
def/acc Hackathon

Assisted Audit of Solana Programs

Yuru Shao
GLAM Systems
yuru@glam.systems

Emanuele Cesena
Jump Crypto
emanuele.cesena@gmail.com

With
Apart Research

Abstract

We present a multi-agent solution that assists in auditing Solana programs, allows to consolidate audit findings into a knowledge base, and can integrate into CI/CD pipelines to prevent security regressions.

Keywords: Multi-agent, AI security, security audit, Solana programs, vulnerabilities.

1. Introduction

The security of smart contracts represents one of the most critical challenges in decentralized finance. Just focusing on the Solana ecosystem (over \$70 billion market cap), many individual programs control tens or even hundreds of millions of dollars in user funds, making them critical software systems where security vulnerabilities can result in catastrophic financial losses.

The problem we address in this work is the automated audit of Solana programs, assisted by a multi-agent AI system. Traditional security audits, while indispensable, represent a snapshot in time of a program's security posture. As programs evolve, teams grow, and new features are added, the security of the overall system can degrade, and new bugs and vulnerabilities are introduced. This creates a fundamental challenge: how can we not only perform effective audits but also maintain security knowledge in the team and prevent regressions over time?

Our approach is centered around the development team, supporting them at three critical stages: proactively identifying vulnerabilities before formal audits, consolidating audit findings into a knowledge base for team learning, and integrating automated checks into CI/CD pipelines to prevent regressions. This

design ensures the system remains easily extensible and maintainable, delivering ongoing value beyond the audit itself.

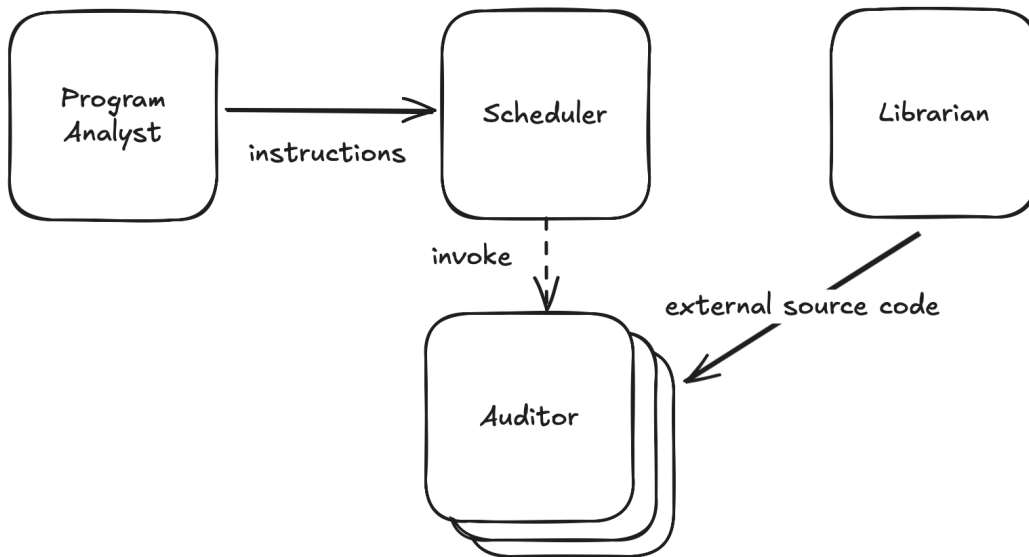
Current LLMs possess several characteristics that make them suitable for this task: they are relatively easy to interact with, can perform basic arithmetic and logic deduction, and are generally good at scouring for information. However, they also have limitations: they struggle with complex mathematics (e.g., cryptography), and they can become confused or stuck in loops when too much information or context is provided.

Our solution takes all of these factors into account, creating a multi-agent system that leverages the strengths of LLMs while mitigating their weaknesses through careful architectural design.

2. Methods

2.1 System Architecture

Our approach employs a multi-agent architecture where each agent is specialized for a specific task. This design philosophy addresses the limitations of LLMs by ensuring that each agent receives a focused, manageable amount of information and context.



Auditor Agents

We created a set of specialized auditor agents, each with a prompt focused on a very specific task. These prompts contain human-readable explanations of how Solana works and frequently found vulnerabilities. Each auditor agent is designed

to examine code for a particular class of security issues or architectural concerns. By specializing each agent, we ensure that:

- The context provided to each agent remains manageable
- The agent can develop deep expertise in its specific domain
- The prompts can be easily maintained and updated as new vulnerability patterns are discovered
- The knowledge base remains human-readable and understandable

Program Analyst Agent

The Program Analyst Agent serves as the entry point to the audit system. Its primary responsibility is to extract all programs and instructions under audit from the target codebase. This agent performs static analysis to identify:

- All program entry points (instructions)
- The structure and organization of the codebase
- Dependencies between different parts of the system
- External program invocations

This analysis provides the foundation for the scheduler to orchestrate the audit process.

Scheduler Agent

The Scheduler Agent orchestrates the audit process by invoking an instance of each auditor agent against every individual instruction in the target program. This creates a comprehensive audit matrix where:

- Each auditor examines each instruction
- The complexity of information each auditor needs to handle is limited to a single instruction at a time
- The audit process is systematic and reproducible

This naive approach of running all agents against all instructions is currently required to limit the complexity of information that each auditor needs to handle. While this may seem computationally expensive, it has proven very effective in practice. It also allows the knowledge base to be maintained in a format that is easy to consume. The approach is relatively trivial to optimize for repeated iterations, for example in a continuous integration system where only changed instructions need to be re-audited.

Librarian Agent

The Librarian Agent maintains references to external codebases, such as other programs that can be invoked by the target program. This is crucial for Solana programs, which frequently interact with external protocols (e.g., Jupiter for swaps, Drift for spots and perpetuals, Marinade for staking, Kamino for lending). The Librarian Agent:

- Maintains a catalog of external programs and their known versions
- Provides auditors with access to relevant external code when needed
- Tracks changes in external dependencies that might affect security

2.2 Evaluation Methodology

To demonstrate and evaluate our methodology, we used the GLAM Systems codebase and its most recent audit. Both the code and audit are currently not publicly available, but they are available upon request.

Agent Development

We built six agents with six prompts, each derived from a specific finding in the GLAM audit report. Each agent was designed to return a simple binary result:

- ok: No vulnerability detected
- err: Vulnerability detected, accompanied by a reference to the line of code and a short description

This binary output format ensures clarity and makes it easy to integrate the results into automated systems.

Test Scenarios

We evaluated each agent using three distinct scenarios:

1. Vulnerable instruction (pre-audit): The original instruction containing the vulnerability identified in the audit
2. Fixed instruction (post-audit): The same instruction after the vulnerability was fixed

3. Non-vulnerable instruction: An instruction that was never vulnerable to the specific issue being tested

This three-scenario approach allows us to verify that agents can:

- Correctly identify vulnerabilities when they exist
- Recognize when vulnerabilities have been fixed
- Avoid false positives on code that was never vulnerable

Extended Testing

After validating the agents on the specific instructions from the audit report, we extended our testing to:

- Other instructions in the GLAM codebase that we knew had similar vulnerabilities
- Instructions from other Solana programs to test generalizability

This extended testing helps validate that the agents can generalize beyond the specific examples they were trained on.

3. Results

3.1 Validation Results

All six agents produced the expected results across all test scenarios:

- Vulnerable instructions (pre-audit): All agents correctly returned "err" with appropriate line references and descriptions
- Fixed instructions (post-audit): All agents correctly returned "ok", indicating they recognized the fixes
- Non-vulnerable instructions: All agents correctly returned "ok", demonstrating no false positives

This 100% accuracy on the validation set demonstrates that the agents can successfully reproduce the findings from the original audit. This is important for example to integrate into a CI/CD to avoid regressions.

3.2 Additional Findings

After building and testing the system on the limited set of instructions from the audit report, we expanded the evaluation to a larger set of instructions from both the GLAM programs and other Solana programs. This broader application led to

the identification of additional vulnerabilities that were not part of the original audit. These findings demonstrate that the system can:

- Scale beyond the original audit scope
- Discover new vulnerabilities through systematic application
- Provide value beyond simple reproduction of known issues

Concretely, in the pre-audit GLAM code base we were able to identify 3 other vulnerabilities. Our team already fixed these vulnerabilities, however this was another validation of the automated approach. One concrete example is given in Appendix B.

Moreover, in the open source code released with this work we have included an example of analysis of the Drift program(s). Our agents, as expected, did not find any vulnerability in Drift main program (drift-protocol-v2), however they spotted a vulnerable program in the same repository -- this program is only used for tests.

3.3 Notable Observations

Some interesting observations emerged during our evaluation.

External Code Versioning

Some agents need to examine external code, as the GLAM program interacts with other open source programs on Solana. We provide agents with a specific, known version of the external code and ask them to check if the code has changed. This approach makes the checks future-proof: if an external dependency is updated in a way that introduces new risks, the agents can detect this change and flag it for review.

Example from an agent:

[...] The correct way to load data from a switchboard oracle account is here:

@https://github.com/switchboard-xyz/solana-sdk/blob/dd1e768/src/on_demand/accouts/pull_feed.rs#L369-L393

First, check the main branch of that git repo and see if the correct way to access switchboard oracle account has changed.

Next, [...]

Cross-Program Deductions

In some cases, agents must perform deductions across multiple programs. For example, when analyzing Drift withdrawals, the agents need to understand:

- How the GLAM program invokes Drift
- What security properties Drift provides
- How the interaction between the two programs could create vulnerabilities

This cross-program analysis is facilitated by the Librarian Agent, which provides access to the relevant external code. An example of this cross-program analysis is presented in the Appendix B.

4. Discussion and Conclusion

Our multi-agent AI system provides a framework for the automated audit of Solana programs. By focusing on specific tasks and maintaining manageable complexity, the system enhances immediate audit outcomes while supporting long-term security maintenance. The approach successfully demonstrates that:

- Audit findings can be encoded into maintainable, human-readable prompts
- These prompts can reproduce audit results with high accuracy
- The system can scale to identify additional vulnerabilities beyond the original audit
- The knowledge base can serve as a valuable resource for teams over time

The system provides a maintainable, extensible foundation that maximizes both immediate audit results and long-term security benefits. As Solana's ecosystem continues to grow and the value locked in its programs increases, tools like this will become increasingly important for maintaining the security and trust that DeFi requires.

Appendix A - Security Considerations

While our results are promising, several limitations and areas for future improvement exist:

1. Coverage: Our current evaluation used six agents based on six audit findings. A production system would need many more agents to achieve comprehensive coverage of all vulnerability classes.

2. False Positives and Negatives: While we achieved full accuracy on our small validation set, we're well aware that a real-world application may reveal edge cases where agents produce false positives or miss subtle vulnerabilities, and even inconsistent results across runs.

3. Performance: The naive approach of running all agents against all instructions may become computationally expensive for very large codebases. Optimization

strategies, such as incremental auditing and parallel execution, need further exploration.

4. LLM Limitations: The system inherits the limitations of the underlying LLMs, including potential issues with complex cryptographic reasoning and context window limitations.

Appendix B - A Full Example

This section details a full example. We show that a vulnerability found in the audit can be reproduced by a sufficiently generic agent, that in turn was able to spot another similar vulnerability.

The GLAM program interacts with the Drift protocol for spot trading. In particular, it's possible to deposit and withdraw from a GLAM vault into a Drift account. When analyzing these operations, we want to guarantee that deposit and withdraw can only move from the corresponding GLAM vault and Drift account, and not to arbitrary destinations, otherwise funds could be exfiltrated.

GLAM had the following vulnerability (found during the audit).

The instruction [withdraw] assumes Drift validates the destination account during withdrawals. However, Drift's withdraw instruction performs no such check, unlike the deposit instruction, which does enforce it.

In plain English, GLAM was calling Drift without any extra check on the destination. While this is ok for deposit because Drift checks the destination, this is not ok for withdraw because Drift would generally allow to withdraw to any account (but GLAM needs to enforce it's a GLAM account).

We built the following Auditor Agent. Note that this agent is generic, it doesn't have any specific knowledge about GLAM nor Drift.

[...] Identify the source and destination token accounts.

The source is generally trusted and even if not explicitly checked it'll fail to move funds when the transaction is executed. The destination must be checked carefully.

In case of a regular instruction invoking transfer or transfer_checked, you need to verify that the authority of the destination is the same as the sender.

In case of CPI into an external program (not the token program), it may or may not be the case that the authority is enforced by the external program. You should try to find the source code of the external program and validate that.

And finally we run the agent against GLAM drift_deposit and drift_withdraw instructions. All tests and corresponding responses are reported below and all responses are as expected.

Interestingly, GLAM has another integration with Marinade for staking and similar deposit and withdraw functions. The agent was able to spot an analogous vulnerability, this time in the marinade_deposit instruction.

Instruction	Result	Explanation
drift_deposit	Ok	GLAM code doesn't perform any check (it's ok because Drift does)
drift_withdraw (before audit)	Err	GLAM code doesn't perform any check - and Drift doesn't either
drift_withdraw (after fix)	Ok	GLAM performs destination check
marinade_deposit (before fix)	Err	GLAM code doesn't perform any check - and Drift doesn't either
marinade_deposit (after fix)	Ok	GLAM performs destination check

We stress that these analyses require the agent to access the external source code of Drift and Marinade, showcasing non-trivial reasoning capabilities.