OFFSIDE LABS

Omnipair

Smart Contract Security Assessment

January 2026

Prepared for:

Omnipair

Prepared by:

Offside Labs

Gongyu Shi

Sirius Xie

Contents

1 About Offside Labs

Offside Labs is a leading security research team, composed of top talented hackers from both academia and industry.

We possess a wide range of expertise in modern software systems, including, but not limited to, *browsers*, *operating systems*, *IoT devices*, and *hypervisors*. We are also at the forefront of innovative areas like *cryptocurrencies* and *blockchain technologies*. Among our notable accomplishments are remote jailbreaks of devices such as the **iPhone** and **PlayStation 4**, and addressing critical vulnerabilities in the **Tron Network**.

Our team actively engages with and contributes to the security community. Having won and also co-organized *DEFCON CTF*, the most famous CTF competition in the Web2 era, we also triumphed in the **Paradigm CTF 2023** within the Web3 space. In addition, our efforts in responsibly disclosing numerous vulnerabilities to leading tech companies, such as *Apple*, *Google*, and *Microsoft*, have protected digital assets valued at over **$300 million**.

In the transition towards Web3, Offside Labs has achieved remarkable success. We have earned over **$9 million** in bug bounties, and **three** of our innovative techniques were recognized among the **top 10 blockchain hacking techniques of 2022** by the Web3 security community.


https://offside.io/


https://github.com/offsidelabs


https://twitter.com/offside_labs

2 Executive Summary

Introduction

Offside Labs completed a security audit of *Omnipair* smart contracts, starting on November 4th, 2025, and concluding on November 19th, 2025.

Project Overview

Omnipair is a decentralized hyperstructure protocol for spot and margin trading, designed to support permissionless, oracle-less, and isolated-collateral markets. It introduces a novel Generalized Automated Market Maker (GAMM) model, in which liquidity is not only used for swaps but is also natively lent to borrowers, maximizing capital efficiency within each pool.

Audit Scope

The assessment scope contains mainly the smart contracts of the omnipair program for the *Omnipair* project.

The audit is based on the following specific branches and commit hashes of the codebase repositories:

- Omnipair
  - Codebase: `https://github.com/omnipair/omnipair-rs`
  - Commit Hash: 4ddef2a2e5606abfb511d11d8bb017d6448dde8e

We listed the files we have audited below:

- Omnipair
  - programs/omnipair/src/state/rate_model.rs
  - programs/omnipair/src/state/pair.rs
  - programs/omnipair/src/state/user_position.rs
  - programs/omnipair/src/state/futarchy_authority.rs
  - programs/omnipair/src/utils/gamm_math.rs
  - programs/omnipair/src/utils/token.rs
  - programs/omnipair/src/utils/math.rs
  - programs/omnipair/src/utils/account.rs
  - programs/omnipair/src/instructions/emit_value.rs
  - programs/omnipair/src/instructions/futarchy/claim_protocol_fees.rs
  - programs/omnipair/src/instructions/futarchy/distribute_tokens.rs
  - programs/omnipair/src/instructions/futarchy/init_futarchy_authority.rs
  - programs/omnipair/src/instructions/futarchy/update_futarchy_authority.rs
  - programs/omnipair/src/instructions/spot/swap.rs
  - programs/omnipair/src/instructions/lending/remove_collateral.rs
  - programs/omnipair/src/instructions/lending/liquidate.rs
  - programs/omnipair/src/instructions/lending/common.rs
  - programs/omnipair/src/instructions/lending/repay.rs

- programs/omnipair/src/instructions/lending/add_collateral.rs
- programs/omnipair/src/instructions/lending/borrow.rs
- programs/omnipair/src/instructions/lending/flashloan.rs
- programs/omnipair/src/instructions/liquidity/add_liquidity.rs
- programs/omnipair/src/instructions/liquidity/remove_liquidity.rs
- programs/omnipair/src/instructions/liquidity/common.rs
- programs/omnipair/src/instructions/liquidity/initialize.rs
- programs/omnipair/src/lib.rs
- programs/omnipair/src/constants.rs
- programs/omnipair/src/errors.rs
- programs/omnipair/src/events.rs

Findings

The security audit revealed:

- 1 critical issue
- 10 high issues
- 8 medium issues
- 4 low issues
- 17 informational issues

Further details, including the nature of these issues and recommendations for their remediation, are detailed in the subsequent sections of this report.

3 Summary of Findings

ID	Title	Severity	Status
01	Withdrawal Limit Bypass on Maximum Collateral Removal	Critical	Fixed
02	Incorrect Token Transfer Amount When Adding Liquidity	High	Fixed
03	Incorrect Mint Supply Used When Removing Liquidity	High	Fixed
04	Incorrect Accumulated Total Debt Calculation	High	Fixed
05	Missing Token-2022 Transfer Fee Support May Lead to User Loss	High	Fixed
06	Incorrect Collateral Amount Used When Updating applied_min_cf_bps During Liquidation	High	Fixed
07	Missing Update of applied_min_cf_bps After Collateral Adjustments	High	Fixed
08	Collateral May Not Be Reclaimed by Borrowers	High	Fixed
09	Rounding Logic Flaws in Division Operations	High	Fixed
10	Borrow Limit Bypass via Split Positions	High	Fixed
11	Borrow Limit Bypass via Spot Price Manipulation	High	Fixed
12	Swap and Flashloan Fees Not Accumulated	Medium	Fixed
13	Incorrect Exponential Decay Interest Calculation Due to Redundant Scalar	Medium	Fixed
14	Collateral Check Bypass Due to Missing Accounts Reload	Medium	Fixed
15	Precision Loss in User Debt Shares Calculation	Medium	Fixed
16	DoS Risk from Vault Token Accounts Not Enforced as ATAs	Medium	Fixed
17	Inconsistent Token-2022 Support May Lead to DoS	Medium	Fixed
18	Underestimated Liquidation Collateral Factor Causes Premature Liquidations	Medium	Fixed
19	Reserve Reset After Liquidation Can Cause DoS	Medium	Fixed
20	EMA Prices Initialized to Zero Incorrectly	Low	Fixed

ID	Title	Severity	Status
21	Incorrect Maximum Borrow Collateral Factor Used When Calculating Debt Utilization	Low	Fixed
22	Unsafe Casting of Interest Rate and Accumulated Interest to u64	Low	Fixed
23	LTV Buffer Applied Inconsistently Between Borrow Limit and Collateral Factor	Low	Fixed
24	Missing Accounts Validation in View Instructions	Informational	Fixed
25	Insufficient Checks on Futarchy Authority Fee Rates	Informational	Fixed
26	Missing Zero-Amount Check When Minting and Burning Tokens	Informational	Fixed
27	Owner of Output Token Account Not Validated During Swap	Informational	Fixed
28	Swap Failure Due to Insufficient Vault Balance to Cover Futarchy Fee	Informational	Fixed
29	Precision Loss Due to Redundant Division	Informational	Fixed
30	Incorrect Handling of Zero reserve1 When Calculating Utilization Rates	Informational	Fixed
31	Inaccurate Account Data Size Calculation on Initialization	Informational	Fixed
32	Flashloan Edge Case with Unsynced WSOL Allows Bypassing Repayment	Informational	Fixed
33	Unsafe Truncation of Pool Invariant to u64	Informational	Fixed
34	Unnecessary Canonical Bump Calculation	Informational	Fixed
35	Misleading Variable Names and Comments	Informational	Fixed
36	Partial Liquidation May Leave Excess User Debt Shares in Edge Cases	Informational	Fixed
37	Hard-Coded Interest Rate Model Limits Flexibility	Informational	Fixed
38	Rate Settings of Futarchy Authority Cannot Be Updated	Informational	Fixed
39	Improper ErrorCodes	Informational	Fixed
40	Flashloan Borrow Limit Restricted by Reserve	Informational	Acknowledged

4 Key Findings and Recommendations

4.1 Withdrawal Limit Bypass on Maximum Collateral Removal

Severity: Critical	Status: Fixed
Target: Smart Contract	Category: Logic Error

Description

When removing collateral, the `validate_remove` function calculates the maximum withdrawable amount as follows:

```
66 let is_withdraw_all = args.amount == u64::MAX;
67 ...
68 // Calculate maximum withdrawable amount
69 let max_withdrawable = user_collateral
70     .checked_sub(min_collateral as u64)
71     .ok_or(ErrorCode::DebtMathOverflow)?;
72
73 let withdraw_amount = if is_withdraw_all { max_withdrawable } else {
   ↪ *amount };
```

programs/omnipair/src/instructions/lending/remove_collateral.rs#L66-L73

If `args.amount` is `u64::MAX`, the function sets `withdraw_amount` to `max_withdrawable` and performs the validation.

However, in `handle_remove_collateral`, the program sets `withdraw_amount` to the user's entire collateral when the maximum withdrawal is requested, bypassing the calculated limit `max_withdrawable`.

```
120     let is_withdraw_all = args.amount == u64::MAX;
121     let is_token0 = user_token_account.mint == pair.token0;
122     let withdraw_amount = if is_withdraw_all {
123         match is_token0 {
124             true => user_position.collateral0,
125             false => user_position.collateral1,
126         }
127     } else {
128         args.amount
129     };
```

programs/omnipair/src/instructions/lending/remove_collateral.rs#L120-L129

Impact

A malicious user could exploit this by adding collateral, borrowing tokens, and then removing all collateral using `u64::MAX`. The borrowed tokens would remain unpaid, which may result in the pool being drained.

Recommendation

Ensure that the `withdraw_amount` in `handle_remove_collateral` is capped by `max_withdrawable`.

Mitigation Review Log

Fixed in PR-7 and PR-25.

4.2 Incorrect Token Transfer Amount When Adding Liquidity

Severity: High	Status: Fixed
Target: Smart Contract	Category: Logic Error

Description

When adding liquidity to the pool, the program computes the number of LP tokens minted to the user as follows.

```
80  let liquidity: u64 = {
81          let liquidity0 = (args.amount0_in as u128)
82              .checked_mul(total_supply as
    ↪  u128).ok_or(ErrorCode::LiquidityMath
83              Overflow)?
84              .checked_div(pair.reserve0 as
    ↪  u128).ok_or(ErrorCode::LiquidityMath
85              Overflow)?;
86          let liquidity1 = (args.amount1_in as u128)
87              .checked_mul(total_supply as
    ↪  u128).ok_or(ErrorCode::LiquidityMath
88              Overflow)?
89              .checked_div(pair.reserve1 as
    ↪  u128).ok_or(ErrorCode::LiquidityMath
90              Overflow)?;
91          liquidity0.min(liquidity1).try_into().map_err(|_|
    ↪  ErrorCode::Liquidity
92          ConversionOverflow)?
93      };
```

programs/omnipair/src/instructions/liquidity/add_liquidity.rs#L80-L88

However, the program transfers the full `amount0_in` and `amount1_in` from the user to the vault directly. If one side is supplied in excess, the surplus portion should be refunded to the user.

Impact

This issue leads to a direct loss of user funds and inconsistent pool accounting. User funds that exceed the required proportional deposit are retained by the vault. And the reserve of the pool is updated incorrectly.

Proof of Concept

Suppose `reserve0` , `reserve1` and `total_supply` are all 1000 in the pool.

1. A user calls `add_liquidity` with `amount0_in = 1000` and `amount1_in = 2000` .
2. The program transfers all 1000 token0 and 2000 token1 from the user to the pool vault.
3. The calculated `liquidity` is 1000, so 1000 LP tokens are minted to the user.
4. The internal accounting is updated: `reserve0` is 1000 + 1000 = 2000, `reserve1` is 1000 + 2000 = 3000, and `total_supply` is 1000 + 1000 = 2000.

Only 1000 token1 should have been transferred to match the proportional liquidity minting. Instead, all 2000 token1 were used, inflating `reserve1` to 3000.

Recommendation

Calculate the exact amount of tokens to be transferred to vault based on the number of LP tokens to be minted (`liquidity`), the total LP token supply (`total_supply`), and the pool's token reserves (`pair.reserve`).

$$\text{amount}_0 = \left\lceil \frac{\text{liquidity} \cdot \text{reserve}_0}{\text{total\_supply}} \right\rceil$$

Mitigation Review Log

Fixed in PR-13 and PR-25.

4.3 Incorrect Mint Supply Used When Removing Liquidity

Severity: High	Status: Fixed
Target: Smart Contract	Category: Logic Error

Description

When removing liquidity, the program directly fetches the total supply of LP tokens from the mint account:

```
59        let total_supply = lp_mint.supply;
```

programs/omnipair/src/instructions/liquidity/remove_liquidity.rs#L59-L59

This approach underestimates the total supply since the 1000 "dead shares" are never actually minted.

Impact

The dead shares fail to serve their intended purpose, allowing users to receive more tokens than expected. The pool creator could withdraw all of these tokens immediately after pool initialization, while other users could partially extract them by simply adding and then removing liquidity.

Additionally, this discrepancy leads to inconsistent internal accounting between the pool's reserves and the LP token supply.

Recommendation

Use the internal accounting variable `pair.total_supply` instead of directly fetching `lp_mint.supply`.

Mitigation Review Log

Fixed in PR-8.

4.4 Incorrect Accumulated Total Debt Calculation

Severity: High	Status: Fixed
Target: Smart Contract	Category: Logic Error

Description

The total debt in the pool is updated in the `pair.update` method.

```rust
let protocol_share0: u64 = ((total_interest0 as u128 * futarchy_authority.
revenue_share.interest_bps as u128) / BPS_DENOMINATOR as u128) as u64;
let protocol_share1: u64 = ((total_interest1 as u128 * futarchy_authority.
revenue_share.interest_bps as u128) / BPS_DENOMINATOR as u128) as u64;
let lp_share0 = total_interest0 as u64 - protocol_share0;
let lp_share1 = total_interest1 as u64 - protocol_share1;
...
// update total debt
self.total_debt0 += lp_share0;
self.total_debt1 += lp_share1;
```

programs/omnipair/src/state/pair.rs#L283-L290

The interest accrued from borrowers is split between the protocol (`protocol_share`) and the liquidity providers (`lp_share`). However, only the latter is accumulated into the total debt.

In the extreme case where the `interest_bps` equals 100%, the `lp_share` will be zero, and the total debt is never updated at all.

Impact

The `protocol_share` is effectively paid by the liquidity pool rather than by borrowers. As a result, debtors pay less interest, while liquidity providers suffer a direct loss.

Recommendation

Correct the update of total debt using total interest.

Mitigation Review Log

Fixed in PR-9.

4.5 Missing Token-2022 Transfer Fee Support May Lead to User Loss

Severity: High	Status: Fixed
Target: Smart Contract	Category: Token-2022

Description

Certain instructions that transfer tokens to vaults (e.g., add liquidity, swap, add collateral) assume that the vault receives the full transferred amount. However, when using a Token-2022 mint with the transfer-fee extension, the vault actually receives less than expected. The current implementation does not account for this case.

Impact

Internal accounting for pool reserves, LP token supply, and collateral balances may be updated using inflated values. This can result in some users gaining more than intended, while others incur losses.

For instance, users who remove liquidity early may receive more tokens than expected because the reserve used in the calculation is inflated. Conversely, some users may be unable to redeem their tokens due to insufficient balances in the vault, suffering direct losses.

Recommendation

If the protocol intends to support Token-2022 with the transfer-fee extension, logic should be added in the relevant IXs to correctly handle the fees.

Otherwise, it is recommended to remove Token-2022 program account from all IXs and avoid using `InterfaceAccount` accounts to prevent confusion.

Mitigation Review Log

Fixed in PR-22 by removing support for Token-2022.

4.6 Incorrect Collateral Amount Used When Updating applied_min_cf_bps During Liquidation

Severity: High	Status: Fixed
Target: Smart Contract	Category: Logic Error

Description

After performing a liquidation, the `applied_min_cf_bps` of a user position is updated, since a portion of their collateral will be seized. However, the calculation uses the total collateral amount before the liquidation, including the seized portion.

```
188 let collateral_token = pair.get_collateral_token(&debt_token);
189 let collateral_amount = match collateral_token == pair.token0 {
190     true => user_position.collateral0,
191     false => user_position.collateral1,
192 };
193 let applied_min_cf_bps = pair.get_max_debt_and_cf_bps_for_collateral
194 (&pair, &collateral_token, collateral_amount)?.1;
```

programs/omnipair/src/instructions/lending/liquidate.rs#L188-L193

Impact

The `applied_min_cf_bps` is overestimated, making it more difficult to liquidate user positions after a partial liquidation. This increases the risk of bad debts in the protocol.

Recommendation

Update the `applied_min_cf_bps` using the collateral amount after the liquidation.

Mitigation Review Log

Fixed in PR-10.

4.7 Missing Update of applied_min_cf_bps After Collateral Adjustments

Severity: High	Status: Fixed
Target: Smart Contract	Category: Logic Error

Description

After adding or removing collateral, the `applied_min_cf_bps` of the user's position is never updated.

Impact

Because `applied_min_cf_bps` becomes outdated, users may be allowed to borrow more or fewer tokens than intended. This completely undermines the protocol's intended design and risk controls.

Recommendation

Update `applied_min_cf_bps` whenever collateral is added or removed.

Mitigation Review Log

Fixed in PR-11.

4.8 Collateral May Not Be Reclaimed by Borrowers

Severity: High	Status: Fixed
Target: Smart Contract	Category: Protocol Design

Description

When users add collateral, it is transferred directly to the vault without immediately updating the recorded reserve. Before any liquidation occurs, this deposited collateral in the vault may be lent out or swapped.

Impact

This can lead to inconsistent internal accounting, and borrowers may not be able to reclaim their collateral after repaying debts.

Recommendation

Introduce a check whenever tokens are transferred out of the vault to ensure the remaining balance is always greater than the total collateral.

It is recommended to use a dedicated PDA account as a collateral vault. All collateral would first be transferred to this isolated vault, only when a liquidation is performed, the corresponding amount of collateral would be moved to the pool vault. This design provides better separation and can enhance security.

Mitigation Review Log

Fixed in PR-22 and commit 0a415bd726d0c2abd64a19ef16ade8f49293f21a.

4.9 Rounding Logic Flaws in Division Operations

Severity: High	Status: Fixed
Target: Smart Contract	Category: Precision Issue

Description

The program uses floor-rounding division `checked_div` across nearly all arithmetic operations. This behavior can be unsafe in certain scenarios, below are two examples.

```
62      let shares = (amount as u128)
63          .checked_mul(pair.total_debt0_shares as u128)
64          .ok_or(ErrorCode::DebtShareMathOverflow)?
65          .checked_div(pair.total_debt0 as u128)
66          .ok_or(ErrorCode::DebtShareDivisionOverflow)?
67          .try_into()
68          .map_err(|_| ErrorCode::DebtShareDivisionOverflow)?;
69      pair.total_debt0_shares =
        ↪ pair.total_debt0_shares.saturating_add(shares);
70      self.debt0_shares = self.debt0_shares.saturating_add(shares);
```

programs/omnipair/src/state/user_position.rs#L62-L70

During the borrow operation, the user debt shares will be updated. The computed `shares` may be underestimated, and in edge cases (e.g., `amount == 1`), the result will always be zero, causing the borrower to receive the borrowed token without taking any debt shares.

```
161     let fee0 = (amount0 as u128)
162         .checked_mul(FLASHLOAN_FEE_BPS as u128)
163         .unwrap()
164         .checked_div(BPS_DENOMINATOR as u128)
165         .unwrap() as u64;
```

programs/omnipair/src/instructions/lending/flashloan.rs#L161-L165

Flashloan fees are also computed using floor rounding. This causes the protocol to charge less than the intended fee.

Impact

Users may gain unintended benefits, resulting in a loss to the protocol and LPs.

Recommendation

Review all division operations across the codebase and adjust rounding behavior according to economic intent. The basic principle is to round in a way that favors the protocol and is unfavorable to the user.

Mitigation Review Log

Fixed in PR-25.

4.10 Borrow Limit Bypass via Split Positions

Severity: High	Status: Fixed
Target: Smart Contract	Category: Protocol Design

Description

The borrow limit enforcement for each user position relies on `get_max_debt_and_cf_bps_for_collateral`, which uses the AMM reserves (`pair.reserve0` / `pair.reserve1`) as the debt-side liquidity input. However, these reserve values are not updated immediately after a borrow. They are only refreshed when interest accrues or a liquidation is triggered. As a result, the debt-side liquidity used in the maximum borrow calculation becomes stale across multiple borrows.

Impact

Since the borrow limit is enforced on a per-position basis, a borrower can exceed the intended borrowing cap by splitting their collateral across multiple user positions (using different accounts) and borrowing from each position independently. This allows the total borrowed amount to surpass the protocol's intended limit, thereby undermining the expected economic model.

Recommendation

Ensure that the borrow limit calculation incorporates a borrow-sensitive and up-to-date liquidity metric, and cannot be bypassed by splitting collateral across multiple positions.

Mitigation Review Log

Fixed in PR-24 (by applying a constant-product price impact in computing dynamic collateral factor) and PR-35.

4.11 Borrow Limit Bypass via Spot Price Manipulation

Severity: High	Status: Fixed
Target: Smart Contract	Category: Protocol Design

Description

Borrow limits and collateral factors incorporate both EMA price and spot price to compute a pessimistic collateral factor intended to protect against EMA lag. However, because the spot price is sourced from the AMM reserves, a user can temporarily manipulate spot (via swaps) to move spot closer to the lagging EMA.

Impact

This manipulation inflates the computed collateral value, allowing the user to borrow more debt than would be permitted under normal conditions. If the borrowed debt exceeds the value obtainable by selling the tokens at the spot price, the user may directly realize a profit.

Recommendation

Mitigating this issue without relying on an external oracle may be challenging. One possible workaround is to limit the maximum rate of spot price change per slot.

Mitigation Review Log

Fixed in PR-27 by introducing an asymmetric one-way EMA in the dynamic collateral-factor calculation.

4.12 Swap and Flashloan Fees Not Accumulated

Severity: Medium	Status: Fixed
Target: Smart Contract	Category: Logic Error

Description

During swaps, the `total_fee` and the `futarchy_fee` are calculated as follows:

```
162     let total_fee = (amount_in as u128)
163         .checked_mul(pair.swap_fee_bps as u128)
164         .ok_or(ErrorCode::FeeMathOverflow)?
165         .checked_div(BPS_DENOMINATOR as u128)
166         .ok_or(ErrorCode::FeeMathOverflow)? as u64;
167
168     // Calculate futarchy fee portion of the total fee
169     let futarchy_fee = (total_fee as u128)
170         .checked_mul(futarchy_authority.revenue_share.swap_bps as u128)
171         .ok_or(ErrorCode::FeeMathOverflow)?
172         .checked_div(BPS_DENOMINATOR as u128)
173         .ok_or(ErrorCode::FeeMathOverflow)? as u64;
```

programs/omnipair/src/instructions/spot/swap.rs#L162-L174

However, the remaining portion of the fee (`total_fee - futarchy_fee`) is ignored, it is neither transferred out nor recorded in internal accounting.

A similar issue exists in the flashloan logic:

```
161       let fee0 = (amount0 as u128)
162           .checked_mul(FLASHLOAN_FEE_BPS as u128)
163           .unwrap()
164           .checked_div(BPS_DENOMINATOR as u128)
165           .unwrap() as u64;
166
167       let fee1 = (amount1 as u128)
168           .checked_mul(FLASHLOAN_FEE_BPS as u128)
169           .unwrap()
170           .checked_div(BPS_DENOMINATOR as u128)
171           .unwrap() as u64;
```

programs/omnipair/src/instructions/lending/flashloan.rs#L161-L171

These fees are not accumulated into the pool reserves or protocol revenue.

Impact

The protocol and liquidity providers may earn less revenue than expected because a portion of the fees is never credited.

Recommendation

Add the logic to accumulate the fees.

Mitigation Review Log

Fixed in PR-17.

4.13 Incorrect Exponential Decay Interest Calculation Due to Redundant Scalar

Severity: Medium	Status: Fixed
Target: Smart Contract	Category: Math Error

Description

When the fund utilization is low, the interest rate decays exponentially over time. Once the rate reaches the minimum cap (1%), it remains constant. The accumulated interest in this case, therefore, consists of two parts, an exponential decay portion, and a flat portion once the rate reaches the minimum.

The function `time_to_reach_closed_form` computes the time (in seconds) needed for the rate to decay to the minimum cap:

```
103  if r0 <= target { return 0; }
104  let ratio_nad = (r0.saturating_mul(NAD as u128)) / target.max(1);
105  let ln_ratio  = Self::ln_nad(ratio_nad as u64);  // NAD (signed)
106  let t = ((NAD as i128) * ln_ratio) / (exp_rate_nad_per_s as i128);
107  if t <= 0 { 0 } else { t as u128 }
```

programs/omnipair/src/state/rate_model.rs#L103-L107

Note that the returned `t` is scaled by an extra factor of `NAD` (10^9).

Later, when computing the interest integral, `t_to_min` is set to the smaller of `t` and the elapsed time `dt` (which is in seconds and not scaled):

```rust
let t_to_min = Self::time_to_reach_closed_form(last, min_nad, exp_rate,
/*up=*/false)
    .min(dt);

// exp part (to floor): (last - MIN) * NAD / exp_rate
let exp_part  = last.saturating_sub(min_nad).saturating_mul(NAD as u128) /
 ↪
exp_rate;
// flat tail: MIN * (dt - t*)
let flat_part = min_nad.saturating_mul(dt.saturating_sub(t_to_min));
let integral  = (exp_part + flat_part) / (SECONDS_PER_YEAR as u128);
return (min_nad as u64, integral as u64);
```

programs/omnipair/src/state/rate_model.rs#L74-L81

Because `t` is over-scaled, it always exceeds `dt`, then `t_to_min` equals `dt`. Consequently, the flat portion (`flat_part`) is always zero.

Impact

Once the interest rate reaches its minimum cap, no further interest is accumulated. This results in underpayment of debt and unrealized yield for liquidity providers or the protocol.

Recommendation

Either apply the same `NAD` scaling factor to `dt`, or remove the redundant `NAD` multiplier from `time_to_reach_closed_form`.

Mitigation Review Log

Fixed in commit PR-14.

4.14 Collateral Check Bypass Due to Missing Accounts Reload

Severity: Medium	Status: Fixed
Target: Smart Contract	Category: Logic Error

Description

When removing liquidity from the pool, a check is added after transferring tokens from the vault to the user to ensure the vault balance is greater than the corresponding collateral:

```
114         // Check collateral requirements
115         require!(
116             token0_vault.amount >= pair.total_collateral0,
117             ErrorCode::Undercollateralized
118         );
119         require!(
120             token1_vault.amount >= pair.total_collateral1,
121             ErrorCode::Undercollateralized
122         );
```

programs/omnipair/src/instructions/liquidity/remove_liquidity.rs#L114-L122

However, the vault token accounts are not reloaded after the token transfer, so the check does not reflect the updated balances and can be bypassed.

Impact

Liquidity providers may successfully remove liquidity even if the remaining tokens in the vault cannot cover the collateral. Consequently, borrowers might be unable to reclaim their collateral after repaying all debts.

Recommendation

Reload both vault token accounts before performing the check.

Mitigation Review Log

Fixed in PR-15.

4.15 Precision Loss in User Debt Shares Calculation

Severity: Medium	Status: Fixed
Target: Smart Contract	Category: Precision Issue

Description

The protocol uses a share-based system for debt tracking. On the first borrow request, both the initial debt shares (`total_debt0_shares`) and the total debt (`total_debt0`) are set equal to the borrowed amount (`amount`). That is, the total shares are of the same order of magnitude as the total debt.

Impact

After interest accrues, the total debt may exceed the total shares. Since user debt is calculated based on shares, this can lead to insufficient precision, causing new borrowers to

incur unintended additional losses.

Recommendation

A simple fix is to multiply the borrowed `amount` with a scalar factor when initializing `total_debt0_shares`, so that it is significantly larger than `total_debt0`. This ensures sufficient precision when calculating user debts.

Mitigation Review Log

Fixed in PR-34.

4.16 DoS Risk from Vault Token Accounts Not Enforced as ATAs

Severity: Medium	Status: Fixed
Target: Smart Contract	Category: Data Validation

Description

Most instructions validate the passed-in vault token accounts by checking only the mint and/or owner. They do not require the accounts to be ATAs. As a result, users can provide arbitrary token accounts that satisfy the constraints, causing the reserve recorded to be scattered across multiple accounts.

Impact

The balance in the default vault token account (which should be an ATA according to the client SDK) may be smaller than the internal accounting value. This can lead to DoS during operations such as swaps or borrows, negatively affecting the user experience.

Recommendation

Add constraints to ensure that vault token accounts are ATAs.

Mitigation Review Log

Fixed in PR-22, by adding checks to ensure that the vault token accounts are the expected PDA accounts.

4.17 Inconsistent Token-2022 Support May Lead to DoS

Severity: Medium	Status: Fixed
Target: Smart Contract	Category: Token-2022

Description

When initializing the `Pair` account, token mint accounts owned by Token-2022 program are supported.

```
58     pub token0_mint: Box<InterfaceAccount<'info, Mint>>,
59     pub token1_mint: Box<InterfaceAccount<'info, Mint>>,
```

programs/omnipair/src/instructions/liquidity/initialize.rs#L58-L59

However, several IXs (`flashloan` , `swap` , `claim_protocol_fees`) do not support Token-2022 token accounts, as the corresponding token accounts are defined as `Account<'info, TokenAccount>` .

Note that `distribute_tokens` also does not support Token-2022, but this is not the core protocol logic.

Impact

If a pair is initialized with either token mint of Token-2022 program, the affected IXs cannot be invoked successfully, leading to a DoS.

Recommendation

If the protocol intends to support Token-2022 mints, then token accounts can be defined as `InterfaceAccount<'info, TokenAccount>` .

Otherwise, it is recommended to remove Token-2022 program account from all IXs and avoid using `InterfaceAccount` to prevent confusion.

Mitigation Review Log

Fixed in PR-22 by removing support for Token-2022.

4.18 Underestimated Liquidation Collateral Factor Causes Premature Liquidations

Severity: Medium	Status: Fixed
Target: Smart Contract	Category: Logic Error

Description

The function `get_liquidation_cf_bps` is intended to return the liquidation collateral factor (CF) of a user position. Internally, it compares the current calculated liquidation CF `cf_bps` with the position's recorded `applied_min_cf_bps` and returns the larger value:

```
243 pub fn get_liquidation_cf_bps(&self, pair: &Pair, debt_token: &Pubkey) ->
244 Result<u16> {
245     match debt_token == &pair.token1 {
246         true => {
247             let cf_bps = pair.get_max_debt_and_cf_bps_for_collateral
248             (pair, &pair.token0, self.collateral0)?.2;
249             let min_cf_bps = self.collateral0_applied_min_cf_bps;
250             Ok(max(cf_bps, min_cf_bps))
251         },
252         false => {
253             let cf_bps = pair.get_max_debt_and_cf_bps_for_collateral
254             (pair, &pair.token1, self.collateral1)?.2;
255             let min_cf_bps = self.collateral1_applied_min_cf_bps;
256             Ok(max(cf_bps, min_cf_bps))
257         }
258     }
259 }
```

programs/omnipair/src/state/user_position.rs#L243-L256

However, the two values being compared do not represent the same type:

- `cf_bps` is taken from the third returned element of `get_max_debt_and_cf_bps_for_collateral`, representing the liquidation CF.
- `applied_min_cf_bps` is taken from the second returned element of the same function, representing the maximum allowed CF.

```
65 let (borrow_limit, max_allowed_cf_bps, _) =
   ↪ pair.get_max_debt_and_cf_bps_for_
66 collateral(&pair, &collateral_token, collateral_amount)?;
67 ...
68 user_position.set_applied_min_cf_for_debt_token(&vault_token_mint.key(),
69 &pair, max_allowed_cf_bps);
```

programs/omnipair/src/instructions/lending/borrow.rs#L65-L69

The protocol defines `liquidation_cf_bps = max_allowed_cf_bps + LTV_BUFFER_BPS`, therefore the recorded `applied_min_cf_bps` is lower than the liquidation CF by `LTV_BUFFER_BPS`.

Impact

The function `get_liquidation_cf_bps` might return a liquidation CF lower than intended, as the `LTV_BUFFER_BPS` is omitted in `min_cf_bps`. The underestimated liquidation CF will cause premature liquidations, users may suffer unexpected losses.

Recommendation

Add back the `LTV_BUFFER_BPS` to `min_cf_bps` before comparison.

Mitigation Review Log

Initially fixed as suggested above in PR - 20. Finally addressed by recording the `liquidation_cf_bps` directly in the user position, as implemented in PR-24.

4.19 Reserve Reset After Liquidation Can Cause DoS

Severity: Medium	Status: Fixed
Target: Smart Contract	Category: Logic Error

Description

After a liquidation, the reserves of the pool are updated:

```
235 match is_collateral_token0 {
236     true => {
237         user_position.collateral0 = user_position.collateral0.checked_sub
238         (collateral_final).unwrap();
239         pair.total_collateral0 = pair.total_collateral0.checked_sub
240         (collateral_final).unwrap();
241         // Add remaining collateral (after incentive) to reserves
242         pair.reserve0 =
    ↪ pair.reserve0.checked_add(collateral_to_reserves).
243         unwrap();
244         pair.reserve1 = pair.reserve1.saturating_sub(debt_to_repay);
245     }
246     ...
247 }
```

programs/omnipair/src/instructions/lending/liquidate.rs#L235-L242

When writing off the debt, the corresponding amounts are deducted from the reserve. However, `saturating_sub` is used, so if the reserve is smaller than the liquidated debt, it is reset directly to zero.

This scenario can arise due to:

- **Excessive debt:** The current program allows collateral to be lent, so the total debt can exceed the recorded reserve.
- **Insufficient reserve:** Collateral added by users that is not lent increases the vault balance without updating the reserve. Higher balances increase the maximum output of swap operations, which can make the reserve appear smaller than expected.

Impact

Once the reserve has been reset to zero, the only way to increase it is through interest accumulation, as `add_liquidity` , `liquidate` and `borrow` IXs are blocked due to division-by-zero errors. If there is no remaining debt in the pool, this can result in a persistent DoS.

In particular, if `reserve1` becomes zero, even the interest accumulation cannot be performed due to a division-by-zero error when updating the pair. Then almost all core IXs of the program cannot be executed successfully.

```
253  let (util0, util1) = if self.reserve0 > 0 {
254      (
255          ((self.total_debt0 as u128 * NAD as u128) / self.reserve0 as
     ↪  u128)
256          as u64,
257          ((self.total_debt1 as u128 * NAD as u128) / self.reserve1 as
     ↪  u128)
258          as u64
259      )
260  } else {
261      (0, 0)
262  };
```

programs/omnipair/src/state/pair.rs#L253-L260

Recommendation

Disallow borrowing the collateral and add a check whenever a recorded reserve decreases to ensure it remains greater than the total debts.

Mitigation Review Log

Fixed in PR-26 (by adding internal accounting for cash reserves) and PR-33.

4.20 EMA Prices Initialized to Zero Incorrectly

Severity: Low	Status: Fixed
Target: Smart Contract	Category: Logic Error

Description

When initializing the `Pair` account, the recorded EMA prices are set to zero.

```
94              last_price0_ema: 0,
95              last_price1_ema: 0,
```

programs/omnipair/src/state/pair.rs#L94-L95

Impact

This has two main consequences. First, price updates will initially be lower than the actual values until sufficient updates occur. Second, borrow operations cannot be performed immediately after initialization because the collateral value is zero, unless the EMA prices are first updated via another instruction, such as a swap.

Recommendation

Set the initial EMA prices based on the token amounts in the initial liquidity.

Mitigation Review Log

Fixed in PR-16.

4.21 Incorrect Maximum Borrow Collateral Factor Used When Calculating Debt Utilization

Severity: Low	Status: Fixed
Target: Smart Contract	Category: Logic Error

Description

In the function `get_debt_utilization_bps`, the borrow limit of a user position is calculated:

```
222 // NOTE: debt in token0 → collateral is token1
223 let applied_min_cf_bps = self.get_liquidation_cf_bps(pair, debt_token)?;
224 let borrow_limit = self.get_remaining_borrow_limit(pair, debt_token,
    ↪ applied_
225 min_cf_bps)?;
```

programs/omnipair/src/state/user_position.rs#L222-L224

However, the liquidation collateral factor is used, instead of the maximum collateral factor. Note that the liquidation collateral factor is higher than the maximum collateral factor by `LTV_BUFFER_BPS`. As a result, it inflates the calculated borrow limit, even though users are not actually allowed to borrow that much.

Impact

The function `get_debt_utilization_bps` will return an underestimated debt utilization, and this may mislead users when assessing the health of their positions.

Recommendation

Use the maximum allowed collateral factor when computing the borrow limit.

Mitigation Review Log

Fixed in PR-19.

4.22 Unsafe Casting of Interest Rate and Accumulated Interest to u64

Severity: Low	Status: Fixed
Target: Smart Contract	Category: Precision Issue

Description

During the calculation of the current interest rate (`curr`) and accumulated interests (`integral`), both are computed as `u128` . However, the function forcibly casts them to `u64` when returning, without checking for overflow.

```
46 if (last_util as u128) > (self.target_util_end as u128) {
47     // r1 = r0 * e^{+k dt} = r0 * NAD / gd
48     let curr = last
49         .saturating_mul(NAD as u128)
50         / gd.max(1);
51
52     // ∫ r dt = (r1 - r0) / k_real = (r1 - r0) * NAD / exp_rate, then /
       ↪ YEAR
53     let numer    = curr.saturating_sub(last).saturating_mul(NAD as u128);
54     let integral = numer / exp_rate / (SECONDS_PER_YEAR as u128);
55     return (curr as u64, integral as u64);
56 }
```

programs/omnipair/src/state/rate_model.rs#L46-L56

Impact

If `curr` or `integral` exceeds `u64::MAX` , the values will be truncated. This can result in borrowers paying significantly less interest than expected.

Recommendation

Add a check to clamp values to `u64::MAX` to prevent unexpected truncation.

Mitigation Review Log

Fixed in PR-18.

4.23 LTV Buffer Applied Inconsistently Between Borrow Limit and Collateral Factor

Severity: Low	Status: Fixed
Target: Smart Contract	Category: Math Error

Description

The function `pessimistic_max_debt` returns `final_borrow_limit` (the maximum allowed borrow amount) and `max_allowed_cf_bps` (the maximum collateral factor). However, these two values are not internally consistent due to how the LTV buffer is applied.

```
137     // Max allowed CF BPS = pessimistic CF BPS - LTV_BUFFER_BPS
138     let max_allowed_cf_bps =
            ↪ liquidation_cf_bps.saturating_sub(LTV_BUFFER_BPS);
139
140     // Max allowed Y = V * max_allowed_cf_bps / BPS
141     let max_allowed_y: u128 = v
142         .saturating_mul(max_allowed_cf_bps as u128)
143         .checked_div(BPS_DENOMINATOR_U128)
144         .unwrap_or(0);
145
146     // Apply LTV buffer: reduce borrow limit by LTV_BUFFER_BPS to create
            ↪ a
147     buffer before liquidation
148     // borrow_limit = max_allowed_y * (1 - LTV_BUFFER_BPS /
            ↪ BPS_DENOMINATOR)
149     let ltv_buffer_scaled =
            ↪ BPS_DENOMINATOR_U128.saturating_sub(LTV_BUFFER_BPS
150     as u128);
151     let final_borrow_limit = max_allowed_y
152         .saturating_mul(ltv_buffer_scaled)
153         .checked_div(BPS_DENOMINATOR_U128)
154         .unwrap_or(0)
155         .min(u64::MAX as u128) as u64;
```

programs/omnipair/src/utils/gamm_math.rs#L137-L153

Here, `LTV_BUFFER_BPS` is first subtracted from `liquidation_cf_bps` to compute `max_allowed_cf_bps` . Then, when computing `final_borrow_limit` , the result is further scaled by `(BPS_DENOMINATOR_U128 - LTV_BUFFER_BPS) / BPS_DENOMINATOR_U128` .

As a result, the LTV buffer is effectively applied twice (subtraction and multiplication) when calculating `final_borrow_limit` , but only applied once (subtraction) for `max_allowed_cf_bps` .

Impact

This inconsistency may lead to confusion or incorrect assumptions when using these values together.

Specifically:

- If the LTV buffer is intended to be applied only once, then `final_borrow_limit` is consistently underestimated.
- If the LTV buffer is intended to be applied twice, then `max_allowed_cf_bps` is consistently overestimated.

Recommendation

Align the calculations according to the intended LTV buffer design.

Mitigation Review Log

Fixed in PR-24.

4.24 Informational and Undetermined Issues

Missing Accounts Validation in View Instructions

Severity: Informational	Status: Fixed
Target: Smart Contract	Category: Data Validation

In both the `view_pair_data` and `view_user_position_data` IXs, the passed-in `rate_model` account is not validated against the `pair.rate_model` . If a `rate_model` account with different parameter settings is provided, the update logic for the current `pair` may behave unexpectedly.

Additionally, the `pair` account is not checked against `user_position.pair` .

Besides adding the missing checks, it is also suggested to make a copy of the `pair` state and then perform a simulated update, instead of updating the real `pair` account directly. This ensures that the view instructions remain read-only.

Fixed in commit 0e37cb0d4038709bf963e37a56610c8fd7852db4.

Insufficient Checks on Futarchy Authority Fee Rates

Severity: Informational	Status: Fixed
Target: Smart Contract	Category: Data Validation

When initializing the `futarchy_authority` account, no check is performed on `swap_bps` and `interest_bps`. It is suggested to add a bound check.

Fixed in commit bf38eb3c4fad1bcd1afbb3c12d15a072b3f8860f.

Missing Zero-Amount Check When Minting and Burning Tokens

Severity: Informational	Status: Fixed
Target: Smart Contract	Category: Code QA

In the functions `token_mint_to` and `token_burn`, the input amount is not checked. If it is zero, an early return could be used to skip the unnecessary CPI.

Fixed in commit d63cf38425b8a893e36a8964667d74fe112500ef.

Owner of Output Token Account Not Validated During Swap

Severity: Informational	Status: Fixed
Target: Smart Contract	Category: Data Validation

In the `swap` IX, the program does not validate the owner of the `user_token_out_account`. This could lead to loss of funds if the user mistakenly provides an incorrect destination account.

It is suggested to add the owner check.

Fixed in commit 191ed9b8b920a43f3c618a678454916a9ba2eab6.

Swap Failure Due to Insufficient Vault Balance to Cover Futarchy Fee

Severity: Informational	Status: Fixed
Target: Smart Contract	Category: Logic Error

In the `swap` IX, the program calculates the `futarchy_fee` and attempts to transfer this amount of tokens from the vault to the futarchy authority, before transferring the user's input tokens into the vault. If the vault's balance is not sufficient to cover the fee at that moment, the entire swap will fail.

It is recommended to perform the fee transfer after the user's input tokens are transferred into the vault.

Fixed in commit 3f808ab3f38da225f2550ccd884a8af382d115c0.

Precision Loss Due to Redundant Division

Severity: Informational	Status: Fixed
Target: Smart Contract	Category: Precision Issue

In the function `compute_ema`, the `exp_time` and `x` are calculated as follows:

```
10 // Calculate exp_time in NAD scale
11 let exp_time = (half_life as u128 * NAD as u128) / NATURAL_LOG_OF_TWO_NAD
    ↪ as
12 u128;
13 // Calculate x in NAD scale
14 let x = (dt as u128 * NAD as u128) / exp_time;
```

programs/omnipair/src/utils/math.rs#L10-L13

The division when calculating `exp_time` is redundant and may introduce precision loss. It is recommended to calculate `x` directly as

```
let x = (dt as u128 * NATURAL_LOG_OF_TWO_NAD as u128) / half_life as u128;
 ↪
```

Fixed in commit b02132bd437b2e8ed8107aae9edb6cdf8c8e5f25.

Incorrect Handling of Zero reserve1 When Calculating Utilization Rates

Severity: Informational	Status: Fixed
Target: Smart Contract	Category: Logic Error

When calculating utilization rates, the edge case where `self.reserve1` is zero is not handled correctly. Once this occurs, the `update` method of the `Pair` will be blocked, affecting many core IXs.

```
252 // Calculate utilization rates
253 let (util0, util1) = if self.reserve0 > 0 {
254     (
255         ((self.total_debt0 as u128 * NAD as u128) / self.reserve0 as u128)
     ↪
256         as u64,
257         ((self.total_debt1 as u128 * NAD as u128) / self.reserve1 as u128)
     ↪
258         as u64
259     )
260 } else {
261     (0, 0)
262 };
```

programs/omnipair/src/state/pair.rs#L252-L260

It is suggested to refine the logic, set the utilization rate to zero when the corresponding reserve is zero.

Fixed in commit 9ea4764791c5d0a52ece329651a8132ce7b08441.

Inaccurate Account Data Size Calculation on Initialization

Severity: Informational	Status: Fixed
Target: Smart Contract	Category: Code QA

During the initialization of accounts, the implementation utilizes `get_size_with_discriminator` to determine the account data size to be allocated, which is based on the default `std::mem::size_of::<T>()` in Rust.

```
5 pub fn get_size_with_discriminator<T>() -> usize {
6     8 + std::mem::size_of::<T>()
7 }
```

programs/omnipair/src/utils/account.rs#L5-L7

However, this approach does not always yield the correct size after serialization using `BorshSerialize` :

- For `Vec` and `String` , it always return a fixed size `24` , regardless of the expected maximum length.
- For `Option` and `Enum` , more spaces might be allocated due to alignment padding.

It is strongly recommended to use the `InitSpace` macro to calculate account space requirements.

Fixed in commit c99bebb8a5081598b8b2769d0c7f31fa1d9b3b09.

Flashloan Edge Case with Unsynced WSOL Allows Bypassing Repayment

Severity: Informational	Status: Fixed
Target: Smart Contract	Category: Data Validation

The current implementation of the flashloan IX checks the balance of the vault token accounts after repayment. There is an edge case for WSOL: if the token vault holds unsynced WSOL, a user could borrow WSOL and simply invoke `sync_native` instead of repaying the tokens themselves.

It is recommended to invoke the `sync_native` IX and reload the token account before calculating `balance_before` for WSOL.

Fixed in commit 6fdc19fe291abc9281f7e5978c8d1b65dd01a02a.

Unsafe Truncation of Pool Invariant to u64

Severity: Informational	Status: Fixed
Target: Smart Contract	Category: Precision Issue

When viewing the current `k` invariant of the pool, the result is directly cast to `u64` .

```
145 PairViewKind::K => (OptionalUint::from_u64(pair.k() as u64),
146 OptionalUint::OptionalU64(None)),
```

programs/omnipair/src/instructions/emit_value.rs#L145-L145

This is unsafe because the `k` invariant may exceed `u64::MAX` , resulting in a truncated and potentially misleading value being returned.

It is suggested to use `OptionalUint::from_u128(pair.k())` instead.

Fixed in commit 1c03b8a573c54300b5f0ec71c4fc9a72c1cf0711.

Unnecessary Canonical Bump Calculation

Severity: Informational	Status: Fixed
Target: Smart Contract	Category: Code QA

`FutarchyAuthority` , `Pair` and `UserPosition` accounts each store the bump used to derive their PDA. However, in most IXs, instead of using the stored bump, the implementation recalculates the canonical bump every time.

```
26     #[account(
27         mut,
28         seeds = [PAIR_SEED_PREFIX, pair.token0.as_ref(),
    ↪ pair.token1.as_ref(),
29         pair.pair_nonce.as_ref()],
30         bump
31     )]
32     pub pair: Account<'info, Pair>,
```

programs/omnipair/src/instructions/spot/swap.rs#L26-L31

Fixed in commit b05b3a35ae8dce3630efd2260ce63be1bf2525e1.

Misleading Variable Names and Comments

Severity: Informational	Status: Fixed
Target: Smart Contract	Category: Code QA

We identified several variables and parameters whose names or associated comments are misleading:

- The function `get_collateral_token` receives the debt token mint instead of the collateral token mint, the parameter `collateral_token_mint` is misleading.

```rust
pub fn get_collateral_token(&self, collateral_token_mint: &Pubkey) ->
    Pubkey
{
 self.get_token_y(collateral_token_mint)
}
```

programs/omnipair/src/state/pair.rs#L114-L116

- The first parameter of the function `pessimistic_max_debt` is named `collateral_amount_scaled`, one of the returned value is `final_borrow_limit`. Comments suggest that both values are NAD-scaled, but they are not scaled in the current implementation.

```rust
/// Maximum borrowable amount of tokenY using either a fixed CF or an
impact-aware CF
/// derived from constant product AMM pricing mechanics, with pessimistic
spot/ema cap.
///
/// Inputs:
/// - collateral_amount_scaled: X (NAD-scaled)
...
/// Returns:
/// - final_borrow_limit (NAD-scaled Y)
...
pub fn pessimistic_max_debt(
collateral_amount_scaled: u64,
```

programs/omnipair/src/utils/gamm_math.rs#L56-L68

- The comment of function `curve_y_from_v` indicates that both the input `v` and returned result `Y` are NAD-scaled. But they are actually not scaled.

```rust
/// Exact curve solution: given V (NAD-scaled Y) and R1 (raw Y units),
/// solve Y from Y = V * (1 - Y/R1)^2.
/// Let a = V/R1, t = Y/R1. Then
///   t = 2a / (2a + 1 + sqrt(4a + 1))
/// Returns Y as NAD-scaled Y.
#[inline]
fn curve_y_from_v(v: u128, r1: u64) -> u128 {
```

programs/omnipair/src/utils/gamm_math.rs#L10-L16

Fixed in commit d4140aa7dedbbc5a3e0146d7800712cbf0ed6ce3 and 359cccef411e9fbb92ca0d8c723d541c0a0c3c8d.

Partial Liquidation May Leave Excess User Debt Shares in Edge Cases

Severity: Informational	Status: Fixed
Target: Smart Contract	Category: Protocol Design

During liquidation, the debt shares recorded in the user position (`user_position.debt0_shares`) are used to calculate the user's debt (`user_debt`). For partial liquidation, half of the `user_debt` is written off (`debt_to_repay`), and the corresponding shares to deduct from the user position are calculated in the `decrease_debt` method.

However, in some edge cases where the shares are very small, users may experience losses:

- If `user_position.debt0_shares` is 1, it can never be fully liquidated unless the position becomes insolvent, because `decrease_debt` calculates zero shares to deduct. Thus the liquidation can be performed repeatedly, causing `user_position.collateral0` to continuously decrease, while `user_position.debt0_shares` remains unchanged.
- Suppose `user_position.debt0_shares = pair.total_debt0_shares = 3` and `pair.total_debt0 = 6` . In a partial liquidation, `debt_to_repay` will be 3, and the shares calculated for deduction are `3 * 3 / 6 = 1` . As a result, the user still holds 2 debt shares, even though half of their debt has been liquidated.

It is recommended to refine the logic as follows:

1. Calculate the shares to repay first: `shares_to_repay = ceil(0.5 * user_position.debt0_shares)`
2. Calculate the debts to repay as `debt_to_repay = floor(shares_to_repay * pair.total_debt0 / pair.total_debt0_shares)`
3. Deduct `shares_to_repay` from `user_position.debt0_shares` and `pair.total_debt0_shares` , and deduct `debt_to_repay` from `pair.total_debt0` .

Fixed in commit 8f8aaf5b8388eb492a640b76100cce7417cfd5a0 and 5ae9657c505246f23246ed74b1e0cf2fefb164e1.

Hard-Coded Interest Rate Model Limits Flexibility

Severity: Informational	Status: Fixed
Target: Smart Contract	Category: Upgradability

Currently, the parameters in the rate model are hard-coded and always initialized with fixed values:

```rust
    pub fn new() -> Self {
        const SECONDS_PER_HOUR: u64 = 3_600;
        Self {
            // For production you likely want ln(2)/day; for testing we use 1 h
            // exp_rate: NATURAL_LOG_OF_TWO_NAD / SECONDS_PER_DAY,
            exp_rate: NATURAL_LOG_OF_TWO_NAD / SECONDS_PER_HOUR,
            target_util_start: Self::bps_to_nad(TARGET_UTIL_START_BPS),
            target_util_end:   Self::bps_to_nad(TARGET_UTIL_END_BPS),
        }
    }
```

programs/omnipair/src/state/rate_model.rs#L15-L24

It is recommended to allow `exp_rate` to be configurable within a defined range to improve flexibility.

Fixed in commit 66413c5a2ceda8bd840eeb676f8840459a0f712c, by allowing the target utilization to be configurable during initialization.

Rate Settings of Futarchy Authority Cannot Be Updated

Severity: Informational	Status: Fixed
Target: Smart Contract	Category: Upgradability

Several fields in the `FutarchyAuthority` account store fee rates and token distribution rates. Currently, these values cannot be updated once the account is initialized.

```
70          let revenue_share = RevenueShare {
71              swap_bps,
72              interest_bps,
73          };
74
75          let revenue_distribution = RevenueDistribution {
76              futarchy_treasury_bps,
77              buybacks_vault_bps,
78              team_treasury_bps,
79          };
```

programs/omnipair/src/state/futarchy_authority.rs#L70-L79

It is recommended to provide an instruction that allows updating these fields.

Fixed in commit 53fa884d30d3856a4302333670a8f1fb7a5172b7.

Improper ErrorCodes

Severity: Informational	Status: Fixed
Target: Smart Contract	Category: Code QA

The following ErrorCodes are semantically inconsistent with their usage. It's recommended to adjust them to more appropriate codes:

- programs/omnipair/src/instructions/spot/swap.rs#L90-L90
- programs/omnipair/src/instructions/liquidity/remove_liquidity.rs#L76-L83
- programs/omnipair/src/instructions/liquidity/remove_liquidity.rs#L30-L30
- programs/omnipair/src/instructions/futarchy/claim_protocol_fees.rs#L130-L132
- programs/omnipair/src/instructions/futarchy/claim_protocol_fees.rs#L151-L153
- programs/omnipair/src/instructions/futarchy/claim_protocol_fees.rs#L43-L43

Fixed in commit a9f0b1537f1dfa560609999e0b0910bcba40f911.

Flashloan Borrow Limit Restricted by Reserve

Severity: Informational	Status: Acknowledged
Target: Smart Contract	Category: Protocol Design

In the current flashloan implementation, the borrowed amount is restricted to be no greater than the recorded reserve. As a result, collateral deposited by other users might not be used for flashloan borrowing.

```
110         // Ensure loan amounts doesn't exceed available reserves
111         if args.amount0 > 0 {
112             require_gte!(
113                 self.pair.reserve0,
114                 args.amount0,
115                 ErrorCode::BorrowExceedsReserve
116             );
117         }
```

programs/omnipair/src/instructions/lending/flashloan.rs#L110-L117

Since the vault balance is checked after repayment, the collateral remain secure. It is suggested to remove the amount check above to increase the liquidity available for flashloans.

5 Disclaimer

This audit report is provided for informational purposes only and is not intended to be used as investment advice. While we strive to thoroughly review and analyze the smart contracts in question, we must clarify that our services do not encompass an exhaustive security examination. Our audit aims to identify potential security vulnerabilities to the best of our ability, but it does not serve as a guarantee that the smart contracts are completely free from security risks.

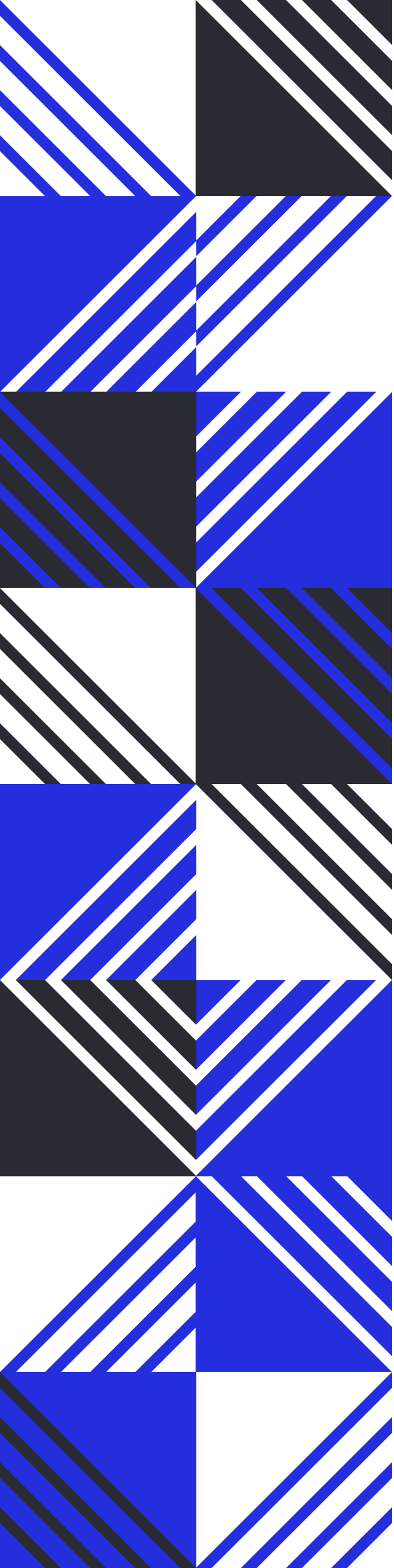
We expressly disclaim any liability for any losses or damages arising from the use of this report or from any security breaches that may occur in the future. We also recommend that our clients engage in multiple independent audits and establish a public bug bounty program as additional measures to bolster the security of their smart contracts.

It is important to note that the scope of our audit is limited to the areas outlined within our engagement and does not include every possible risk or vulnerability. Continuous security practices, including regular audits and monitoring, are essential for maintaining the security of smart contracts over time.

Please note: we are not liable for any security issues stemming from developer errors or misconfigurations at the time of contract deployment; we do not assume responsibility for any centralized governance risks within the project; we are not accountable for any impact on the project's security or availability due to significant damage to the underlying blockchain infrastructure.

By using this report, the client acknowledges the inherent limitations of the audit process and agrees that our firm shall not be held liable for any incidents that may occur subsequent to our engagement.

This report is considered null and void if the report (or any portion thereof) is altered in any manner.



OFFSIDE LABS

https://offside.io/

https://github.com/offsidelabs

https://twitter.com/offside_labs