Security Assessment Final v2 Report

June 2025

Prepared for Heaven

Table of contents

Project Summary

Project Scope

Project Name	Repository (link)	Audited Commits	Platform
Heaven AMM	https://github.com/heavenxyz/heaven-amm	996ba64	Solana
Heaven AMM	https://github.com/heavenxyz/heaven-amm	20d6e9d	Solana

Project Overview

This document describes the findings obtained during the manual code review of **Heaven AMM.** The work was undertaken in 2 separate phases:

- **June 2nd** to **June 12th:** commit 996ba640a48af3bdc55a16b86913761282291c93
- **June 16th** to **June 27th:** commit 20d6e9d2f5b545af69d20d19cdaeac499b5809e7

At the request of the client, these two phases have been consolidated into one report.

The scope covers all files within the following folder structure:

```
{programs/heaven-anchor-amm/src/...}
```

Protocol Overview

Heaven AMM is a decentralized exchange protocol on Solana aiming to facilitate fair token launches through a constant product market maker (x*y=k) for token pair liquidity pools and sandwich resistance mechanisms.

It operates with a dual function as both a token launchpad and a trading venue with customizable tokenomics support. Technically, the system implements dynamic fee structures based on market capitalization calculations, with bracket-based fee tiers. The protocol classifies

tokens using a TaxableSide taxonomy to determine appropriate execution paths for different trade types (Buy, Sell, Swap). It implements mathematical protections against MEV through sandwich resistance algorithms that detect price manipulation attempts and implements virtual orders to mitigate front-running attacks.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	2	2	2
Medium	4	4	4
Low	3	3	1
Informational	12	12	3
Total	21	21	11

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
			Likelihood	

Detailed Findings

ID	Title	Severity	Status
H-01	claim_standard_creator_trading_fees does not check if the trading volume has been reached	High	Fixed
H-02	Slot_offset fees cannot be enabled for protocol_trading fee and liquidity_provider fee	High	Fixed
M-01	create_liquidity_pool_standard can overspend from payer without explicit cap	Medium	Fixed
M-02	update_protocol_config does not check valid fee config	Medium	Fixed
M-03	Inconsistent rounding can lead to edge cases where the swap returns tokens > max_limit, causing a revert	Medium	Fixed
M-04	amount_in calculation does not consider fees, causing users to be unable to buy up to maximum limit	Medium	Fixed
L-01	Updating supported_pool_type will lead to corrupted pools	Low	Fixed
L-02	admin_mint_msol does not limit staking up to any % of available liquidity, which can break all pools	Low	Acknowledged
L-03	admin_unstake_msol breaks if yield is lower than historical cost	Low	Acknowledged

High Severity Issues

H-01 claim_standard_creator_trading_fees does not check if the trading volume has been reached		
Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: swap.rs#806	Status: fixed in f208d	

Description:

The protocol pool configuration includes a field called `creator_trading_fee_trading_volume_threshold`, which defines the minimum cumulative trading volume (in USD) that must be reached before a pool creator becomes eligible to claim their `creator_trading_fee`.

Trading volume is incremented using the `increment_trade_volume_usd` function, which is called during buy and sell operations. When the threshold is crossed, the `creator_trading_fee_trading_volume_threshold_reached_unix_timestamp` field is updated.

However, this updated value is **never checked** during fee claiming in either of the claim functions:

- `claim_standard_creator_trading_fees`
- `admin_claim_standard_creator_trading_fees`

This contradicts the intended behavior as described in the sponsor system documentation, which states:

> *"claim_standard_creator_trading_fee / admin_claim_standard_creator_trading_fee allows the creator of a standard liquidity pool to claim their trading fees only if these*

> *conditions are met: pool has equal to or more than $250K in trading volume (buy/sell)…"*

As a result, fees may be claimed even if the required trading volume threshold has not been reached, violating a key system invariant.

Recommendations:

It is recommended in the functions `admin_claim_standard_creator_trading_fees` and `claim_standard_creator_trading_fees` to implement a check to ensure that the required trading volume is reached.

Customer's response: fixed in f208d

Fix Review: The issue has been resolved for the non-admin function, which now only unlocks when the defined threshold has been reached. For the admin function the protocol has chosen to explicitly not set this threshold. This does conflict with the project documentation.

H-02 Slot_offset fees cannot be enabled for protocol_trading fee and liquidity_provider fee		
Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: protocol.rs#2237	Status: fixed in 5720	

Description:

The `slot_offset_trading_fees` are set through the admin function `set_protocol_slot_fees`.

Once set, whenever calculate_fee is called, it will check if `slot_offset_based_fee.is_enabled()` to determine whether to apply market or slot fees.

```rust
if slot_offset_based_fee.is_enabled()
        && slot_offset <= slot_offset_based_fee.max_slot_offset as u64
{
        slot_offset_based_fee.get_fee(slot_offset, trade_type, amount_in)
} else {
        market_cap_based_fee.get_fee(current_market_cap, trade_type, amount_in)
}
```

However, 2 types of slot fees, the `set_slot_protocol_trading_fee` and the `set_slot_liquidity_provider_trading_fee`, are not enabled when they are set.

```rust
    pub fn set_slot_protocol_trading_fee(
        &mut self,
        mut brackets: SlotFeeBracketsParams,
    ) -> &mut Self {
        brackets.sort_brackets();
        brackets.assert_valid();
        self.slot_offset_based_fees.protocol_trading_fee.count = brackets.count;
        self.slot_offset_based_fees.protocol_trading_fee.max_slot_offset =
brackets.max_slot_offset;
        self.slot_offset_based_fees.protocol_trading_fee.max_fee_bps = brackets.max_fee_bps;
        self.slot_offset_based_fees.protocol_trading_fee.brackets[..brackets.count as usize]
            .copy_from_slice(&brackets.brackets[..brackets.count as usize]);
        self
    }

    pub fn set_slot_liquidity_provider_trading_fee(
        &mut self,
        mut brackets: SlotFeeBracketsParams,
    ) -> &mut Self {
        brackets.sort_brackets();
        brackets.assert_valid();
        self.slot_offset_based_fees.liquidity_provider_trading_fee.count = brackets.count;
        self.slot_offset_based_fees.liquidity_provider_trading_fee.max_slot_offset =
brackets.max_slot_offset;
        self.slot_offset_based_fees.liquidity_provider_trading_fee.max_fee_bps =
brackets.max_fee_bps;
        self.slot_offset_based_fees.liquidity_provider_trading_fee.brackets[..brackets.count
as usize]
            .copy_from_slice(&brackets.brackets[..brackets.count as usize]);

        self
    }
```

As a result, even though the admins have set the `slot_fees`, the market fees will always be applied for these 2 fee types.

Recommendations:

Add the lines:

```
Rust
self.slot_offset_based_fees.protocol_trading_fee.enabled = brackets.enabled;
self.slot_offset_based_fees.liquidity_provider_trading_fee.enabled = brackets.enabled;
```

Customer's response: fixed in 5720

Fix Review: The issue has been resolved.

Medium Severity Issues

M-01 create_liquidity_pool_standard can overspend from payer without explicit cap		
Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: [mod.rs#232]	Status: fixed in [6def]	

Description:

In the `create_liquidity_pool_standard` function, both the `user` and the `payer` are required to sign. While these are often the same entity, valid use cases exist where:

- The `user` is a smart wallet or multisig managing the pool,
- The `payer` is a third-party sponsor covering account creation and initial SOL contribution.

The function allows the user to specify an `initial_purchase_amount`, representing the number of tokens to be bought without fees during pool initialization. However, the SOL input required (`amount_in`) is calculated dynamically based on:

- The protocol's current `target_market_cap`, which is admin-controlled, and
- The live SOL/USD price, which may change with market conditions.

The calculated `amount_in` is pulled directly from the payer's balance, not the user's.

```rust
    pub fn from_target_market_cap(
        initial_base: u64,
        target_market_cap: f64,
        current_sol_price: f64,
        leader_slot_window: u8,
    ) -> Self {
        let initial_quote =
            Self::calculate_quote_reserve_by_market_cap(target_market_cap,
current_sol_price);
```

```
        let mut reserve = Self {
            token_a: initial_base,
            token_b: initial_quote,
            leader_slot_window,
            initial_a: initial_base,
            initial_b: initial_quote,
            ..Default::default()
        };
        reserve.snapshot();
        reserve
    }
    ...
         let amount_in = TokenSwapCalculator::<ConstantProduct>::calculate_swap_exact_out(
         U128::from(initial_purchase_amount),
         &SwapDirection::TokenB2TokenA,
         token_a_reserve,
         token_b_reserve,
    )
    .as_u64();
```

Once the `payer` signs the transaction (or associated instructions), the user can delay execution, waiting for conditions that increase the cost to the payer. For example:

- Waiting for the SOL price to drop, increasing how much is required to match the target market cap.
- Waiting for the admin to update `target_market_cap`, affecting how much token B reserve is needed.

Since `amount_in` is computed only at execution time, the payer has no guarantee that the cost will match what they signed off on.

Recommendations: Introduce a max_amount_in parameter to explicitly cap the amount of SOL the payer is willing to contribute.

Customer's response: fixed in 6def

Fix Review: The issue has been resolved.

M-02 update_protocol_config does not check valid fee config

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: protocol.rs#526	Status: fixed in f73d	

Description:

In `create_protocol_config`, we can see that fee configuration is checked through a call to `assert_valid_fees_config`.

However, in `update_protocol_config`, this assert is missing. This can lead to a situation where an update protocol call with new fees could exceed 100%, thereby breaking the entire protocol.

Recommendations:

Include the below assert after the state has been updated:

```rust
state.assert_valid_fees_config();
```

Customer's response: fixed in f73d

Fix Review: The issue has been resolved.

M-03 Inconsistent rounding can lead to edge cases where the swap returns tokens > max_limit, causing a revert

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: swap.rs#112	Status: fixed in 8d07	

Description:

The protocol has 2 functions that return an amount required to buy a certain amount of tokens.

- `calculate_swap_exact_out` (uses `CEILING` rounding for the denominator)
- `calculate_swap_exact_in` (uses `FLOOR` rounding for the denominator)

Both are legitimate ways of calculating the required amount, but due to the difference in rounding, the amounts returned will be slightly different.

In the buy function, assuming `max_sol_spend` is larger, the `amount_in` to purchase the maximum amount of tokens is calculated using the first method which uses `Ceiling` rounding.

However, in the `swap_exact_in` function, the actual transfer of tokens is done through the `calculate_swap_in` function, which uses `Floor` rounding.

Due to this, in the edge-case where there are no fees, this can cause the `user_token_a_vault_amount` to exceed the `max_supply_per_wallet` by 1 Token.

```rust
Example:
User balance= 0 Tokens
Max_supply_per_wallet = 1000 Tokens
calculate_swap_exact_out = 2.8754 SOL is required to buy 1000 Tokens.
NOTE: the amount is slightly overestimated due to Ceiling rounding in the denominator.
Then in swap_exact_in, the amount of tokens returned for 2.8754 is calculated through
calculate_swap_exact_in, which uses Floor rounding.
output = 1001 Tokens
```

This will then cause a revert on line:

```rust
assert!(
   ctx.accounts.user_token_a_vault.amount <= max_supply_per_wallet,
   "user_token_a_vault.amount must be less than or equal to max_supply_per_wallet"
);
    // 1001 <= 1000 This will revert the transaction
```

Having valid transactions revert is normally a critical issue in an AMM. However, this case can only occur in Heaven when there are no fees at all. Even the slightest fee, `1 bps`, would fully absorb the calculation error.

This can be confirmed by running `test_standard_liquidity_pool_swap_token_b_to_token_a` with `buy_fee_bps` set to `0`.

Recommendations:

Harmonize the rounding used to be consistent Floor or Ceiling.

Customer's response: fixed in 8d07

Fix Review: The mathematical inconsistency persists, but adding a buffer does solve the issue of transactions reverting.

M-04 amount_in calculation does not consider fees, causing sub-optimal trades and making it very difficult to buy up to maximum limit

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: swap.rs#112	Status: fixed in 69ae	

Description:

Within the buy() function, the amount of SOL that is exchanged for token_a, is determined by taking the minimum between max_sol_spend and the output of calculate_swap_exact_out, which is based on the max_purchasable_amount.

This value is then sent to swap_exact_in to be exchanged into token_a tokens.

However, the value is solely calculated on the state of the reserves and does not take into account that fees will reduce the amount before it is exchanged.

As a result, the user will receive less tokens.

Example:

```Rust
max_purchasable_amount = 1000
max_sol_spend = 10 SOL
Amount calculated from calculate_swap_exact_out = 5 SOL
fees = 2000 bps

5 SOL < 10 SOl => swap_exact_in is called with 5 SOl
fees get deducted from the amount: 5 * 0.8 = 4 SOL
The 4 SOL is exchanged for 800 tokens.
```

This has two main impacts:

- Whenever the user indicates a great willingness to buy tokens by setting a high max_sol_spend, the logic will choose the calculated value and the user will paradoxically receive less tokens compared to setting a lower value.
- It is very difficult for any user to buy tokens up to max_supply_per_wallet since the fee delta will make sure there is always a gap remaining between the limit and actual tokens bought. This would require either dozens of transactions to make the fee delta inconsequential or a series of transactions with an extremely low max_sol_spend. Neither are options that should be required under normal circumstances.

NOTE: This would normally also cause a significant issue with the slippage control parameter minimum_out, since the fee delta is sufficiently large to cause reverts. However, per communication by protocol developers, the minimum_out parameter is not a user-provided input but will be calculated by the frontend and will take the fee effect into account. Thereby preventing reverts due to slippage mismatch.

Recommendations:

The amount calculated in calculate_swap_exact_out needs to take the fee deduction into account.

Customer's response: fixed in 69ae

Fix Review: The issue has been resolved.

Low Severity Issues

L-01 Updating supported_pool_type will lead to corrupted pools		
Severity: **Low**	Impact: **High**	Likelihood: **High**
Files: protocol.rs#552	Status: fixed in f73d	

Description:

The `update_protocol_config` function allows changing `supported_pool_type` after initial deployment, which can create a fundamental mismatch between pool types and their fee configurations. Both `Standard` and `Pro` pools inherit their fee structure from the same protocol config, so changing the supported pool type allows creation of pools with incorrect economic parameters.

When a protocol is deployed with `supported_pool_type = Standard`, all fee configurations are set for Standard pools. If an admin later changes this to Pro, newly created Pro pools will inherit the Standard pool fee structure instead of appropriate Pro pool fees, breaking the intended economic model.

- **Economic Model Breakdown**: Pro pools operating with Standard fees (or vice versa) completely undermines the protocol's fee design
- **Operational Inconsistency**: Mixed pool types with mismatched configurations create unpredictable behavior
- **Protocol Integrity Loss**: The fundamental distinction between pool types becomes meaningless when fees don't match the pool type

Recommendations: The supported_pool_type should be immutable after initial protocol creation, so the set_supported_pool_type should be removed from update_protocol_config.

Customer's response: fixed in f73d

Fix Review: The functionality has been removed, which resolves the issue.

L-02 admin_mint_msol does not limit staking up to any % of available liquidity, which can break all pools		
Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: [protocol.rs#326]()	Status: Acknowledged	

Description:

The `admin_mint_msol` function allows the owner to stake available liquidity in Marinade for additional yield.

The amount that can be staked is equal to `100% of ALL` SOL available in heaven since `sol_available_for_stake` == `unstaked_wsol_reserve`.

If a large amount of SOL were to be staked, it is possible that due to market actions, the requirement for SOL would be greater than the remaining SOL present in the protocol.

In such a case, all `Sell` operations would fail due to insufficient liquidity.

Recommendations:

Introduce a protocol config variable `minimal_liquidity` in BPS and use it in the function to ensure that only a certain % of liquidity can be staked.

Customer's response: This is a design choice since the optimal amount of sol to be deposited is something which requires careful consideration, which cannot be done through automated logic. So we acknowledge the possibility, but we believe this is the better approach.

Fix Review: Acknowledged.

L-03 admin_unstake_msol breaks if yield is lower than historical cost

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: protocol.rs#204	Status: Acknowledged	

Description:

The protocol uses the delayed unstake to retrieve `msol` from marinade. During the `admin_unstake_msol` call, a `cost_basis` is calculated using the `total_sol_spent` and `total_msol_received`. This gives an historical average dating from the inception of the protocol.

When the msol is actually claimed, this value is compared against the current market and profit is calculated by subtracting `actual_received` with `cost_basis`.

```rust
state.cost_basis = U128::from(amount)
    .checked_mul(U128::from(config.total_sol_spent))      // Historical average
    .checked_ceil_div(U128::from(config.total_msol_received))
    .0.as_u64();

let actual_received = lending.claim_msol(...);  // Current market rate
let profits = actual_received
    .checked_sub(cost_basis)
    .expect("Overflow occurred while calculating profits"); // PANICS if actual < cost_basis
```

This only works if the current value is always greater than the historical average. There are however black swan events where this will not be the case.

This can be:

- Hack of Marinade
- Reputation loss through scandals
- Upgrade gone wrong, which impacts functionality, etc..

All these can significantly reduce the value of mSOL by a vast amount.

```
Rust
EXAMPLE: 40% mSOL Devaluation Scenario
Setup:
- Heaven historical average: 1.05 SOL per mSOL
- Admin unstakes 1,000 mSOL
- Cost basis = 1,050 SOL

10 Hours Later:
- Black swan: mSOL depegs 40% to 0.63 SOL per mSOL
- Admin claims: actual_received = 630 SOL
- Calculation: 630 - 1,050 = -420 SOL

Result:
- checked_sub() returns None for negative result
- expect() panics: "Overflow occurred while calculating profits"
- 1,000 mSOL ticket permanently locked
- 630 SOL value locked
```

When such events happen, it is very unlikely that the value increases again to the point of profit. The only way to retrieve whatever value is left, is to hotfix the protocol with an additional direct `unstake` function and get the remaining SOL back this way.

Recommendations: Either add an emergency direct admin `unstake` function to retrieve the SOL or allow for the possibility of loss.

```
Rust
if actual_received >= cost_basis {
    let profits = actual_received - cost_basis;
    config.total_realized_profit = config.total_realized_profit.checked_add(profits)?;
} else {
    let loss = cost_basis - actual_received;
    config.total_realized_profit = config.total_realized_profit.checked_sub(loss)?;
}
```

Customer's response: Acknowledged, but this is resolved through a quick protocol upgrade or direct negotiations with Marinade.

Fix Review: Acknowledged.

Informational Issues

I-01. max_creator_trading_fee_bps returns global constraint instead of max actual fee

Description:
In the `assert_valid_fees_config` function, the max actual fee per type is located and added together to ensure that a situation where fees > 100% is not possible.

However, for the `creator_trading_fees`, in contrast to the other fee types, the logic does not compare between `market_cap` and `slot_offset` fees and returns the greatest value.

Instead, it compares the slot_offset trading fee with the global constraint `max_creator_trading_fee` and then always returns the global constraint since that is the theoretical maximum and not the actual fee.

As a result, the `total_bps` will be overinflated and this can cause a valid fee config to be rejected in rare cases.

Recommendation:
Change the max_creator_trading_fee_bps to follow the same logic as the other fee types.

Customer's response: Since this requires an admin setting an extremely unrealistic creator_trading_fee, we acknowledge the theoretical possibility.

Fix Review: Acknowledged

I-02. Changing standard pool fee mode freezes fees to deploy status without update possibility

Description:
When a pool is created, the `FeeConfigurationMode` is hardcoded to `Global`. The creation process does copy the global fee config to local, but there are no functions to update these local values.

If an admin calls `admin_update_standard_liquidity_pool_state` and changes the `FeeConfigurationMode` from `Global` to `Local`, the logic will use the fee structure copied from the global config at deployment.

This can be problematic since it is quite possible the global fee structure has changed since deployment and there is no user nor admin function to update the `Local` fees on a `Standard` pool.

Recommendation:
Remove the FeeConfigurationMode from the admin_update_standard_liquidity_pool_state, there should never be a reason to change the fee mode on a Standard pool.

Customer's response: This requires an admin to make a massive mistake, so very unlikely to ever happen. Acknowledged.

Fix Review: Acknowledged

I-03. assert_input_equal_token_supply is not used

Description:
The pub fn assert_input_equal_token_supply function is declared but never used in the protocol.

Recommendation:
Remove unused functions

Customer's response: fixed.

Fix Review: The function has been removed.

I-04. assert_processed_fee_status named could be clarified

Description:
The naming of the assert_processed_fee_status does not let the reader know what about the status is being asserted and requires reading the actual function logic to understand.

Recommendation:
Change the naming to: assert_fee_status_is_processed

Customer's response: Naming has been changed

Fix Review: fixed

I-05. Allowing supported_pool_type None is redundant

Description:
The create_pool_config function calls set_supported_pool_type to set a pool to either Standard or Pro.

However, the enum also allows a type None. This would break the pool config since many if not all core functions can only work with either a standard or pro pool.

Since emergency pause functionality is already covered through the `allow_create_pool`, there is no reason to have the None option.

Recommendation:
Remove the None from the enum to avoid it being set by accident.

Customer's response: Removed since there are no more Pro pools.

Fix Review: fixed.

I-06. checked_sub has an error message stating Overflow

Description:
Whenever a calculation uses checked_sub, the error message erroneously states that an Overflow occurred. This occurs with every use of checked_sub throughout the protocol.

Example:

```rust
        .checked_sub(amount)
        .expect("Overflow occurred while calculating total_realized_profit");
```

Recommendation:
Change the message to:

```rust
"Underflow ...."
```

Customer's response: Acknowledged

Fix Review: Acknowledged

I-07. credit from debt_and_credit is never used

Description:
The `debt_and_credit` helper function is used by 4 admin functions:

- admin_borrow_sol
- admin_deposit_msol
- admin_repay_sol
- admin_withdraw_sol

In each case, the `debt` and `credit` variable are defined but in every case the `credit` variable is **not** used. Only `debt` is used.

Since `credit` is never used, it is a code inefficiency and computational unit waste to call `kamino.get_credit` when it is unused.

Recommendation:
Refactor `debt_and_credit` to only call and return `debt`.

Customer's response: Acknowledged
Fix Review: Acknowledged

I-08. Unused variable in unstake_msol

Description:

There is a variable in `unstake_msol`, which is defined but never used.

```rust
let msol_balance = self.temp_sol_holder_msol_vault().balance()?;
```

Recommendation:
Unused variables should be removed

Customer's response: Acknowledged
Fix Review: Acknowledged

I-09. Naming ambiguity in fee and staking create functions

Description:
The `create_or_update_protocol_fee_admin` and `create_or_update_protocol_staking_admin` functions set the fee and staking admin for the protocol.

However, in the state field, both are called `current_protocol_admin`, which leads to confusion when called in the code.

The pub fn `assert_protocol_admin` for example, gives no indication to which admin is being checked. This is used in the `update_allow_create_pool` function, where only after digging through the struct does it become clear that it is the Fee admin that can call this function.

Recommendation:
Change the naming to:

- `current_fee_admin`
- `current_staking_admin`

Customer's response: Acknowledged

Fix Review: Acknowledged

I-10. Unused functions in token_util

Description:
There are several functions in token_util.rs which are defined but never used anywhere in the protocol.

- `checked_sufficient_balance`
- `get_transfer_fee`
- `get_transfer_inverse_fee`

Recommendation:
Unused functions should be removed.

Customer's response: Acknowledged

Fix Review: Acknowledged

I-11. Input parameter order in transfer_with_hook can lead to errors

Description:
The `transfer_with_hook` function expects the following input accounts:

- `authority`
- `from`
- `to`
- `mint`

But the standard order for `Token_2022` `transfer_checked` is:

- `token program`
- `from`
- `mint`
- `to`

The `to` and `mint` are inverted. This can lead to input errors which would cause transactions to revert. Since the functionality is not yet active, this is graded as an Informational.

Recommendation:
Switch the `to` and `mint` in the input account list to be consistent with the instruction account order.

Customer's response: Acknowledged

Fix Review: Acknowledged

I-12. Unused circulating_lp_token_supply function

Description:

Since LP functionality has been removed from the protocol, there is no need for the `circulating_lp_token_supply` function.

Recommendation:
Remove the function.

Customer's response: Acknowledged

Fix Review: Acknowledged

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.