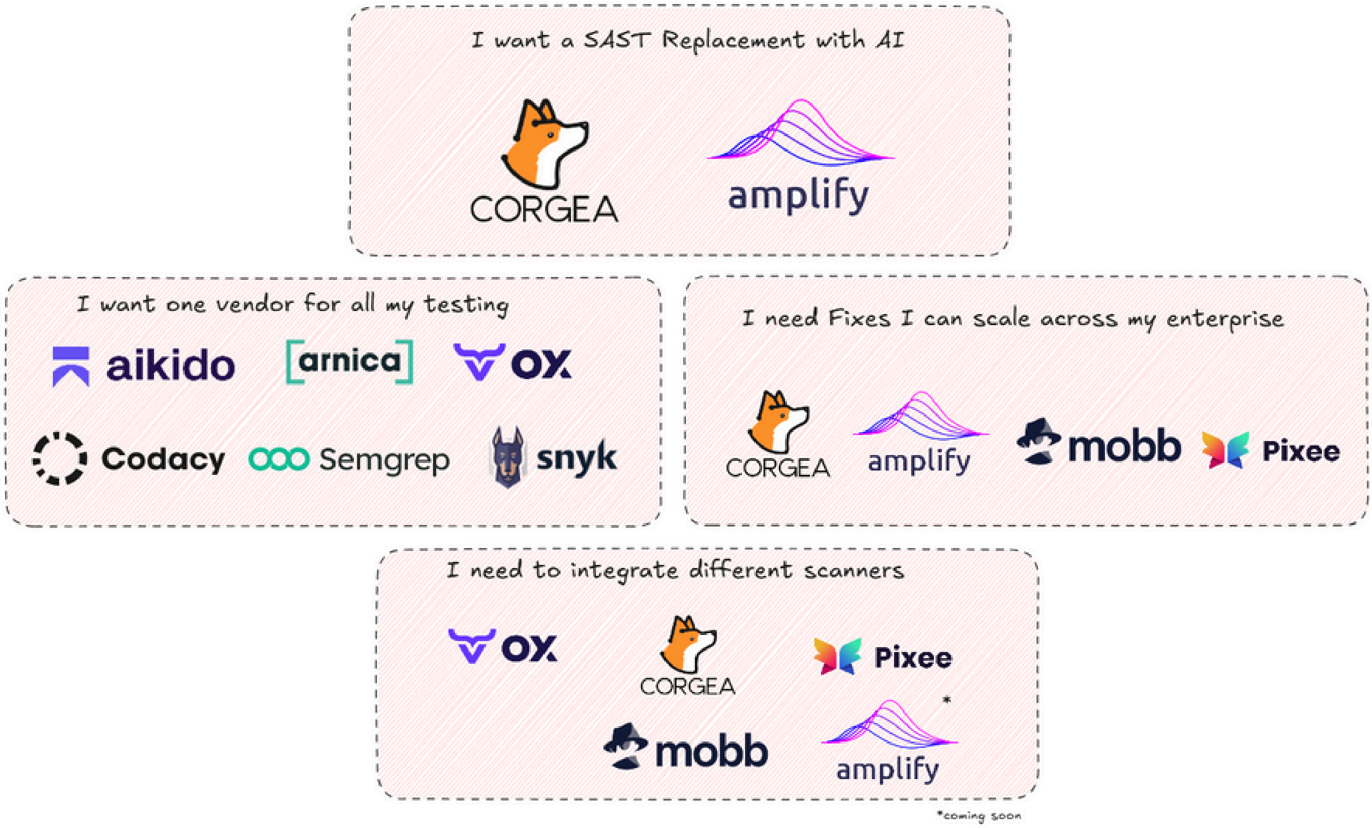
Not give the LLM enough context to make a good decision

How to Choose an Auto-Fix Vendor

All diagrams are available here.

Unlike the picture some quadrant-based approaches tend to paint, the reality of a security buying decision isn't so simple - the right tool for one company might be very different than the tool for another.

Instead, we've selected vendors based on those best fitting specific use cases to aid your buying decision. In-depth opinions on each vendor are at the bottom of the article.


I want a SAST Replacement with AI
CORGEA
amplify
I want one vendor for all my testing
aikido
arnica
OX
Codacy
Semgrep
snyk
I need Fixes I can scale across my enterprise
CORGEA
amplify
mobb
Pixee
I need to integrate different scanners
OX
CORGEA
Pixee
mobb
amplify
*
*coming soon

Actually Useful Logo Image™

Simplifying the Complexity of Modern Security Tooling Buying Decisions

Unlike the picture some quadrant-based approaches tend to paint, the reality of a security buying decision isn't so simple - the right tool for one company might be very different than the tool for another.

Instead, we've selected vendors based on those best fitting specific use cases to aid your buying decision. In-depth opinions on each vendor are at the bottom of the article.

T**he first question to answer when choosing an auto-fix vendor is the specific problem you're trying to solve. Here are the most common problems that cause an auto-fixing vendor to be considered:**

1. An enterprise has too large a backlog and isn't making progress
2. A mid-market company doesn't want to waste developer time fixing imaginary issues
3. Security teams don't have the code expertise needed to provide relevant guidance

On the one hand, most mid-market companies will ultimately only use one scanning tool, so buying a consolidated choice is a clear advantage to a tool for your tool. On the other hand, the number of false positives produced by SAST vendors is shocking to first-time application security practitioners.

This problem lies at the heart of the challenge: **SAST often finds more problems than it does solutions.** For this reason, a dedicated solution built around providing the solutions can make sense. However, SAST is one scanning option for many teams wrapped up into larger platforms they're trying to manage.

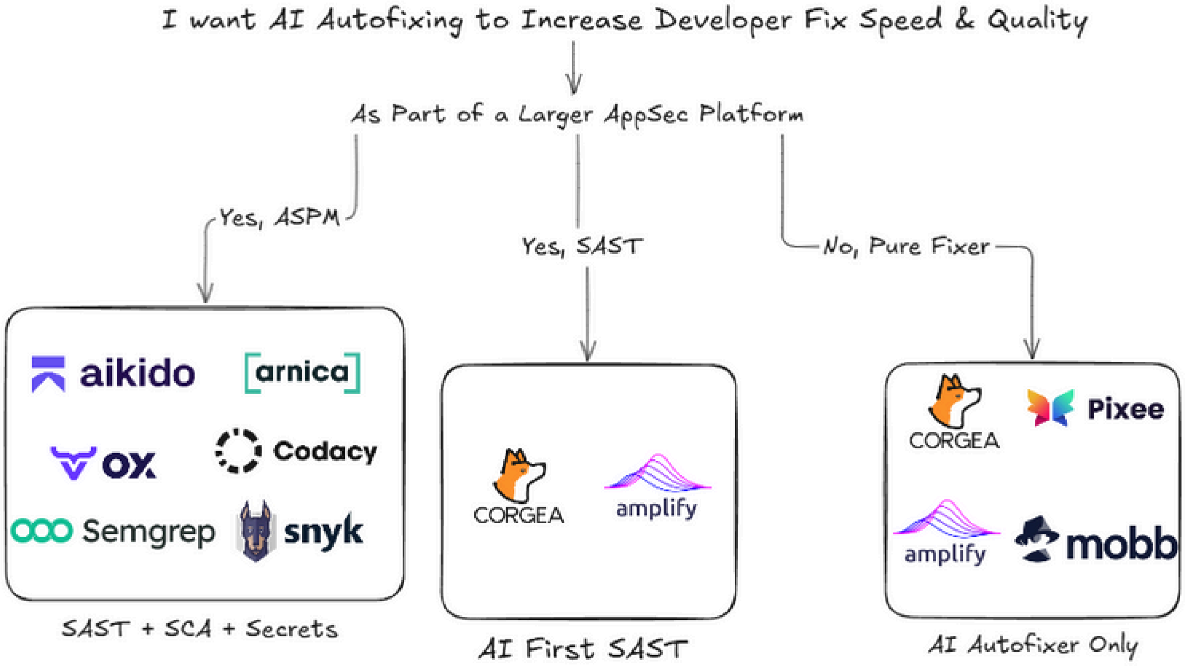
Another challenge for the mid-market is that it's unclear what exactly application security means to first-time practitioners - some think mostly of Software Composition Analysis (SCA). In contrast, others think more of Static Application Security Testing (SAST) or Dynamic (DAST). In my eyes, SAST is the best bang for your security buck. However, there are several companies in the market now that don't force you to make that choice.

For the mid-market, there are two choices: purchase a platform with varying degrees of auto-fix maturity or purchase Corgea or Amplify to focus specifically on doing SAST in the fastest way possible.

Tools built with deterministic engines typically had a high average fix quality but at the cost of low fix coverage. These tools often began with deterministic fixing but have since begun adding LLM fix capabilities.

These tools were clearly more mature for specific languages, scanners, and finding types that likely reflect their early customers as they built capabilities. For example, it was easy to see where Pixee had a common pattern for fixing insecure usage of the request library in python. Critically, these vendors have been expanding into LLM, recognizing the importance of scaling fix coverage.

In general, the biggest benefit of these tools is knowing what the pattern of the fix is going to be, allowing you to have repeatable approaches to fixing common weaknesses - which is why they might appeal more to enterprise buyers. The extremely limited sample size is why Mobb was excluded from elsewhere in this report.



For the enterprise, being a scanner replacement is a much harder sell than a fixer. Instead, these teams are looking to accelerate their time to fix with the help of AI. These are the use cases around which Mobb, Pixee, and Corgea have built mature offerings. In our assessment, the hardest part of implementing these tools is deciding how to surface them into a developer workflow. This integration challenge is hard enough for the scanning part of the tool, doing it as an extra layer creates another layer of challenges.

As for an analyst "who's going to win the market" perspective, **we love the potential of an AI-first approach to all application security.** We think the long-term possibility of Corgea and vendors who are also using AI for detection itself is massive - reimagining security scanning but built around LLMs. If that doesn't work, however, AI auto-fixing will continue to be an important comparative advantage for ASPM platforms. Here, Amplify has a distinct advantage in being built around auto-fixing Semgrep findings, it would be very easy to integrate with a larger ASPM provider.

Reflections on the Market

Here are some general takeaways from this study:

1. Agentic AI usage is a differentiator, but it's too early to tell if it provides any potential moat - the vendors who invested in this space have created much better fixes and scanning options than elsewhere. Still, as resources to create these tools become more common, it's unclear how long the moat will last.
2. The big enterprises are still pretty bad at this. As much as it's clear that auto-fixes should be a feature of ASPM, none of the significant SAST players performed very well.
3. Validation of the issue and fix is key to success, as well as creativity to fix other relevant code like errors or HTML output
4. We are still far from auto-merging these fixes ourselves, and it's doubtful we'll ever get there. Ultimately, the LLM can only intuit so much of what an application tries to do unless we run it.
5. The LLM-based testing is still very exciting for identifying BOLAs and real issues.
6. For as much as AI's capabilities for triaging have been touted, the feature feels very much in beta from most providers because they don't surface the logic behind the reprioritization decision. Sometimes, it makes sense, sometimes, it's extremely puzzling.

Agentic approaches offer a worthwhile path forward for auto-fixing tools, but we're still far away from a friendly neighborhood AI AppSec professional.

Latio

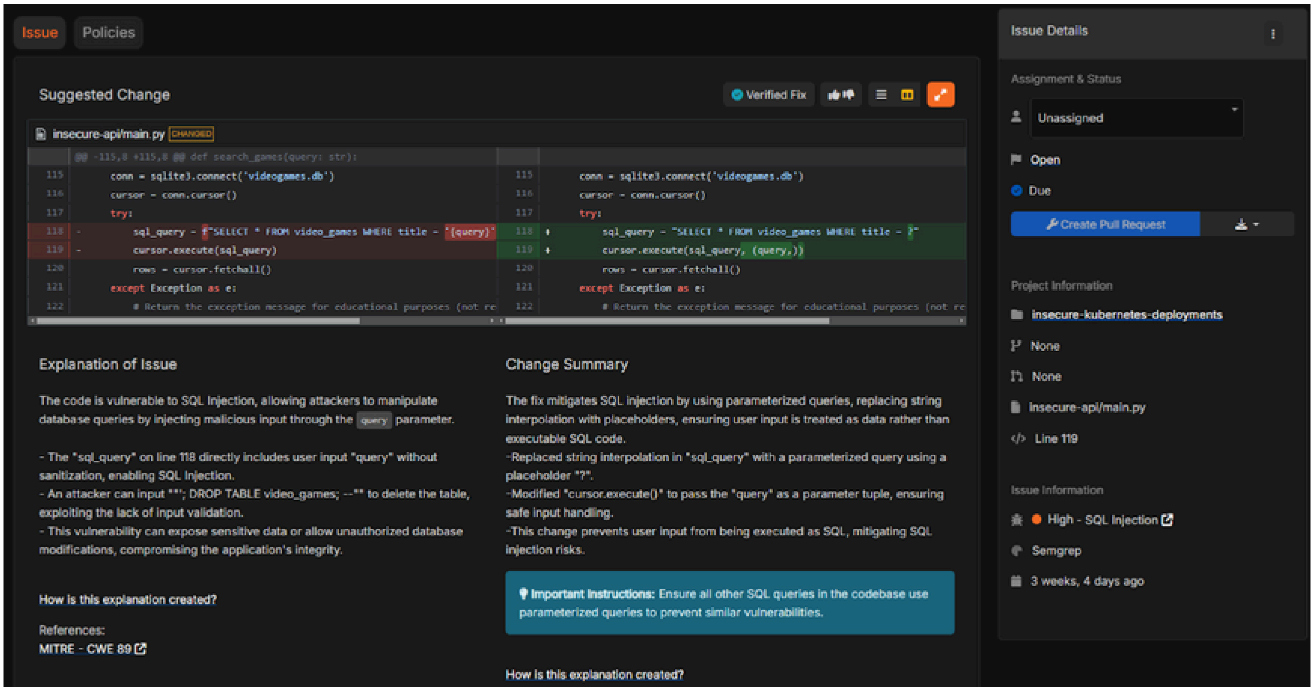
Vendor Breakdown

FEB • 2025

Prepared by
James Berthoty

CORGEA

Corgea was pretty mindblowing in a lot of respects. Before getting to the quality of their auto-fixing, they've also built a robust LLM-based SAST scanner. This scanner, called BLAST, is alongside DryRun and Zeropath and is one of the only AI scanning engines out there.



insecure-api/main.py

```
@@ -115,8 +115,8 @@ def search_games(query: str):
115        conn = sqlite3.connect('videogames.db')
116        cursor = conn.cursor()
117        try:
118  -         sql_query = f"SELECT * FROM video_games WHERE title = '{query}'
119  -         cursor.execute(sql_query)
120            rows = cursor.fetchall()
121        except Exception as e:
122            # Return the exception message for educational purposes (not re
```

```
115        conn = sqlite3.connect('videogames.db')
116        cursor = conn.cursor()
117        try:
118  +         sql_query = "SELECT * FROM video_games WHERE title = ?"
119  +         cursor.execute(sql_query, (query,))
120            rows = cursor.fetchall()
121        except Exception as e:
122            # Return the exception message for educational purposes (not re
```

Explanation of Issue

The code is vulnerable to SQL Injection, allowing attackers to manipulate database queries by injecting malicious input through the query parameter.

- The "sql_query" on line 118 directly includes user input "query" without sanitization, enabling SQL Injection.
- An attacker can input "'; DROP TABLE video_games; --" to delete the table, exploiting the lack of input validation.
- This vulnerability can expose sensitive data or allow unauthorized database modifications, compromising the application's integrity.

How is this explanation created?

References:
MITRE - CWE 89

Change Summary

The fix mitigates SQL injection by using parameterized queries, replacing string interpolation with placeholders, ensuring user input is treated as data rather than executable SQL code.
-Replaced string interpolation in "sql_query" with a parameterized query using a placeholder "?".
-Modified "cursor.execute()" to pass the "query" as a parameter tuple, ensuring safe input handling.
-This change prevents user input from being executed as SQL, mitigating SQL injection risks.

Important Instructions: Ensure all other SQL queries in the codebase use parameterized queries to prevent similar vulnerabilities.

How is this explanation created?

Issue Details

Assignment & Status
Unassigned
Open
Due
Create Pull Request

Project Information
insecure-kubernetes-deployments
None
None
insecure-api/main.py
Line 119

Issue Information
High - SQL Injection
Semgrep
3 weeks, 4 days ago

The developer experience of Corgea was simple and intuitive, with good explanations alongside the change summaries to fix an issue. It also supports some of the contextual policy building that's emerging as a way to scale out false positives and create scalable custom fixes for specific issues.

- Its urgency calculations seemed good, but without explanations as to why something got prioritized or deprioritized, they can be challenging to assess.

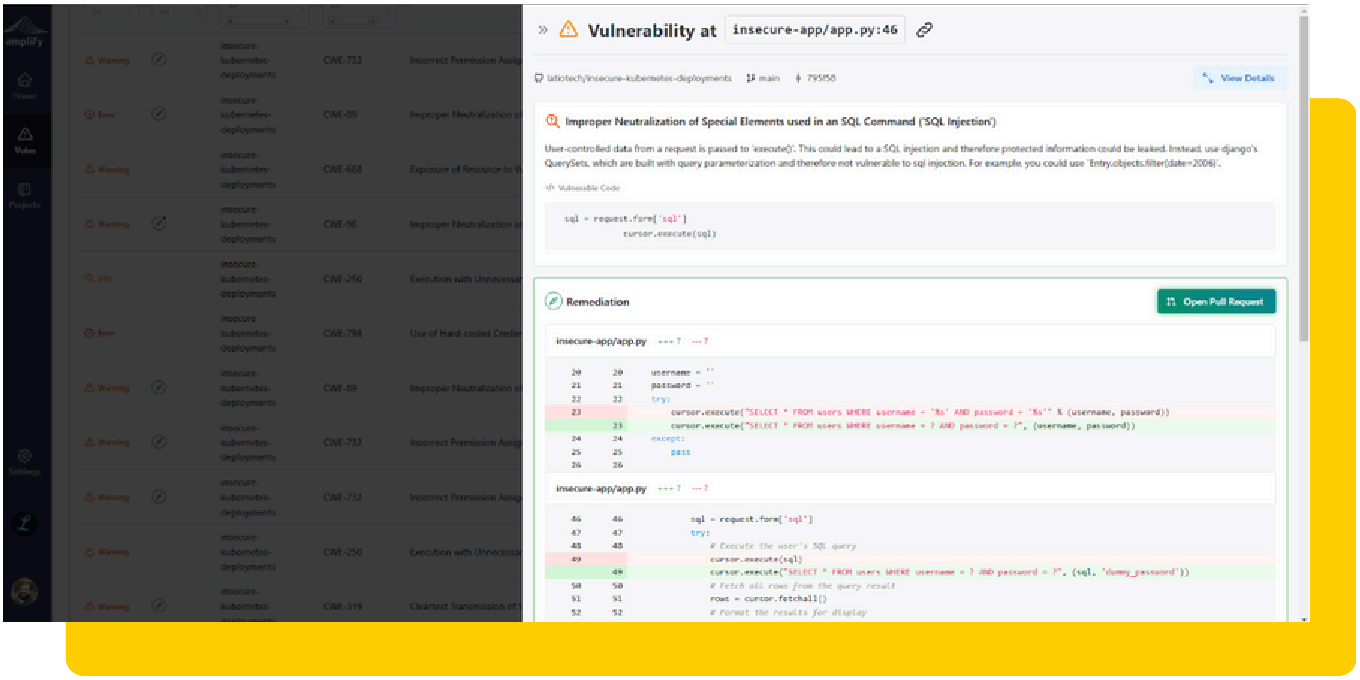
- Corgea's approach to prompting has been carefully engineered around an agentic approach, involving separate flows for fixes, validations, and contextually finding relevant functions and files.

- One potential fear could be indeterministic fixes. However, in testing, Corgea always fixed the same issue with the same fix but didn't always follow the same patterns. While other vendors are very concerned about this, we think it better fits reality - there are times to follow specific patterns, but real code bases are messy and sometimes demand messy fixes.

- Corgea is building more and more of a fully featured SAST platform with some of the boring enterprise features like reporting, SLA tracking, and user permissioning getting launched. Currently, the weakest part of the product is the CI/CD workflow for auto-fixing findings from a known scanner.

amplify

Amplify provides everything a business needs to get started with SAST. Built on top of Opengrep, they provide static results and monitoring like most, but with their agentic system for creating AI fixes. Their fixes were high-quality and relevant, using the appropriate libraries and suggestions for fixing broader issues.



Warning		insecure-kubernetes-deployments	CWE-732	Incorrect Permission Assig
Error		insecure-kubernetes-deployments	CWE-89	Improper Neutralization o
Warning		insecure-kubernetes-deployments	CWE-668	Exposure of Resource to W
Warning		insecure-kubernetes-deployments	CWE-96	Improper Neutralization o
Info		insecure-kubernetes-deployments	CWE-250	Execution with Unnecessar
Error		insecure-kubernetes-deployments	CWE-798	Use of Hard-coded Creder
Warning		insecure-kubernetes-deployments	CWE-89	Improper Neutralization o
Warning		insecure-kubernetes-deployments	CWE-732	Incorrect Permission Assig
Warning		insecure-kubernetes-deployments	CWE-732	Incorrect Permission Assig
Warning		insecure-kubernetes-deployments	CWE-250	Execution with Unnecessar
Warning		insecure-kubernetes-deployments	CWE-319	Cleartext Transmission of S

Vulnerability at `insecure-app/app.py:46`

latiotech/insecure-kubernetes-deployments main 795f58 View Details

Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

User-controlled data from a request is passed to 'execute()'. This could lead to a SQL injection and therefore protected information could be leaked. Instead, use django's QuerySets, which are built with query parameterization and therefore not vulnerable to sql injection. For example, you could use 'Entry.objects.filter(date=2006)'.

Vulnerable Code

```
sql = request.form['sql']
            cursor.execute(sql)
```

Remediation   Open Pull Request

insecure-app/app.py +++ ? --- ?

```
    username = ''
    password = ''
    try:
        cursor.execute("SELECT * FROM users WHERE username = '%s' AND password = '%s'" % (username, password))
        cursor.execute("SELECT * FROM users WHERE username = ? AND password = ?", (username, password))
    except:
        pass

```

insecure-app/app.py +++ ? --- ?

```
            sql = request.form['sql']
            try:
                # Execute the user's SQL query
                cursor.execute(sql)
                cursor.execute("SELECT * FROM users WHERE username = ? AND password = ?", (sql, 'dummy_password'))
                # Fetch all rows from the query result
                rows = cursor.fetchall()
                # Format the results for display
```

One approach to fixing that proved consistently helpful was creativity. In the example above, we take user input as an SQL query and run it. Most tools struggled to suggest anything for this, but Amplify assumed, based on context, that it was a user lookup since it exists elsewhere in the application.

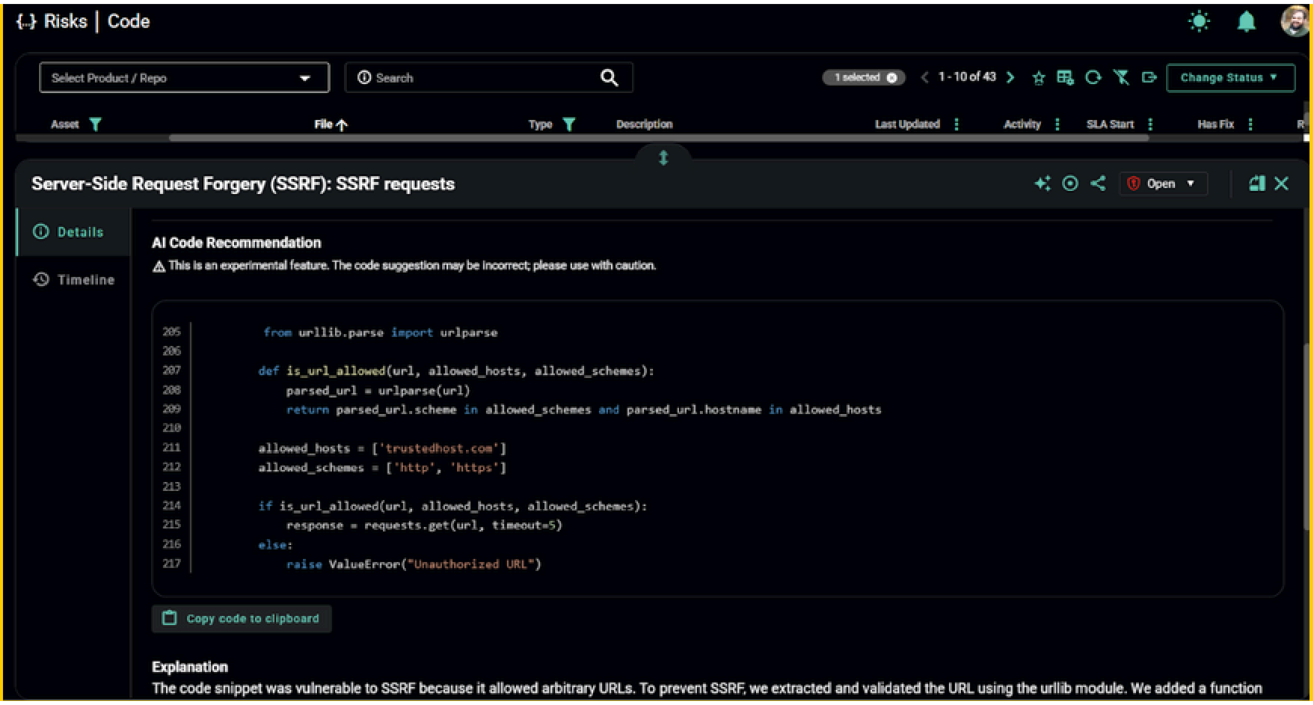
From a developer perspective, this approach is more helpful than just erroring out or failing. Amplify's approach to prompting has been engineered around an agentic approach, where a developer and security professional discuss the relevant code snippets and files and determine a solution together. One potential fear would be indeterministic fixes; however, in my testing, Amplify always fixed the same issue with the same fix but didn't always follow the same patterns.

Amplify's UI is built mostly around usability, especially for developer teams looking to get started with SAST without all the traditional overhead of running these tools. Developers are given quick views into pull request vulnerabilities and if new vulnerabilities are getting added or prevented. Currently, the weakest part of the platform is the UI for sorting and prioritizing large amounts of findings.

[arnica]

Arnica is the first larger platform to appear on this list. They provide a single orchestration platform for SAST, SCA, IaC, and Secret scanning, focusing on developer workflows. Arnica's workflow orchestration is extremely impressive, with a small example being their ability to Slack responsible developers automatically for new code pushes rather than the generic Slack channels most tools use.

Arnica's scoring this high reinforces the hard reality that some of the more sophisticated approaches to this problem just aren't worth doing. Arnica uses smart contextual prompting to get relevant fixes (i.e., not sending the bare minimum of code to the LLM), but doesn't do anything crazy with custom models or reinforcement testing. While some tools are now heavily emphasizing custom policies or otherwise, these will probably also be easy to integrate as they amount to little more than custom parts of the prompt.



Server-Side Request Forgery (SSRF): SSRF requests

AI Code Recommendation

⚠ This is an experimental feature. The code suggestion may be incorrect; please use with caution.

```
 from urllib.parse import urlparse

def is_url_allowed(url, allowed_hosts, allowed_schemes):
    parsed_url = urlparse(url)
    return parsed_url.scheme in allowed_schemes and parsed_url.hostname in allowed_hosts

allowed_hosts = ['trustedhost.com']
allowed_schemes = ['http', 'https']

if is_url_allowed(url, allowed_hosts, allowed_schemes):
    response = requests.get(url, timeout=5)
else:
    raise ValueError("Unauthorized URL")
```

Explanation

The code snippet was vulnerable to SSRF because it allowed arbitrary URLs. To prevent SSRF, we extracted and validated the URL using the urllib module. We added a function

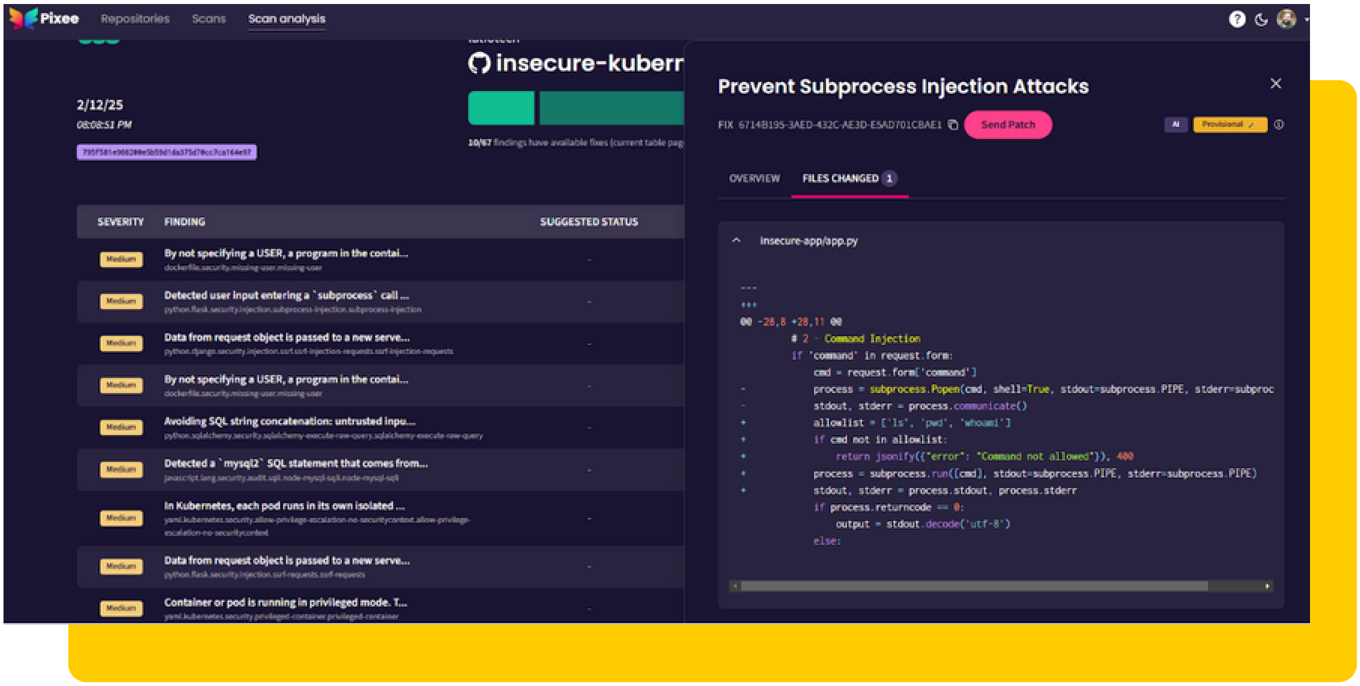
Arnica's fixes were often not quite as elegant as some of the dedicated solutions but typically followed similar fix patterns for limiting allowed queries. Every once in a while, things would get really funky or verbose, but we believe this is a byproduct of this less-deterministic approach. Additionally, Arnica hooks into your own LLM or OpenAI via an API key. For an idea of costs, we ran approximately 40 fixes through different vendors and GPT-4, which cost 3 cents.

Arnica continues to be a great value proposition for integrated scanning focused on getting shift left done properly. The weakest part of the platform is navigating the UI at scale through numerous projects.

Pixee

Pixee started ambitiously focused on developer-first security - where the helpful Pixee bot would raise PRs with auto-fixed vulnerabilities that developers simply had to merge. Unfortunately, enterprise environments need a little more prodding than GitHub bots tend to offer. Pixee has since built a much more standard interface for enterprises to get fixes to their developers.

The team at Pixee is highly focused on merge rates and triage capabilities. These focuses address two core issues with developer adoption: wasting time with bad fixes or needless fixes. Pixee will also occasionally suggest using secure libraries for common functions - such as their “secure requests” replacement for requests. This helps create repeatable fixes for common issues, but at the cost of a slightly larger learning curve as developers understand the libraries. They also integrate with a wide variety of scanners used by enterprises.



2/12/25
08:08:51 PM

10/67 findings have available fixes (current table pag

FIX 6714B195-3AED-432C-AE3D-E5AD701CBAE1

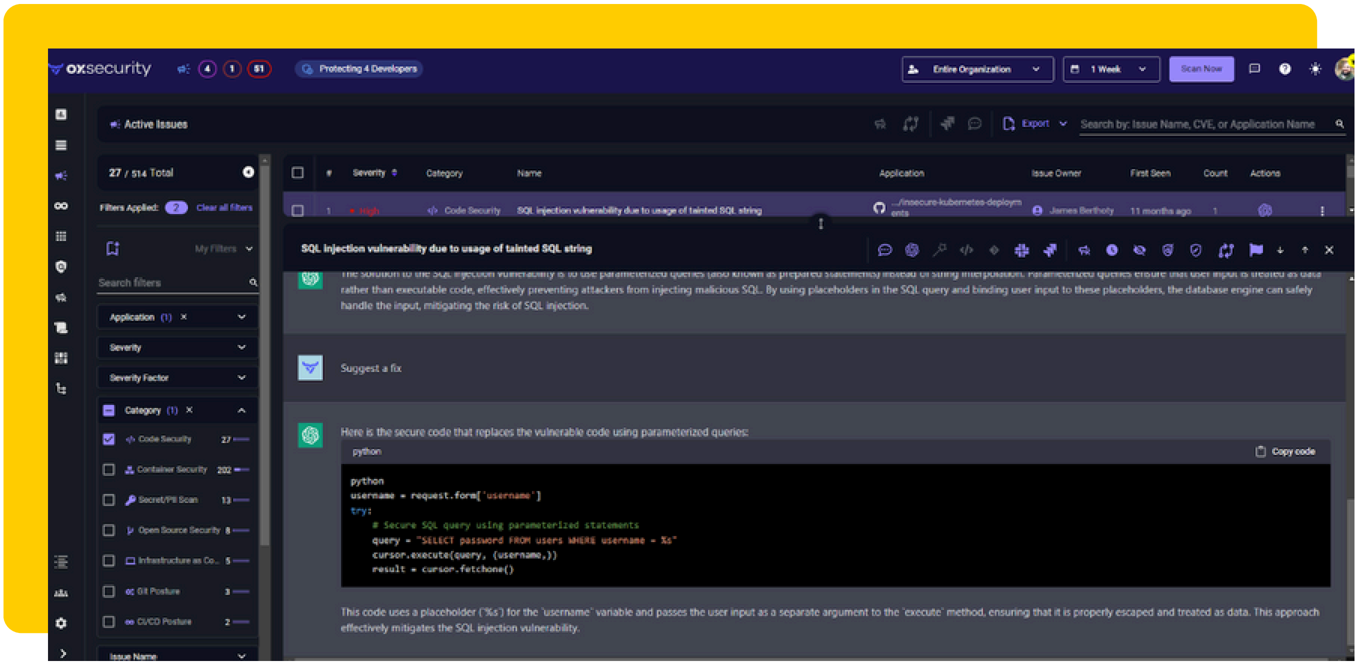
SEVERITY	FINDING	SUGGESTED STATUS
Medium	By not specifying a USER, a program in the contai... dockerfile.security.missing-user.missing-user	-
Medium	Detected user input entering a `subprocess` call ... python.flask.security.injection.subprocess-injection.subprocess-injection	-
Medium	Data from request object is passed to a new serve... python.django.security.injection.ssrf.ssrf-injection-requests.ssrf-injection-requests	-
Medium	By not specifying a USER, a program in the contai... dockerfile.security.missing-user.missing-user	-
Medium	Avoiding SQL string concatenation: untrusted inpu... python.sqlalchemy.security.sqlalchemy-execute-raw-query.sqlalchemy-execute-raw-query	-
Medium	Detected a `mysql2` SQL statement that comes from... javascript.lang.security.audit.sqli.node-mysql-sqli.node-mysql-sqli	-
Medium	In Kubernetes, each pod runs in its own isolated ... yaml.kubernetes.security.allow-privilege-escalation-no-securitycontext.allow-privilege-escalation-no-securitycontext	-
Medium	Data from request object is passed to a new serve... python.flask.security.injection.ssrf-requests.ssrf-requests	-
Medium	Container or pod is running in privileged mode. T... yaml.kubernetes.security.privileged-container.privileged-container	-

insecure-app/app.py

```
---
+++
@@ -28,8 +28,11 @@
         # 2 - Command Injection
         if 'command' in request.form:
             cmd = request.form['command']
-            process = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE, stderr=subproc
-            stdout, stderr = process.communicate()
+            allowlist = ['ls', 'pwd', 'whoami']
+            if cmd not in allowlist:
+                return jsonify({"error": "Command not allowed"}), 400
+            process = subprocess.run([cmd], stdout=subprocess.PIPE, stderr=subprocess.PIPE)
+            stdout, stderr = process.stdout, process.stderr
             if process.returncode == 0:
                 output = stdout.decode('utf-8')
             else:
```

The team is early in its move to using AI as an engine for generating fixes, but these results have proven very promising. One thing that can’t be overstated is their team’s commitment to developers first. The weakest part of the platform is navigating the fixes in the UI as the app was initially designed to be used primarily via the Github app. It’s also worth noting that their triage capabilities are more advanced for other scanning engines than Semgrep due to their focus on enterprise, so those capabilities weren’t fully explored.

oxsecurity



#	Severity	Category	Name	Application	Issue Owner	First Seen	Count	Actions
1	High	Code Security	SQL injection vulnerability due to usage of tainted SQL string	.../insecure-kubernetes-deployments	James Berthoty	11 months ago	1	

SQL injection vulnerability due to usage of tainted SQL string

The solution to the SQL injection vulnerability is to use parameterized queries (also known as prepared statements) instead of string interpolation. Parameterized queries ensure that user input is treated as data rather than executable code, effectively preventing attackers from injecting malicious SQL. By using placeholders in the SQL query and binding user input to these placeholders, the database engine can safely handle the input, mitigating the risk of SQL injection.

Suggest a fix

Here is the secure code that replaces the vulnerable code using parameterized queries:

python

```
python
username = request.form['username']
try:
    # Secure SQL query using parameterized statements
    query = "SELECT password FROM users WHERE username = %s"
    cursor.execute(query, (username,))
    result = cursor.fetchone()
```

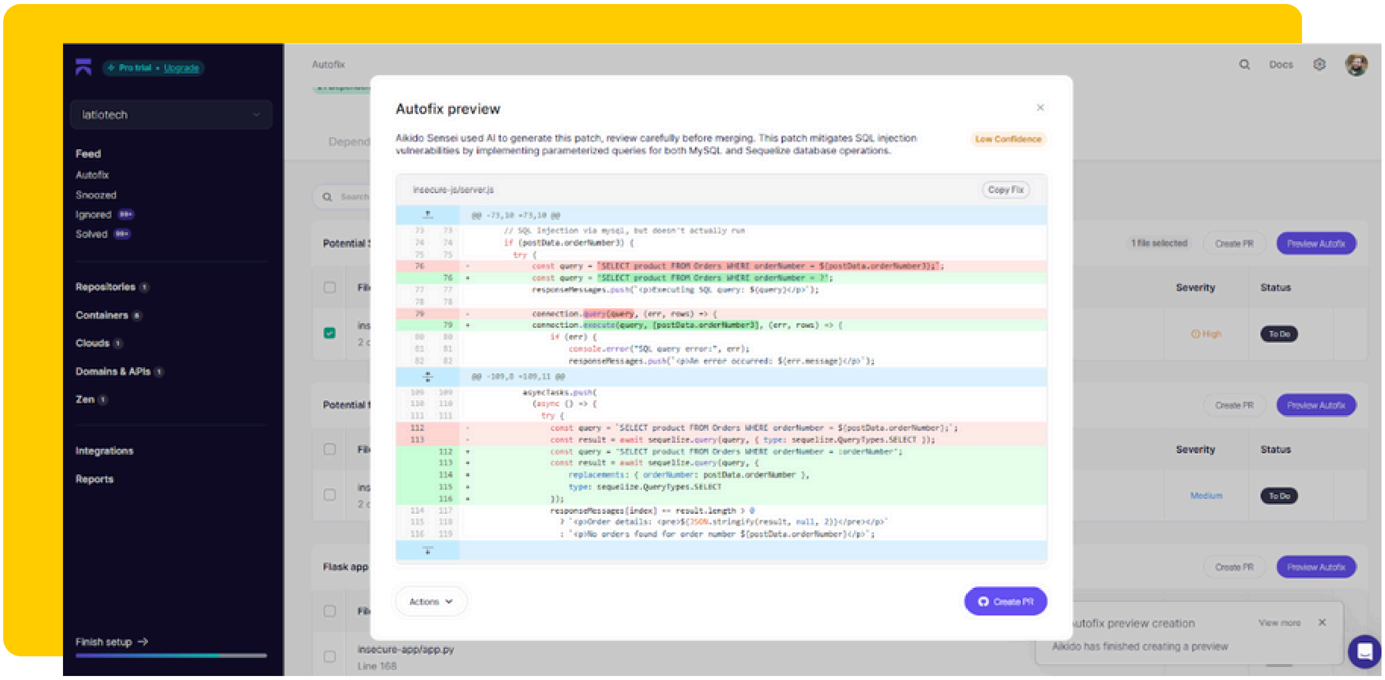
This code uses a placeholder (`%s`) for the `username` variable and passes the user input as a separate argument to the `execute` method, ensuring that it is properly escaped and treated as data. This approach effectively mitigates the SQL injection vulnerability.

While Ox performed in the middle of the pack here, the platform is doing so much that it really demonstrates how simple approaches still work well.

Ox is easily the most comprehensive platform on this list, providing most scanners you could want alongside integrations and workflows to most other providers. If you can think of a security test, Ox does it or integrates with it. Ox's implementation of AI auto-fixing is pretty basic but once again reflects the reality that GPT-4o is a pretty good model.

It seemed that most of Ox's issues stemmed from trying to send minimal code to the LLM - likely due to privacy concerns from larger customers (for clarity, you also use your API key for easy setup). This minimal code led to some fixes that either missed larger contexts or the language nuances of specific libraries.

Aikido has recently been investing massively in its AI capabilities, with the feature being launched only a couple of months ago. Their results are rapidly improving in a way that shows they'll be a major player here very shortly. Most importantly, while their fix coverage was low, their accuracy rivaled the best scores we had available - that's because the team has been rolling out the fixes on a rule-by-rule basis to prioritize accuracy.



Autofix preview

Aikido Sensei used AI to generate this patch, review carefully before merging. This patch mitigates SQL injection vulnerabilities by implementing parameterized queries for both MySQL and Sequelize database operations.

insecure-js/server.js

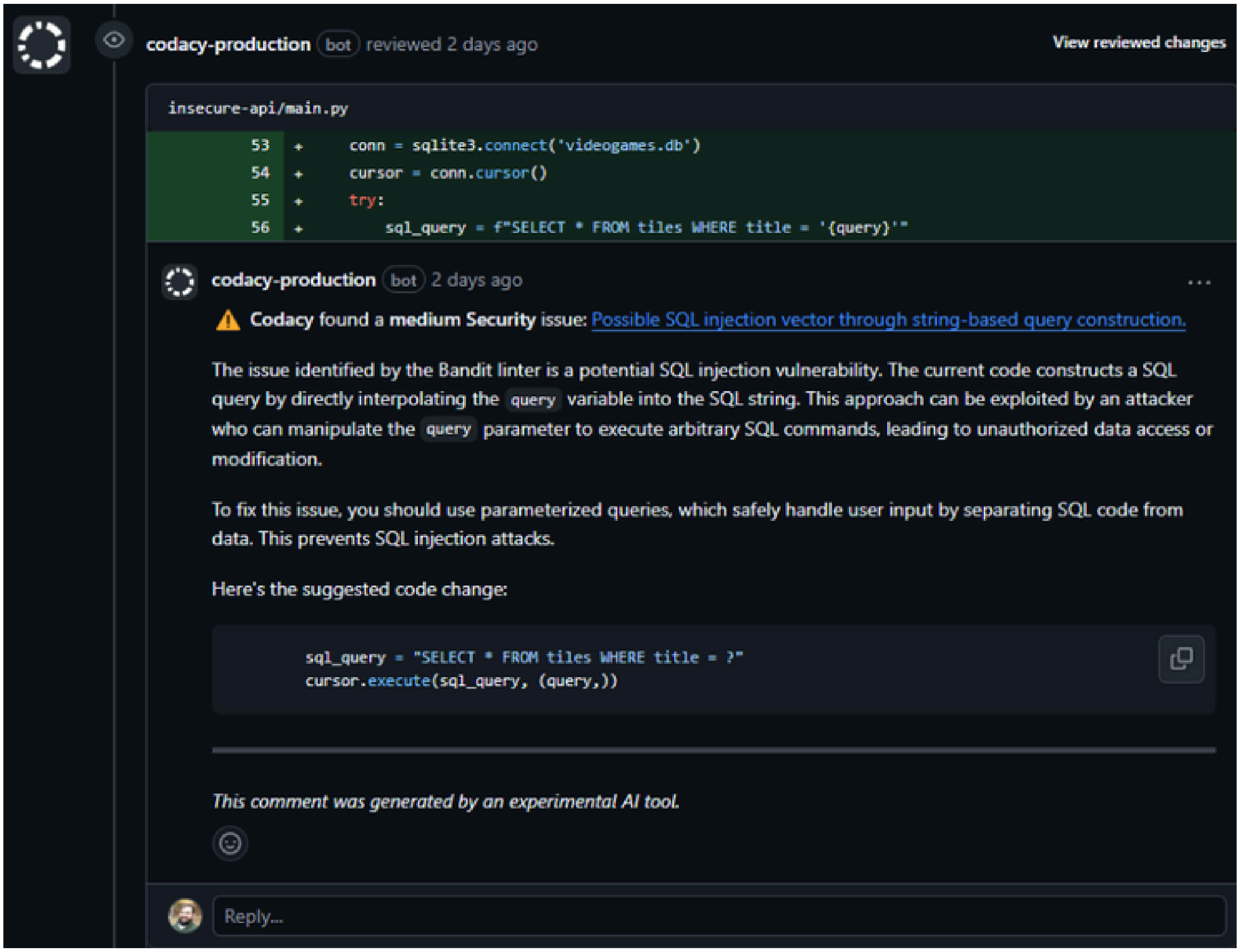
```
@@ -73,10 +73,10 @@
        // SQL injection via mysql, but doesn't actually run
        if (postData.orderNumber)) {
          try {
-             const query = `SELECT product FROM Orders WHERE orderNumber = ${postData.orderNumber}};`;
+             const query = 'SELECT product FROM Orders WHERE orderNumber = ?';
              responseMessages.push(`<p>Executing SQL query: ${query}</p>`);

-             connection.query(query, (err, rows) => {
+             connection.execute(query, [postData.orderNumber]), (err, rows) => {
                  if (err) {
                      console.error("SQL query error:", err);
                      responseMessages.push(`<p>An error occurred: ${err.message}</p>`);
@@ -109,8 +109,11 @@
            asyncTasks.push(
              (async () => {
                try {
-                   const query = `SELECT product FROM Orders WHERE orderNumber = ${postData.orderNumber};`;
-                   const result = await sequelize.query(query, { type: sequelize.QueryTypes.SELECT });
+                   const query = 'SELECT product FROM Orders WHERE orderNumber = :orderNumber';
+                   const result = await sequelize.query(query, {
+                       replacements: { orderNumber: postData.orderNumber },
+                       type: sequelize.QueryTypes.SELECT
+                   });
                    responseMessages[index] = result.length > 0
                      ? `<p>Order details: <pre>${JSON.stringify(result, null, 2)}</pre></p>`
                      : `<p>No orders found for order number ${postData.orderNumber}</p>`;
```

Aikido continues to be my first recommendation to mid-market security teams looking for encompassing developer-first testing. The platform's weakest point is that it lacks the workflow or integration capabilities of most other ASPMs targeting enterprise accounts.

Codacy

Codacy provides developer-first security testing alongside code quality and test coverage similar to Sonarqube. Their team is rapidly maturing the platform's security capabilities, and AI auto-fixing is an interesting place. However, a significant consideration is that the AI auto-fixing capabilities here also extend into code quality findings, which can bring more overall developer value. This put them at a disadvantage for this testing, but they have broader long-term developer capabilities.

Part of their prompt clearly tells the AI to try and do a one-line code fix to best fit within pull request workflows. This creates fixes that are sometimes clever and, at other times, pretty funny or unrealistic. What matters, though, is that as they mature the specificity of the prompt and where to surface results, the workflow is already built for scaling into the future.



codacy-production bot reviewed 2 days ago

insecure-api/main.py

```
+     conn = sqlite3.connect('videogames.db')
+     cursor = conn.cursor()
+     try:
+         sql_query = f"SELECT * FROM tiles WHERE title = '{query}'"
```

codacy-production bot 2 days ago

⚠ **Codacy found a medium Security issue:** Possible SQL injection vector through string-based query construction.

The issue identified by the Bandit linter is a potential SQL injection vulnerability. The current code constructs a SQL query by directly interpolating the `query` variable into the SQL string. This approach can be exploited by an attacker who can manipulate the `query` parameter to execute arbitrary SQL commands, leading to unauthorized data access or modification.

To fix this issue, you should use parameterized queries, which safely handle user input by separating SQL code from data. This prevents SQL injection attacks.

Here's the suggested code change:

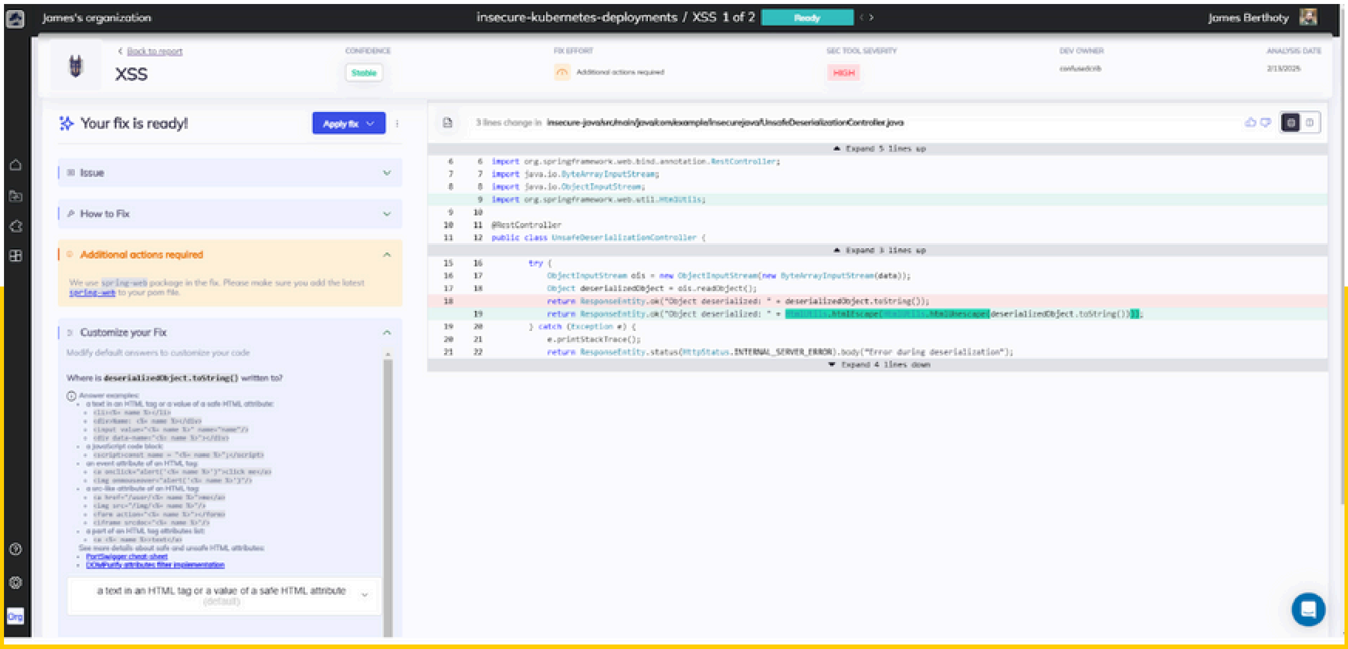
```
        sql_query = "SELECT * FROM tiles WHERE title = ?"
        cursor.execute(sql_query, (query,))
```

This comment was generated by an experimental AI tool.

Mobb has heavily focused on enterprise use cases for auto-fixing, which is why they're asterisked on this report - their support for Semgrep Community Edition is in beta, no small part of that is that most organizations at large scale are not using Semgrep compared to the bigger scanners like Checkmarx or Snyk.

Mobb also focused early on providing high-quality deterministic fixes that didn't use AI very much. This is aligned with the enterprise purpose, where if a code fix suffers from the occasional hallucination or missed fix, the consequences can be more severe due to the scale. Mobb is slowly launching pure AI-based fixes and categorizing them in a way that shows their increased volatility and whether they work or not.

The final results also reflect the approach, where Mobb had high fix accuracy for deterministic fixes, but pure AI was very early in development and not very useful. Mobb's test results should be understood extremely contextually here due to the lack of Semgrep support.



We use spring-web package in the fix. Please make sure you add the latest spring-web to your pom file.

Modify default answers to customize your code

Where is `deserializedObject.toString()` written to?

Answer examples:

- a text in an HTML tag or a value of a safe HTML attribute:
  - `<li><%= name %></li>`
  - `<div>Name: <%= name %></div>`
  - `<input value="<%= name %>" name="name"/>`
  - `<div data-name="<%= name %>"></div>`
- a JavaScript code block
  - `<script>const name = "<%= name %>";</script>`
- an event attribute of an HTML tag
  - `<a onclick="alert('<%= name %>')">click me</a>`
  - `<img onmouseover="alert('<%= name %>')"/>`
- a src-like attribute of an HTML tag
  - `<a href="/user/<%= name %>">me</a>`
  - `<img src="/img/<%= name %>"/>`
  - `<form action="<%= name %>"></form>`
  - `<iframe srcdoc="<%= name %>"/>`
- a part of an HTML tag attributes list
  - `<a <%= name %>>text</a>`

See more details about safe and unsafe HTML attributes:

- PortSwigger cheat sheet
- DOMPurify attributes filter implementation

a text in an HTML tag or a value of a safe HTML attribute (default)

3 lines change in insecure-java/src/main/java/com/example/insecurejava/UnsafeDeserializationController.java

```
▲ Expand 5 lines up
import org.springframework.web.bind.annotation.RestController;
import java.io.ByteArrayInputStream;
import java.io.ObjectInputStream;
import org.springframework.web.util.HtmlUtils;

@RestController
public class UnsafeDeserializationController {
▲ Expand 3 lines up
        try {
            ObjectInputStream ois = new ObjectInputStream(new ByteArrayInputStream(data));
            Object deserializedObject = ois.readObject();
            return ResponseEntity.ok("Object deserialized: " + deserializedObject.toString());
            return ResponseEntity.ok("Object deserialized: " + HtmlUtils.htmlEscape(HtmlUtils.htmlUnescape(deserializedObject.toString())));
        } catch (Exception e) {
            e.printStackTrace();
            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Error during deserialization");
▼ Expand 4 lines down
```

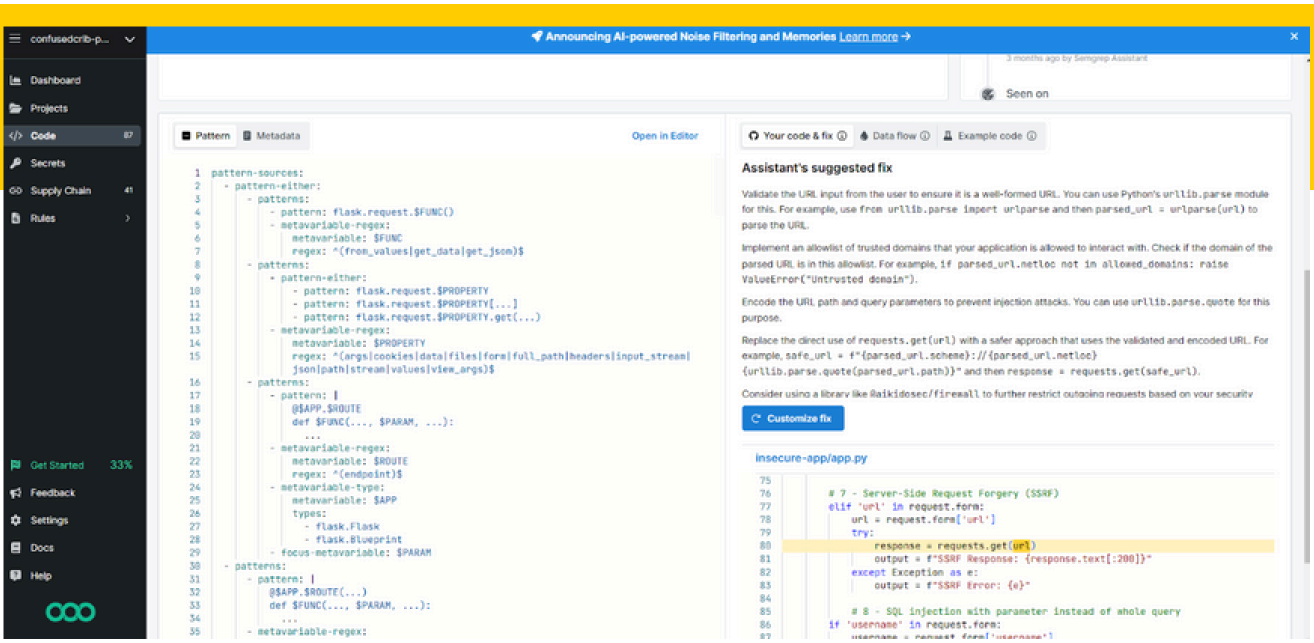
Semgrep

Semgrep's AI is in an interesting place. Of the larger incumbent SAST players, they're the furthest along in terms of fix quality and coverage; however, the current product experience is disjointed. Semgrep's triaging and fix guidance in their UI was lacking, but most of their development effort has gone towards the code fix quality in pull requests. When testing the fixes in pull requests, coverage was limited (it's possible we were experiencing a bug), but the fixes were roughly equivalent with the LLM approach.

When it comes to their methodology, Semgrep takes what I'd call an "almost agent based" approach.

The LLM creates a fix, and then another agent approves or rejects it until the system creates an acceptable space. It's possible this is one reason we experienced a lack of coverage - the autofix validator may have continued failing due to the nature of the code being insecure by design.

Additionally, the AI auto triage didn't find any false positives (despite one finding being a Django template issue in a repo that doesn't use Django, which was categorized as a True positive). While this testing didn't focus on triage capabilities specifically, nothing stood out as particularly strong.



```yaml
pattern-sources:
  - pattern-either:
    - patterns:
      - pattern: flask.request.$FUNC()
      - metavariable-regex:
          metavariable: $FUNC
          regex: ^(from_values|get_data|get_json)$
    - patterns:
      - pattern-either:
        - pattern: flask.request.$PROPERTY
        - pattern: flask.request.$PROPERTY[...]
        - pattern: flask.request.$PROPERTY.get(...)
      - metavariable-regex:
          metavariable: $PROPERTY
          regex: ^(args|cookies|data|files|form|full_path|headers|input_stream|
          json|path|stream|values|view_args)$
    - patterns:
      - pattern: |
          @$APP.$ROUTE
          def $FUNC(..., $PARAM, ...):
            ...
      - metavariable-regex:
          metavariable: $ROUTE
          regex: ^(endpoint)$
      - metavariable-type:
          metavariable: $APP
          types:
            - flask.Flask
            - flask.Blueprint
      - focus-metavariable: $PARAM
  - patterns:
    - pattern: |
        @$APP.$ROUTE(...)
        def $FUNC(..., $PARAM, ...):
          ...
    - metavariable-regex:
```

Assistant's suggested fix

Validate the URL input from the user to ensure it is a well-formed URL. You can use Python's `urllib.parse` module for this. For example, use `from urllib.parse import urlparse` and then `parsed_url = urlparse(url)` to parse the URL.

Implement an allowlist of trusted domains that your application is allowed to interact with. Check if the domain of the parsed URL is in this allowlist. For example, `if parsed_url.netloc not in allowed_domains: raise ValueError("Untrusted domain")`.

Encode the URL path and query parameters to prevent injection attacks. You can use `urllib.parse.quote` for this purpose.

Replace the direct use of `requests.get(url)` with a safer approach that uses the validated and encoded URL. For example, `safe_url = f"{parsed_url.scheme}://{parsed_url.netloc}{urllib.parse.quote(parsed_url.path)}"` and then `response = requests.get(safe_url)`.

Consider using a library like `@aikidosec/firewall` to further restrict outgoing requests based on your security

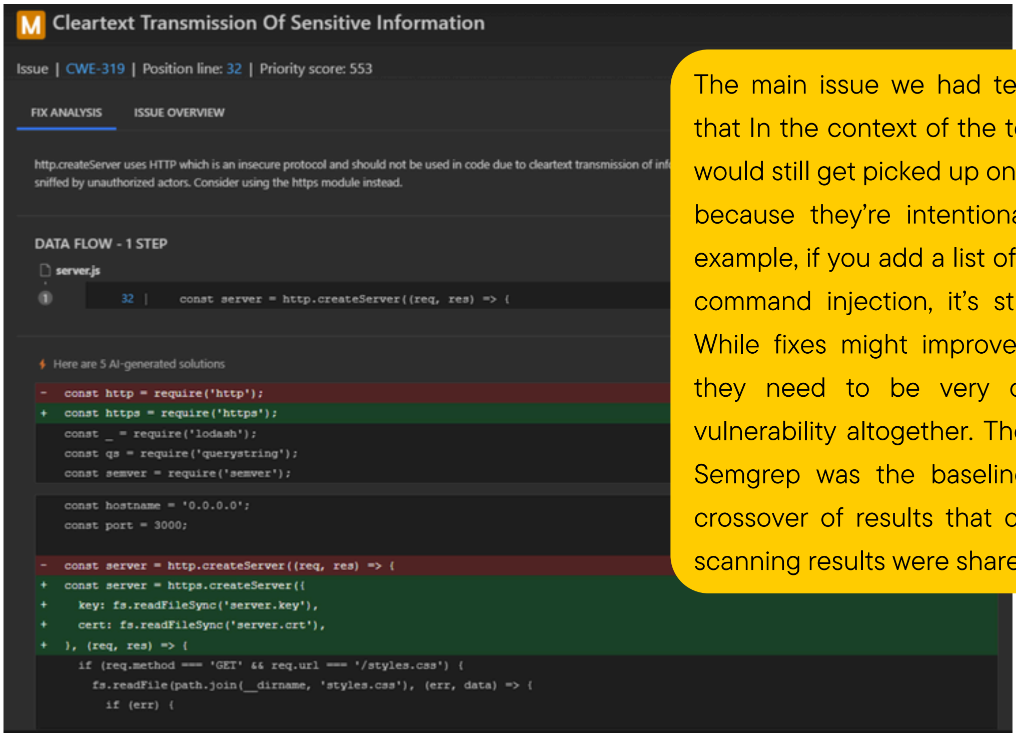
```python

    # 7 - Server-Side Request Forgery (SSRF)
    elif 'url' in request.form:
        url = request.form['url']
        try:
            response = requests.get(url)
            output = f"SSRF Response: {response.text[:200]}"
        except Exception as e:
            output = f"SSRF Error: {e}"

        # 8 - SQL injection with parameter instead of whole query
    if 'username' in request.form:
        username = request.form['username']
```

While some of this summary is negative, there's a lot of potential in the memory feature for teams heavily invested in Semgrep's ecosystem. Semgrep works best for teams that are pretty mature - they know what they're looking for and what the fixes look like. The ability to customize fixes in human language, like "when you're using SQL, use this annotation," will be a great fit for those teams. The greatest challenge with Semgrep is learning the notation for custom rules, and AI does a lot to solve that problem.

snyk

Of all the tests done here, Snyk needs the most qualifiers for two reasons:

1. Snyk only provides a fix if, upon a rescan, the issue is remediated by the fix.
2. Semgrep was the baseline scanner used for testing



M Cleartext Transmission Of Sensitive Information

Issue | CWE-319 | Position line: 32 | Priority score: 553

http.createServer uses HTTP which is an insecure protocol and should not be used in code due to cleartext transmission of inf
sniffed by unauthorized actors. Consider using the https module instead.

DATA FLOW - 1 STEP

server.js

```
1        32 |    const server = http.createServer((req, res) => {
```

Here are 5 AI-generated solutions

```
-  const http = require('http');
+  const https = require('https');
   const _ = require('lodash');
   const qs = require('querystring');
   const semver = require('semver');

   const hostname = '0.0.0.0';
   const port = 3000;

-  const server = http.createServer((req, res) => {
+  const server = https.createServer({
+    key: fs.readFileSync('server.key'),
+    cert: fs.readFileSync('server.crt'),
+  }, (req, res) => {
     if (req.method === 'GET' && req.url === '/styles.css') {
       fs.readFile(path.join(__dirname, 'styles.css'), (err, data) => {
         if (err) {
```

The main issue we had testing Snyk's approach is that In the context of the test repo, most code fixes would still get picked up on a subsequent SAST scan because they're intentionally insecure things. For example, if you add a list of approved commands for command injection, it's still a command injection. While fixes might improve them in a certain way, they need to be very creative to remove the vulnerability altogether. The second challenge, that Semgrep was the baseline scanner, reduced the crossover of results that could be fixed. Only four scanning results were shared between the tools.

Because of the re-testing issue, we can instead talk about the autofixing approach more broadly and qualitatively. From a workflow perspective, Snyk provides five potential AI solutions in the IDE that require clicking through to find the best fit. Typically, the first fix was pretty lacking, like in the above example where the first fix suggested simply changing HTTP to HTTPS, but the fourth fix correctly identifies you'd need a public/private key pair to make it work.

The significant difference with Snyk here is that the AI auto-fix happens entirely inside the developer IDE. Building things this way is pretty risky, in our opinion, due to the struggle of getting developers to adopt IDE extensions. However, it certainly would be the best place to access and apply an auto-fix, and saves on compute costs using cloud-hosted models. Snyk is working on adding the code fixes inside pull requests, which we think will provide a better workflow.

OTHERS

Other great SAST providers weren't included, here is a brief synopsis of why:

Qwiet

We weren't able to get Qwiet tested in time, but will cover them in a re-test

Apiiro

Apiiro doesn't yet provide AI auto-fixes fixes for SAST findings

Cycode

Cycode has AI auto-fix and SAST capabilities, however we couldn't get a test environment in time for this report.

Kodem

Kodem provides runtime SAST with remediation guidance but not code fixes

Backslash

Backslash security has great LLM-generated fix guidance, but it doesn't provide an actual code change

CodeQL

CodeQL didn't have meaningful test coverage, but results can be seen here. Generally was on par with basic LLM solutions.

Thank you!

Thank you for reading our first industry report! We're excited to continue delivering high quality actually useful product assessments that go deeper than any other reports in the industry. Your support is what makes it all possible! Follow our work at https://pulse.latio.tech or browse the full catalogue of vendors at https://list.latio.tech

✉ james@latio.tech
🌐 list.latio.tech
📍 Raleigh, NC

Latio

The Actually Useful Product Guide

Q1 • FEB 2025