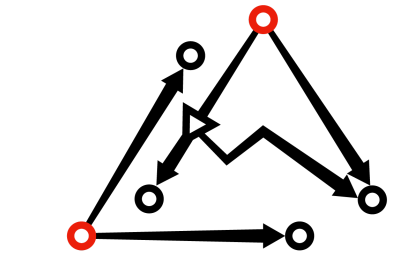

Least Authority

PRIVACY MATTERS

PropAMMRouter

Security Audit Report

LambdaClass

Final Audit Report: 28 August 2026

Table of Contents

Overview

Background

LambdaClass has requested that Least Authority perform a security audit of `PropAMMRouter`, an on-chain router that connects swaps with propAMMs on Ethereum L1.

Project Dates

- **July 15, 2026 - July 22, 2026:** Initial Code Review *(Completed)*
- **July 24, 2026:** Delivery of Initial Audit Report *(Completed)*
- **28 August, 2026:** Verification Review *(Completed)*
- **28 August, 2026:** Delivery of Final Audit Report *(Completed)*

Review Team

- Nikos Iliakis, Security Researcher and Engineer
- Dominic Tarr, Security Researcher and Engineer
- Burak Atasoy, Project Manager
- Jessy Bissal, Technical Editor

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the `PropAMMRouter` followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repository was considered in scope for the review:

- `propamm-router-contracts`: https://github.com/lambdaclass/propamm-router-contracts/tree/contracts/audit/base

Specifically, we examined the following Git revision for our initial review:

- `e413bc931318fcd72a18c892a74db993145d5c98`

For the verification, we examined the following Git revision:

- `38b3b4254218d232724a3d5eff862c01b0dc4d1f`

For the review, this repository was cloned for use during the audit and for reference in this report:

- `lambdaclass-propamm-router-contracts`: https://github.com/LeastAuthority/lambdaclass-propamm-router-contracts

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third-party code, unless specifically mentioned as in scope, were considered out of scope for this review.

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Adversarial actions and other attacks on the network;
- Potential misuse and gaming of the smart contracts;
- Attacks that impact funds, such as the draining or manipulation of funds;
- Mismanagement of funds via transactions;
- Denial of Service (DoS) and other security exploits that would impact the intended use of the smart contracts or disrupt their execution;
- Vulnerabilities in the smart contracts' code;
- Protection against malicious attacks and other ways to exploit the smart contracts;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Our team performed a review of `PropAMMRouter`, an automated market maker that selects the best quote from several AMMs. New AMMs can be added by the `Listing` role after the contract is deployed, but `FermiSwap`, `Kipseli`, and `Bebop` are supported by default, with `UniSwapV3` as the fallback. For certain calls, the user can specify a particular AMM or list of AMMs, but the selection is limited to approved AMM contracts rather than arbitrary contracts. Each AMM is called externally, allowing reverts to be caught and the next AMM to be tried. Calls to external contracts present a reentrancy risk, but `PropAMMRouter` mitigates this risk by calling only approved contracts and using reentrancy checks to the function that performs the direct external call.

System Design

Our team reviewed `PropAMMRouter` and found that it has been well designed, with security at the forefront of its architecture. The router restricts proprietary execution to approved venues, validates deadlines and minimum outputs, measures delivered balances, and falls back to Uniswap when a proprietary venue fails. Public swap functions use reentrancy protection, while isolated external self-calls allow failed venue attempts and their associated token transfers to be rolled back before fallback execution.

The administrative model separates responsibilities among upgrade, emergency, resumption, listing, and administrative roles. Emergency pausing is immediate, while sensitive configuration changes, venue management, upgrades, and resumption are subject to delays.

However, the findings demonstrate inconsistent assumptions across the routing and settlement stages, including differences between quoting and execution (Issue A), net and gross minimum-output requirements (Issue C), and nominal and actual token receipts (Issue B).

Dependencies

The contracts use pinned revisions of OpenZeppelin's upgradeable contracts and Foundry upgrade tooling, `Uniswap V3 Core`, `Uniswap V3 Periphery`, `SwapRouter02`, and `Forge Standard Library`. Our team did not identify any known vulnerabilities in these dependencies during this review.

Code Quality

We performed a manual review of the contracts in scope and found the code to be generally clean, modular, and well organized. Interfaces, errors, events, fee accounting, and Uniswap integration are separated into appropriate files, and the implementation uses established Solidity patterns and libraries.

The code is straightforward to navigate, and descriptive function and variable names facilitate understanding. The use of custom errors, `SafeERC20`, explicit access-control roles, and append-only storage considerations demonstrates adherence to smart contract development best practices.

However, closely related functions do not consistently enforce the same routing and settlement invariants. Quote and swap paths consider different venue sets (Issue A), selected-venue fee swaps apply different minimum-output thresholds during selection and execution (Issue C), and fee-bearing swaps validate receipt by the router rather than final delivery to the recipient (Issue B).

Tests

The repository includes a substantial Foundry test suite. The available tests provide sufficient coverage of the router's basic functionality and several failure paths.

Documentation and Code Comments

The project documentation provides a useful overview of the router. The access-control and design-rationale documents are particularly helpful for understanding the intended architecture.

In addition, the codebase is extensively commented, and most functions contain useful NatSpec descriptions. Comments explaining the rationale at critical points, such as balance-delta accounting, offer valuable implementation context.

Scope

The scope was sufficient to assess the router's security-critical functionality. However, we recommend conducting a focused review of the off-chain infrastructure in the future.

Specific Issues & Suggestions

We list the issues and suggestions identified during the review in descending order of severity. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	SEVERITY	STATUS
Issue A: Swap Functions May Ignore a Better Swap Quote	Medium	Resolved
Issue B: Fee-Bearing Swaps May Report More Output Than the Recipient Receives	Medium	Resolved
Issue C: Selected Venue Fee Swaps Attempt Venues That Cannot Meet the Gross Minimum	Low	Resolved

Issue A: Swap Functions May Ignore a Better Swap Quote

Location

PropAMMRouter.sol#L470

src/PropAMMRouter.sol#129

Synopsis

`quoteV1` and the best-venue swap functions evaluate different venue sets. `quoteV1` calls `_pickBestVenue`, which compares all whitelisted propAMMs and the Uniswap fallback. Conversely, `swapV1` and `swapWithFeeV1` call `_pickBestPropAMM`, which excludes Uniswap whenever a propAMM quote satisfies the caller's minimum output.

As a result, `quoteV1` can report Uniswap as the best venue while `swapV1`, under identical state conditions, executes through a less favorable propAMM.

Preconditions

The issue occurs when Uniswap offers more output than the best available propAMM, while that propAMM still satisfies the caller's `amountOutMin`.

Likelihood

Medium.

The issue occurs only when Uniswap offers a better price while the best propAMM quote still satisfies the caller's `amountOutMin`.

Impact

Medium.

Affected users can receive less output than the router's own quote advertises, resulting in direct economic loss. The loss is bounded by `amountOutMin` and does not threaten protocol solvency or other users' funds. However, repeated suboptimal execution across significant trading volume could produce material aggregate losses.

Severity

Medium.

Technical Details

`quoteV1` calls:

```
(bestQuote, venue) = _pickBestVenue(tokenIn, tokenOut, amount);
```

The `_pickBestVenue` function compares the propAMM result with the Uniswap quote.

In contrast, `swapV1` calls:

```
(uint256 bestQuote, address venue) =
    _pickBestPropAMM(tokenIn, tokenOut, amountIn);
```

Because the propAMM quote satisfies `amountOutMin`, `swapV1` does not select the more favorable Uniswap route.

Remediation

We recommend using the same candidate set for quoting and best-venue execution. In particular, `swapV1` and `swapWithFeeV1` should also call `_pickBestVenue`.

If the additional Uniswap quote cost is unacceptable, these functions should instead be redesigned and documented as propAMM-preferred swaps rather than best-venue swaps. A separate quote function should then be provided whose venue-selection behavior exactly matches that of the corresponding swap function.

Status

The LambdaClass team has acknowledged the finding and documented it as an intended propAMM-preferred routing design.

Verification

Resolved.

Issue B: Fee-Bearing Swaps May Report More Output Than the Recipient Receives

Location

`libraries/FrontendFees.sol#L59`

Synopsis

Fee-bearing swaps enforce the minimum output before transferring the net proceeds from the router to the final recipient. The router subsequently returns the nominal amount transferred without measuring how much the recipient actually received.

If the output token charges a transfer fee or otherwise credits less than the transferred amount, the transaction can succeed even though the recipient receives less than `amountOutMin`. The returned value and `swapped` event also overstate the recipient's actual proceeds.

Likelihood

Medium.

Fee-on-transfer and other short-crediting tokens are less common than standard ERC-20 tokens, but the router accepts arbitrary token addresses and does not document or enforce their exclusion. Once such a token is used, the behavior occurs deterministically.

Impact

Medium.

A successful swap can deliver less than the caller's minimum output while returning and emitting a larger nominal amount. This breaks the router's slippage guarantee.

Severity

Medium.

Technical Details

The fee-bearing entry points direct the underlying swap output to the router before distributing the frontend fee and net proceeds. `_coreSwap` measures the router's balance increase and verifies it against `grossMin`, correctly accounting for deductions during the venue-to-router transfer.

`FrontendFees._skimAndDisburse` then calculates the nominal fee and net amounts and transfers the net amount to the recipient. However, it does not measure the recipient's balance change during this final transfer and returns the nominal net amount instead.

Consequently, the initial balance-delta check cannot detect deductions during the router-to-recipient transfer. The transaction can therefore succeed and report the nominal net output even when the recipient's actual receipt is below `amountOutMin`.

Remediation

We recommend measuring the recipient's actual balance increase during the final payout and enforcing `amountOutMin` against that value.

Status

The LambdaClass team has resolved the issue.

Verification

Resolved.

Issue C: Selected Venue Fee Swaps Attempt Venues That Cannot Meet the Gross Minimum

Location

`PropAMMRouter.sol#L263`

Synopsis

The selected-venue fee swap applies inconsistent minimum-output requirements during venue selection and execution. As a result, the router may select a venue that cannot provide enough output to cover both the user's requested amount and the frontend fee.

Execution through the selected venue consequently fails, after which the router uses the Uniswap fallback. Although the transaction can still settle safely, the unnecessary venue attempt increases gas consumption and creates avoidable routing divergence.

Preconditions

For this issue to occur, the swap must charge a nonzero frontend fee, and at least one caller-selected venue must return a valid quote.

Likelihood

Medium.

The behavior occurs deterministically whenever the selected quote falls within the affected interval.

Impact

Low.

The caller incurs additional gas cost for a venue attempt that cannot satisfy the execution requirement. Although the router subsequently attempts the Uniswap fallback, the entire transaction reverts if that fallback fails. In particular, gas consumed by the unnecessary venue call may leave insufficient gas for an otherwise executable fallback. The issue does not cause token loss.

Severity

Low.

Technical Details

`amountOutMin` is the net amount the recipient must receive after the frontend fee. The router therefore calculates the higher gross requirement, including the frontend fee:

```
uint256 grossMin = FrontendFees._grossUp(amountOutMin, fee.bps);
```

However, `swapViaSelectedVenuesWithFeeV1` compares the selected venue's gross quote against the net minimum:

```
if (venue == address(0) || bestQuote < amountOutMin) {
```

It then executes the selected venue using `grossMin`:

```
(delivered, executedVenue) = _coreSwap(
    …,
    grossMin,
    … );
```

The venue passes selection because its quote satisfies the net minimum, even though it is below the gross minimum required during execution. If the venue delivers its quoted amount, the call fails, and `_coreSwap` attempts the Uniswap fallback.

In that case, the venue call is rolled back, but its gas is consumed. If the fallback also fails, for example because insufficient gas remains, the entire swap reverts. By contrast, `swapWithFeeV1` compares the quote against `grossMin` and routes directly to the fallback.

Remediation

We recommend using `grossMin` consistently during venue selection and execution.

Status

The LambdaClass team has resolved the incorrect minimum comparison.

Verification

Resolved.

About Least Authority

Founded in 2011 and based in Berlin, Germany, since 2016, Least Authority provides security consulting for privacy-preserving technologies and decentralized systems. We are committed to transparency, openness, and advancing secure digital infrastructure.

We believe that people have a fundamental right to privacy and that secure solutions enable people to use the internet and other connected technologies more freely. Our security consulting services help teams make their solutions more resistant to unauthorized access and unintended system manipulation. We support teams from the design phase through production launch and beyond.

Our team has experience reviewing code for security vulnerabilities and specific attack vectors in multiple languages, including Rust, Go, Python, C, C++, Solidity, Cairo, Circom, Noir, Haskell, OCaml, TypeScript, Swift, ZoKrates, and others. We have reviewed implementations of cryptographic protocols and distributed system architecture in blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. Additionally, we use various tools to scan code and networks and build custom tools as necessary.

For more information about our security consulting, please visit:
https://leastauthority.com/security-consulting.

Our Methodology

We use a transparent review process that provides the project team visibility into our findings, assumptions, and remediation considerations throughout the audit. The goals of our security audits are to improve the quality of the systems we review and support sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Code Review

In manually reviewing all of the code, we identify potential issues, including those related to code logic, error handling, cryptographic implementations, and correctness with respect to specifications, standards, and best practices. We also evaluate areas where more defensive programming could reduce the risk of future mistakes and accelerate future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

When appropriate, we also use artificial intelligence (AI) to support the efficiency of our security audits without reducing rigor or reliability. AI-assisted outputs are carefully guided, reviewed, and validated by our team so that conclusions remain accurate and defensible. This supports stronger audit decisions, not only faster review.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and white box penetration testing, supplemented by various tools, including AI, that support the code review process. Discussions with the development team help clarify their vision for the software. Installing and using the relevant software helps us understand user interactions and roles, while also informing our analysis of threat models and attack surfaces. Design documentation, relevant papers, other audit results, similar projects, source code dependencies, and open issue tickets may also be reviewed to inform the analysis. We

identify potential vulnerabilities, and, when appropriate, create issue entries. For each entry, we follow the issue investigation and remediation process described below.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and supporting successful remediation. Whenever a potential issue is identified, we raise it with the client through a dedicated secure communication channel to discuss its validity and impact, to the extent these can be determined at that stage. This process allows potentially relevant findings to be reviewed with the client early, before they are documented in greater detail in the initial audit report. For each confirmed issue, we assess impact and feasibility and use those ratings to determine the overall severity, following the OWASP Risk Severity Matrix.

Suggested Solutions

We identify immediate and comprehensive mitigations available to live deployments and then suggest remediation requirements for future releases. When code is already deployed in production, we include mitigation strategies in the report to provide interim support until appropriate remediation is available. The mitigation and remediation recommendations should be reviewed by the development team and deployment engineers. After delivering the Initial Audit Report, we remain available to support implementation of the proposed mitigations and remediations before the verification review.

Development teams should address findings in an order and with an urgency that prioritizes the security of their users. Findings should be addressed as soon as possible, but remediation should be carried out pragmatically and should not cause unnecessary disruption to project timelines.

Although issues must be handled on a case-by-case basis, we work toward a resolution timeline that balances the impact on users and the needs of the project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess whether the issues and suggestions have been sufficiently addressed. When this analysis is complete, we update the report and provide a Final Audit Report that can be published in full. If any critical issues remain, we recommend that the Final Audit Report not be published and that users and other stakeholders be alerted to the impact. We encourage all findings to be addressed and the Final Audit Report to be shared publicly to support transparency and advance security knowledge within the industry.