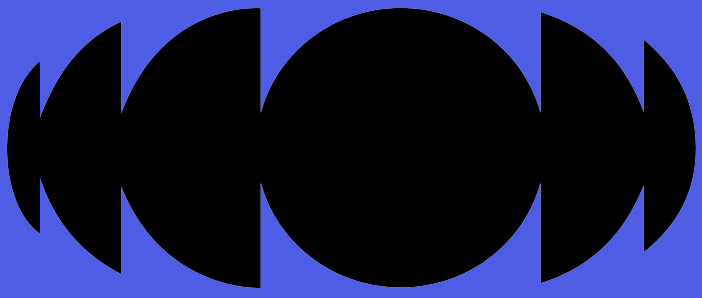
OpenZeppelin | security

# 9Lives CPMM Audit



September 19, 2025

# Table of Contents

# Summary

| | |
|---|---|
| Type | DeFi |
| Timeline | From 2025-08-11<br>To 2025-09-03 |
| Languages | Rust |

| | |
|---|---|
| Total Issues | 16 (9 resolved, 1 partially resolved) |
| Critical Severity Issues | 0 (0 resolved) |
| High Severity Issues | 2 (0 resolved, 1 partially resolved) |
| Medium Severity Issues | 6 (4 resolved) |
| Low Severity Issues | 4 (3 resolved) |
| Notes & Additional Information | 4 (2 resolved) |

# Scope

OpenZeppelin audited the fluidity-money/9lives repository at commit 5e90e7b.

In scope were the following files:

```
src
├── contract_trading_extras.rs
├── contract_trading_mint.rs
├── contract_trading_price.rs
├── contract_trading_quotes.rs
├── contract_trading.rs
├── fees.rs
├── maths.rs
├── storage_trading.rs
├── trading_amm.rs
├── trading_private.rs
└── Share.sol
```

# System Overview

9lives is a prediction market deployed on the Superposition blockchain. The protocol allows for the creation of markets where users can buy and sell positions based on the perceived likelihood of a future event happening, such as the results of an election. It includes an order book and an AMM feature, with this audit exclusively focusing on the AMM aspect.

The AMM-trading contract is an Arbitrum Stylus smart contract implemented with a factory/pair pattern. A factory contract takes a list of outcomes and creates a minimal viable proxy for the trading contract pointing to several implementation facets. It also deploys an ERC-20 contract for each outcome representing shares. In addition, a trading contract, `contract_trading.rs`, directly adds `user_entrypoint` to the contract facet being selected with its corresponding feature in the Rust crate (bypassing the usual `#[entrypoint]` macro logic with one Stylus contract per crate). Each facet implementation adds the following functionality:

- **`contract_trading_extras`**: Allows the market oracle to decide the market outcome.
- **`contract_trading_mint`**: Allows users to mint or burn ERC20 shares which correspond to a particular market outcome.
- **`contract_trading_price`**: Allows users to add or remove market liquidity, claim fees, and rebalance outcome prices.
- **`contract_trading_quote`**: Allows for retrieving the estimated amounts for minting and burning the given outcome shares, as well as allowing users to claim their payoff when a market is resolved.

Each market is configured with an oracle role which is an account that determines the resolution of the market among a variable number of outcomes. Liquidity providers deposit collateral into the AMM, which prices shares based on existing reserves, allowing users to buy or sell outcome tokens during active markets. The AMM dynamically adjusts share prices as trades occur, ensuring that the relative likelihood of each outcome is reflected in market prices, while liquidity providers earn fees proportional to their contribution.

# Security Model and Trust Assumptions

During the audit, the following trust assumptions were made:

- The resolution of markets is decided by a trusted oracle. In extreme scenarios, such as oracle failure or exploits, operator privileges allow funds to be removed through a rescue function. Shares cannot be redeemed until the oracle has decided on the resolution, and if no resolution takes place, then the shares cannot be redeemed for fUSDC tokens. Only one outcome can be set as a winner and the contracts do not support split winners.
- A paymaster or claimant helper can execute transactions on behalf of users. Hence, they are a trusted party which can control user funds. Transaction submitted through the paymaster or claimant helper can only be sent with the original sender as the recipient.
- The contract is deployed via a factory and this must be trusted to set reasonable parameters and not set excessive fees. The system rounds in favor of the protocol when calculating fees.

## Privileged Roles

The market oracle, which is set during market initialization, sets the winning outcome to resolve the market. The `DAO_OP_ADDR` is able to extend the market ending time arbitrarily or transfer out all fUSDC tokens from a trading contract via the `rescue` function.

# High Severity

## H-01 Lack of Slippage Checks When Minting and Burning Outcomes

The `mint_8_A_059_B_6_E` and `burn_854_C_C_96_E` functions are missing slippage checks and, therefore, trades can execute at unintended prices. Even if an attacker cannot influence the order of operations in a block (e.g., to perform a sandwich attack), without a slippage check in the contract, it becomes impossible for users to specify trades that only go through within a desired price range.

The `min_shares` parameter in the `burn_854_C_C_96_E` function does not work for slippage control because a trader will get a worse price if they are burning a larger amount of shares for a fixed amount of `fUSDC`. A "maximum shares" parameter should be used instead, while a "minimum shares" parameter should be used in the `mint_8_A_059_B_6_E` function to ensure that a trader receives at least a minimum number of shares when buying.

Consider adding the appropriate slippage parameters to both the `mint_8_A_059_B_6_E` and `burn_854_C_C_96_E` functions to give users proper control over trade execution.

***Update:*** *Partially resolved in pull request #28. The team stated:*

> *We added controls to the BuyHelper2 contracts to enforce restrictions. We'll add something similar to Paymaster once h-02 has been reviewed.*

## H-02 Shares Cannot Be Claimed in Unresolvable Market

A market may become unresolvable in cases when there are ties, incorrectly specified or unforeseen outcomes, or when an event that the resolution of a market depends on never actually happens. In such a situation, the market will eventually end, but a winner can never be decided, and shares cannot be burned or claimed as a winning payoff, leaving users with shares that can never be redeemed for fUSDC.

Consider implementing additional logic for the oracle to mark unresolvable markets as void, and allowing users to redeem their shares at the market price.

***Update:*** *Acknowledged, will resolve. The team stated:*

> *We perceive this issue as being a product issue that should be addressed in the frontend experience, as opposed to a contract change. The infra market contains a check for the outcome being zero, which can be used to indicate another round of voting, which can be used to continue voting until resolution is made. The DPM needs to be used with multiple two-sided markets combined together, with each individual market having a "no" state being the second outcome. So we feel that this is a problem we should address with better outcome design on behalf of the user.*

# Medium Severity

## M-01 Perpetual Extension of Market Ending Time

When an outcome share is minted within 3 hours of the ending time, the ending time of the market is extended by 3 hours. This occurs regardless of how many times it has been extended before. Therefore, an attacker could perpetually delay the finalization of the market and the redeeming of shares by minting a small amount every 3 hours. Even without a malicious user, the finalization of markets becomes extremely unpredictable in a market where frequent minting occurs.

Consider extending each market's ending time only once in case an outcome is minted within the buffer period.

***Update:*** *Acknowledged, will resolve. The team stated:*

> *We perceive this feature as being something we need to gauge in practice, as this is used for the "beauty contest" type of outcome, and exists as a low effort protection mechanism for last minute providing of liquidity to the market. We plan to leave this in the contract for now, and will make adjustments based on how our users use the feature.*

## M-02 Adding Liquidity Can Be Sandwiched Despite the `min_shares` Parameter

In the `add_liquidity_638_E_B_2_C_9` function, the formula virtually mints outcomes that are equal to the deposited fUSDC tokens for all outcomes, deposits the most likely outcome,

and adds the corresponding outcomes to the liquidity pool to maintain a constant token ratio. While the depositor receives the outcome shares for each outcome, the least likely outcome has no excess shares deposited.

The maximum amount of shares are always minted when the pool has an exactly balanced token ratio with equal amounts of each outcome. For example, even if the correct pool ratio is `4:1`, more liquidity shares are minted when the token ratio is `1:1.` However, this balanced ratio results in lower liquidity-per-dollar (a loss for the depositor) compared to the correct ratio, where the pool ratio reflects the value and likelihood of each outcome.

As such, specifying a minimum amount of liquidity shares does not enforce that a liquidity deposit will be minted at the most cost-effective ratio.

Consider enforcing slippage protection with an array of minimum or maximum outcomes during liquidity addition. These would be the minimum or maximum outcome shares that are minted to the user.

***Update:*** *Resolved in pull request #29. Consider adding a check in the external add liquidity function to ensure that* `max_liquidity >= min_liquidity` *to fail early in case of incorrectly specified parameters. The team stated:*

> *We added a check for the maximum amount of shares to receive using the add liquidity function. We did not add a check for each share that's been sent (there is an issue in this SDK release where we can't encode a vector, so it would be a complex fix if it's not necessary).*

## M-03 Wrong Use of `msg_sender` When Claiming Fees

The `internal_amm_claim_lp_fees` function uses `msg_sender()`, whereas, it should be using the `sender` parameter. It deducts liquidity shares from `msg_sender` but uses the `spender` parameter for claimed fees, updates, and logging. This breaks fee claims when the claimant or paymaster sends the transaction, silently returning zero fees without errors. This behavior can also be exploited to claim undue fees by privileged roles. The claimant or paymaster could create a large LP position to accumulate fees, and then claim fees with a dummy LP account as the recipient, effectively redirecting the fees intended for their main position.

Consider consistently using the `sender` parameter throughout `internal_amm_claim_lp_fees` instead of `msg_sender`.

***Update:*** *Resolved in pull request #30. The team stated:*

> *We've made a fix we hope resolves the issue!*

## M-04 Liquidity Removal Can Revert Due to Underflow

In the `rebalance_fees` function, the fee weight is calculated using ceiling division. This fee weight is added to the user's collected amount when they are adding liquidity and subtracted from their fee weight when removing liquidity.

The use of ceiling division correctly rounds against the user when adding liquidity, but in the context of removing liquidity, it actually rounds in favor of the user. The fee weight is overestimated by the ceiling division. If the user is removing all of their liquidity, then the fee weight can exceed the `amm_lp_user_fees_claimed` value, which would revert due to the use of checked subtraction.

Consider rounding down when calculating fee weight for liquidity removal while still rounding up when calculating fee weight during liquidity addition.

***Update:*** *Resolved in pull request #32. The team stated:*

> *We changed the behaviour to round down when removing liquidity.*

## M-05 Mint Fee Can Be Circumvented in Imbalanced Markets

The `mint_8_A_059_B_6_E` function allows users to exchange fUSDC tokens for one of the outcome shares. A fee is charged to the minter in proportion to the amount of shares they mint. However, no fees are charged for adding and removing liquidity or burning outcome shares. As such, a trader could mint shares while not paying any fees by using a combination of the above three actions.

For example, consider an imbalanced 2-outcome market `(X, Y)` with less `X` shares than `Y` shares. A user adds liquidity and gets shares of outcome `X` in addition to their liquidity (because `X` is not a member of the least-likely outcome set). The user then removes liquidity and gets `Y` shares and some amount of fUSDC back. If the user wants to bet on `X`, they will burn all the `Y` outcome shares they received to get back fUSDC, and then they end up with purely `X` outcome shares. This series of steps mints `X` shares with the same price impact as

minting directly. However, the user avoids paying a fee and therefore gets the `X` outcome shares for a cheaper price, while liquidity providers and other fee recipients get shortchanged.

Consider charging a fee for burning as well as minting shares. Since each burn operation makes a swap through the AMM, it is fair to charge a fee for burning and not just minting outcomes.

***Update:*** *Resolved in pull request #36. The team stated:*

> *We've added back the selling feature.*

## M-06 Liquidity Cannot Be Removed Between Ending Time and Market Decision

The `is_not_done_predicting` function returns `false` when the current timestamp is later than the market ending time, even when the market has not been decided via the `decide` function. Hence, during such a period when the market has ended but not decided, removing liquidity will execute `internal_amm_claim_liquidity`, which will revert. This causes liquidity providers to be indefinitely unable to retrieve their funds from the market during this time window.

Consider modifying the conditions such that `internal_amm_claim_liquidity` is only ever called when the market outcome has been decided.

***Update:*** *Acknowledged, not resolved. The team stated:*

> *We feel that this is a feature that could be useful to prevent fraudulent situations, and we haven't noticed a product need from our users so far to change it. We'll potentially revisit this behaviour in the future, but for now, we won't implement a change.*

# Low Severity

## L-01 Incorrect Order of Checks in `Rooti`

The first check in the `rooti` function returns `0` when `x` is `0`, as usually the root is `0` whenever the `x` parameter is zero. However, if `n` is also zero, then the answer is undefined and the function should revert.

Consider putting the check which reverts when `n` is zero before the check which returns early when `x` is zero.

***Update:*** *Resolved in pull request #33. The team stated:*

> *This code reorders the check.*

## L-02 `shutdown` Function Does Not Affect State

The `shutdown` function can only be called when the market has already ended, which means that the `is_not_done_predicting` check already returns `false`. Since there is no check for whether a market has shutdown aside from within the `is_not_done_predicting` check, calling `shutdown` does not make any difference in which functions can be successfully called. In addition, shares can only be claimed when the oracle decides the outcome, and the `decide` function also calls `internal_shutdown` which closes the market.

Consider either removing the external `shutdown` function or changing it so that it restricts some contract functionality.

***Update:*** *Acknowledged, will resolve. In its current state, the `shutdown` function is not to the detriment of the rest of the contract functionality. The team stated:*

> *We'll redesignate it's purpose informally to be a place for external contract methods to be called when a market is concluded or expired. This is the current behaviour for the DPM version of the contract.*

## L-03 Unused Fee for Minter Is Accounted for in Total Fee

A `fee_for_minter` is included the calculation of the total minting fee, but is not allocated to any address to later claim in `calculate_and_set_fees`. This leads to a portion of the overall fees being stuck in the contract, and the only way to retrieve it is via the `rescue_2_7_6_D_D_9_A_B` function.

Consider removing `fee_for_minter` and any associated logic from the codebase to avoid the above-described scenario.

***Update:*** *Resolved in pull request #34. The team stated:*

> *We removed the fee for minter from calculations.*

## L-04 Ending Time Can Be Set in the Past

The operator can call `extend_time` to update the `time_ending` of a market, but there are no checks to ensure that the new time is later than the current time as in the constructor. The operator can thus inadvertently end a market by setting a `time_ending` value that is earlier than the current time.

Consider validating that the new ending time is greater than the old one in `extend_time` to avoid unexpected changes to an active market.

***Update:*** *Resolved in pull request #31. The team stated:*

> *We implemented a fix with a new error type.*

# Notes & Additional Information

## N-01 Lack of Security Contact

Providing a specific security contact (such as an email address or ENS name) within a smart contract significantly simplifies the process for individuals to communicate if they identify a vulnerability in the code. This practice is quite beneficial as it permits the code owners to dictate the communication channel for vulnerability disclosure, eliminating the risk of miscommunication or failure to report due to a lack of knowledge on how to do so. In addition, if the contract incorporates third-party libraries and a bug surfaces in those, it becomes easier for their maintainers to contact the appropriate person about the problem and provide mitigation instructions.

Consider adding a security contact in all the contracts throughout the codebase.

***Update:*** *Acknowledged, will resolve. Existing security instructions are contained in markdown file. The team stated:*

> *We are unable to verify our deployment on-chain using explorers (wasm-opt is in the pipeline).*

## N-02 Unused Seller Fee Logic

The `calculate_fees` function includes an `is_buy` parameter and logic to calculate a fee for selling. However, the `is_buy` parameter and the associated logic are not used during any operations in the contract since seller fees are never applied.

Consider fully implementing the necessary logic to charge seller fees.

***Update:*** *Resolved in pull request #36. The team stated:*

> *We've made use of the selling fee now using the estimate burn code.*

## N-03 Incorrect Error Message When Market Already Decided

The `internal_decide` function throws a `NotTradingContract` error when the market has already been decided. However, this error is intended to be used under different circumstances.

Consider throwing a more accurate error message in the aforementioned case, such as `DoneVoting`.

***Update:*** *Resolved in pull request #35. The team stated:*

> *We changed the error type!*

## N-04 Unused `is_escaped` Flag

The `is_escaped` flag exists in storage along with a function for the oracle to set it. However, this flag is not used anywhere in the codebase.

To improve overall clarity and maintainability of the codebase, consider removing the unused variable and its associated setter function.

***Update:*** *Acknowledged, will resolve. The team stated:*

> *We see this feature evolving gradually as we understand better how to handle erroneous situations, including as we add extra oracle resolving backends. We'll leave the flag in for now, with an eye for utilising it later.*

# Conclusion

The 9lives AMM facilitates the trading of prediction market shares through liquidity provisioning and pricing outcomes via buying and selling. It allows users to buy or sell shares during active markets while reflecting market-implied probabilities, and rewards liquidity providers through trading fees.

Overall, the system combines automated pricing with decentralized access to create an efficient and transparent prediction market via a unique implementation using the Arbitrum Stylus SDK. The codebase was found to be well structured and well tested, reflecting a strong commitment to code quality and maintainability. Several high-severity issues were uncovered during the audit, primarily related to a need for better slippage protection on user trades, as well missing logic to handle unresolvable markets.

The Fluidity Labs team is appreciated for their prompt and thoughtful responses to all the questions asked about the protocol during the audit.