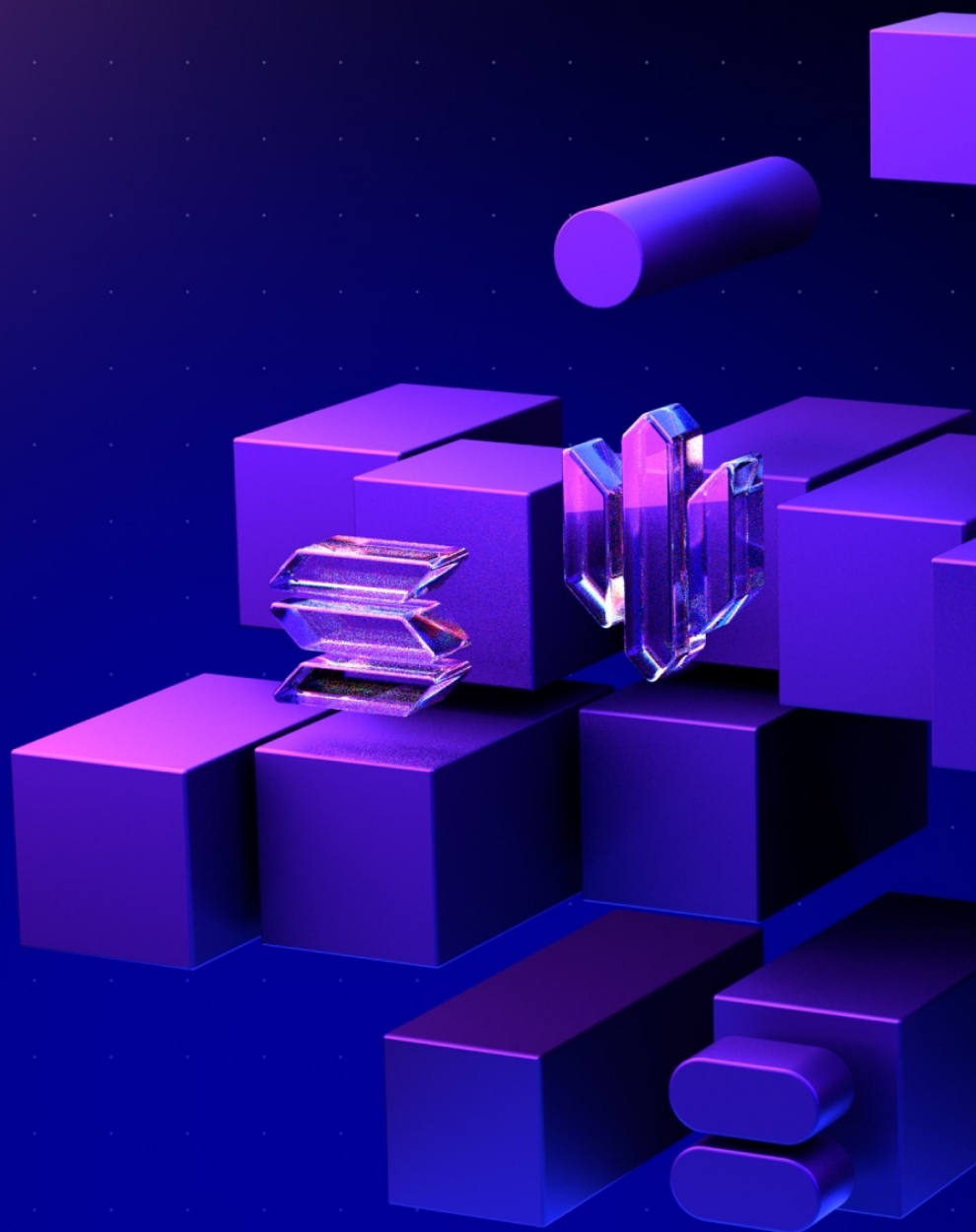
ackee
blockchain security

Omnipair

Oracle-less Lending

27.11.2025



Fuzzing Report

Contents

1. Document Revisions

1.0-draft	Draft Report	21.11.2025
1.0	Final Report	27.11.2025

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain Security

Ackee Blockchain Security is an in-house team of security researchers performing security audits focusing on manual code reviews with extensive fuzz testing for Ethereum and Solana. Ackee is trusted by top-tier organizations in web3, securing protocols including Lido, Safe, and Axelar.

We develop open-source security and developer tooling Wake for Ethereum and Trident for Solana, supported by grants from Coinbase and the Solana Foundation. Wake and Trident help auditors in the manual review process to discover hardly recognizable edge-case vulnerabilities.

Our team teaches about blockchain security at the Czech Technical University in Prague, led by our co-founder and CEO, Josef Gattermayer, Ph.D. As the official educational partners of the Solana Foundation, we run the School of Solana and the Solana Auditors Bootcamp.

Ackee's mission is to build a stronger blockchain community by sharing our knowledge.

Ackee Blockchain a.s.
Rohanske nabrezi 717/4
186 00 Prague, Czech Republic
https://ackee.xyz
hello@ackee.xyz

2.2. Fuzz Testing Methodology

The Ackee Blockchain Security fuzz testing methodology follows a systematic approach:

1. **Code and Architecture Analysis**
    a. High-level review of the Solana program specifications, Rust sources, and instruction handlers to understand the program's size, scope, and functionality.
    b. Analysis of Solana program entry points to identify instruction processors, account validation logic, and critical operations.
    c. Comparison of the Rust implementation and given specifications, ensuring that the program logic correctly implements everything intended.
2. **Fuzz Testing with Trident**
    a. **Interface Analysis**
        - Detailed examination of Solana instruction handlers and their account parameters
        - Identification of Program Derived Addresses (PDAs), account ownership, and cross-program invocation patterns
        - Mapping of account state transitions and Solana runtime data flows
    b. **Initial Behavior Exploration**
        - Writing simple Trident fuzz tests to observe Solana program instruction execution
        - Understanding account validation constraints and Solana runtime limitations
        - Identifying unexpected program behaviors, panics, or edge cases in

instruction processing

c. **Invariant Definition**

- Writing invariants based on expected Solana program properties and account state requirements
- Defining security-critical conditions for account ownership, balance constraints, and authority validation
- Establishing assertions for account state consistency and program-derived address integrity

d. **Complex Stateful Fuzz testing**

- Writing complex Trident fuzz tests that model stateful interactions across multiple Solana instructions
- Testing transaction sequences and their effects on account states and program data
- Exploring interdependencies between instruction handlers and cross-program invocations

e. **Extended Fuzz testing Campaigns**

- Running extended Trident fuzz testing campaigns to explore all edge cases in instruction execution
- Allowing the fuzzer to explore deep account state combinations and program execution paths
- Maximizing Rust code coverage and Solana instruction handler path exploration

f. **Dashboard Analysis**

- Continuous analysis of the Trident fuzz testing dashboard throughout the process

- Monitoring for program panics, instruction failures, and Rust code coverage metrics
- Identifying patterns that indicate potential Solana program vulnerabilities or runtime issues

3. **Vulnerability Assessment**
    a. Classification of discovered Solana program issues by severity and impact on protocol security
    b. Development of proof-of-concept transaction sequences for critical findings
    c. Recommendations for Rust code remediation based on Trident fuzz testing results

2.3. Finding Classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in *configuration* (system settings or parameters, such as deployment scripts, compiler configurations, using multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**N/A**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or *configuration*, but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or *configuration* was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review Team

The following table lists all contributors to this report. For authors of the specific revision, see the “Revision team” section in the respective “Report revision” chapter.

Member’s Name	Position
Matej Hyčko	Lead Auditor
Max Kupchenko	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We’ve put our best effort to find all vulnerabilities in the system, however our findings shouldn’t be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

Omnipair is a decentralized, oracle-less spot and margin trading hyperstructure for permissionless, isolated-collateral markets on Solana. Oracle-less Lending lends pool liquidity to borrowers and enables leveraged trading of long-tail assets without whitelists, external oracles, or centralized risk controls.

Revision 1.0

Omnipair engaged Ackee Blockchain Security to perform fuzz testing of Omnipair Oracle-less Lending with a total time donation of 9 engineering days in a period between November 6 and November 21, 2025, with Matej Hyčko as the lead auditor.

The fuzz testing was performed on the commit `4ddef2a`[1] and the scope was the following:

- Omnipair Oracle-less Lending Protocol, excluding external dependencies.

We began our review by familiarizing ourselves with the protocol's interface and structure. This included understanding the instructions, accounts passed as instruction parameters, and inputs to the instructions.

The next part was dedicated to writing simple fuzz tests to familiarize ourselves with instructions more deeply, to create a simple benchmark for which parts might be more difficult to fuzz, and to understand the whole flow of the scope. This included:

- writing fuzz tests for account initialization instructions;
- writing fuzz tests for operations on initialized accounts;
- writing fuzz tests for protocol state modifications; and

- writing fuzz tests for final execution paths.

After the initial part, we started to implement complex fuzz tests dedicated to the protocol's main logic. This included:

- creating independent fuzz tests for distinct protocol components;
- implementing invariant checks; and
- creating instruction flows to test user workflows.

During fuzz testing, we identified issues primarily caused by missing account validation (see W1, W3) and revealed incompatibilities with Token-2022 mint support (see M1).

Our review resulted in 5 findings, ranging from Warning to High severity. The most severe one H1, which allowed pairs to be initialized with mints that retained mint authorities or disallowed extensions, enabling post-deployment mint inflation or unauthorized token movement that undermined pool accounting.

Ackee Blockchain Security recommends Omnipair:

- investigate the findings and their severity;
- read and review the complete audit report;
- harden account validation and token handling (Token-2022, fees, mint/extension restrictions); and
- address all identified issues.

See Report Revision 1.0 for the system overview.

[1] full commit hash: `4ddef2a2e5606abfb511d11d8bb017d6448dde8e`

4. Findings Summary

The following section summarizes findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- *Description*
- *Exploit scenario* (if severity is low or higher)
- *Recommendation*
- *Fix* (if applicable).

Summary of findings:

Critical	High	Medium	Low	Warning	Info	Total
0	1	1	0	3	0	**5**

Table 2. Findings Count by Severity

Findings in detail:

Finding title	Severity	Reported	Status
H1: Pair Initialization Accepts Unvetted Mints Allowing Malicious Authorities and Extensions	**High**	1.0	Reported
M1: Initialize Does Not Support Token-2022	Medium	1.0	Reported
W1: View Instructions Accept Unbound Accounts for Rate Model and User Position	Warning	1.0	Reported
W2: Initialize Accepts Self-Pair Without Distinct Token Check	Warning	1.0	Reported

Finding title	Severity	Reported	Status
W3: CommonAdjustPosition Context Accepts Non-Canonical Pair-Owned Token Vaults	Warning	1.0	Reported

Table 3. Table of Findings

Report Revision 1.0

Revision Team

Member's Name	Position
Matej Hyčko	Lead Auditor
Max Kupchenko	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

System Overview

Omnipair is a decentralized, oracle-less spot and margin trading hyperstructure for permissionless, isolated-collateral markets on Solana.

Fuzzing

Fuzz testing is an automated testing technique that discovers bugs by providing invalid, unexpected, or random data as inputs to a Solana program. Fuzz testing helps identify edge cases and vulnerabilities that might be missed during manual review.

Manually guided fuzzing was used throughout the review. We setup accounts and instruction parameter ranges, that were then used to create various instruction sequences to verify real workflows.

Property-based fuzzing verified instruction correctness and expected behavior. We specified invariants—properties that must always hold—which were automatically checked after each successful execution. These checks surfaced subtle logic errors and account-state inconsistencies, and the campaign exercised a broad range of Solana system instructions under strong assertions.

The complete list of implemented execution flows and invariants is available

in Appendix B.

Findings

The following section presents the list of findings discovered in this revision. For the complete list of all findings, Go back to Findings Summary

H1: Pair Initialization Accepts Unvetted Mints Allowing Malicious Authorities and Extensions

High severity issue

Impact:	High	Likelihood:	Medium
Target:	programs/omnipair/src/instructions/liquidity/initialize.rs	Type:	Data validation

Description

The pair initialization accepts arbitrary `token0_mint` and `token1_mint` without vetting mint authorities or token extensions. Mints with active `mint_authority` or extensions such as a permanent delegate can undermine accounting and enable mint inflation or unauthorized transfers after the pool is live. The current validation does not restrict mint authorities or detect disallowed extensions.

Listing 1. Excerpt from initialize

```
155 pub fn validate(&self, args: &InitializeAndBootstrapArgs) -> Result<()> {
156     let InitializeAndBootstrapArgs {
157         swap_fee_bps,
158         half_life,
159         fixed_cf_bps,
160         amount0_in,
161         amount1_in,
162         lp_name,
163         lp_symbol,
164         lp_uri,
165         ..
166     } = args;
167
168     // validate pool parameters
169     require_gte!(BPS_DENOMINATOR, *swap_fee_bps,
    ErrorCode::InvalidSwapFeeBps); // 0 <= swap_fee_bps <= 100%
170     require_gte!(*half_life, MIN_HALF_LIFE, ErrorCode::InvalidHalfLife); //
    half_life >= 1 minute
171     require_gte!(MAX_HALF_LIFE, *half_life, ErrorCode::InvalidHalfLife); //
```

```
    half_life <= 12 hours
172
173     // validate fixed_cf_bps if provided
174     if let Some(cf_bps) = fixed_cf_bps {
175         require_gte!(BPS_DENOMINATOR, *cf_bps, ErrorCode::InvalidArgument);
    // 0 <= fixed_cf_bps <= 100%
176         require_gte!(*cf_bps, 100, ErrorCode::InvalidArgument); //
    fixed_cf_bps >= 1% (100 bps) minimum
177     }
178
179     // validate bootstrap parameters
180     require!(*amount0_in > 0 && *amount1_in > 0, ErrorCode::AmountZero);
181     require_gte!(self.deployer_token0_account.amount, *amount0_in,
    ErrorCode::InsufficientAmount0In);
182     require_gte!(self.deployer_token1_account.amount, *amount1_in,
    ErrorCode::InsufficientAmount1In);
183
184     #[cfg(feature = "production")]
185     {
186         let lp_mint_key: String = self.lp_mint.key().to_string();
187         let last_4_chars = &lp_mint_key[lp_mint_key.len() - 4..];
188         require_eq!("omLP", last_4_chars, ErrorCode::InvalidLpMintKey);
189     }
190
191     require!(lp_name.len() <= 32, ErrorCode::InvalidLpName);
192     require!(lp_name.is_ascii(), ErrorCode::InvalidLpName);
193     require!(lp_symbol.len() <= 10, ErrorCode::InvalidLpSymbol);
194     require!(lp_symbol.is_ascii(), ErrorCode::InvalidLpSymbol);
195     require!(lp_uri.len() <= 200, ErrorCode::InvalidLpUri);
196     require!(lp_uri.starts_with("http"), ErrorCode::InvalidLpUri);
197
198     Ok(())
199 }
```

Exploit scenario

Alice a deployer who wants to create a pair.

1. Alice chooses a custom mint `M` as `token0_mint` during the `Initialize` instruction while retaining the `mint_authority` for `M`;
2. Alice deposits an initial amount and the pool initializes successfully;

3. After initialization, Alice mints additional `M` to her account using the retained `mint_authority`; and
4. Alice uses the newly minted tokens to extract value from the pool, breaking assumptions about fixed supply.

Recommendation

Enforce strict mint vetting during pair initialization by requiring that `mint_authority` is disabled (or set to an allowlisted authority) for both `token0_mint` and `token1_mint` and rejecting mints that expose disallowed token extensions (for example, permanent delegate).

Go back to Findings Summary

M1: Initialize Does Not Support Token-2022

Medium severity issue

Impact:	Low	Likelihood:	High
Target:	instructions/liquidity/initialize.rs	Type:	Logic error

Description

The `InitializeAndBootstrap` instruction claims to support both standard SPL Token and Token-2022 mints by including both `token_program` and `token_2022_program` in the account structure. However, the vault initialization constraints do not specify which token program to use when creating the associated token accounts.

Listing 2. Excerpt from initialize

```
108 #[account(
109     init_if_needed,
110     payer = deployer,
111     associated_token::mint = token0_mint,
112     associated_token::authority = pair,
113 )]
114 pub token0_vault: Box<InterfaceAccount<'info, TokenAccount>>,
115
116 #[account(
117     init_if_needed,
118     payer = deployer,
119     associated_token::mint = token1_mint,
120     associated_token::authority = pair,
121 )]
122 pub token1_vault: Box<InterfaceAccount<'info, TokenAccount>>,
```

When the `init_if_needed` constraint is used without specifying `associated_token::token_program`, Anchor defaults to using the standard Token program. This causes the instruction to fail when attempting to

initialize a pair with Token-2022 mints, as the associated token account creation will use the wrong program.

For comparison, the token transfer logic correctly determines which program to use based on mint ownership:

Listing 3. Excerpt from initialize

```
292 match ctx.accounts.token0_mint.to_account_info().owner ==
    ctx.accounts.token_program.key {
293     true => ctx.accounts.token_program.to_account_info(),
294     false => ctx.accounts.token_2022_program.to_account_info(),
295 },
296 amount0_in,
297 ctx.accounts.token0_mint.decimals,
```

However, this runtime check cannot be applied to the `init_if_needed` constraint, which executes before the instruction handler logic.

Exploit scenario

1. Alice attempts to initialize a new pair using two Token-2022 mints.
2. The instruction fails during vault initialization because Anchor tries to create the associated token accounts using the standard Token program instead of Token-2022.
3. Alice is unable to create pairs with Token-2022 mints, despite the protocol claiming to support them.

Recommendation

Specify the token program for each vault initialization.

Go back to Findings Summary

W1: View Instructions Accept Unbound Accounts for Rate Model and User Position

Impact:	Warning	Likelihood:	N/A
Target:	programs/omnipair/src/instructions/emit_value.rs	Type:	Data validation

Description

The view instructions in `programs/omnipair/src/instructions/emit_value.rs` allow mismatched accounts:

1. `view_pair_data` accepts any `RateModel` account instead of enforcing `address = pair.rate_model`, then calls `pair.update(&ctx.accounts.rate_model, …)`, which mutates `Pair` state using a potentially foreign model.
2. `view_user_position_data` does not bind `user_position` to the provided `pair`, allowing reads that mix a `user_position` from Pair A with `pair` data from Pair B, emitting nonsensical metrics.

Listing 4. Excerpt from emit_value

```
107 #[derive(Accounts)]
108 pub struct ViewPairData<'info> {
109     #[account(mut)]
110     pub pair: Account<'info, Pair>,
111     pub rate_model: Account<'info, RateModel>,
112     #[account(
113         seeds = [FUTARCHY_AUTHORITY_SEED_PREFIX],
114         bump
115     )]
116     pub futarchy_authority: Account<'info, FutarchyAuthority>,
```

Listing 5. Excerpt from emit_value

```
120 pub struct ViewUserPositionData<'info> {
```

```
121     #[account(mut)]
122     pub pair: Account<'info, Pair>,
123     #[account(mut)]
124     pub user_position: Account<'info, UserPosition>,
125     pub rate_model: Account<'info, RateModel>,
126     #[account(
127         seeds = [FUTARCHY_AUTHORITY_SEED_PREFIX],
128         bump
129     )]
130     pub futarchy_authority: Account<'info, FutarchyAuthority>,
131 }
```

Exploit scenario

Alice is a user who wants to view the data of a pair.

1. Alice supplies a `RateModel` account different from `pair.rate_model` when calling `view_pair_data`;
2. Alice invokes the instruction on-chain, which calls `pair.update` with Alice's `RateModel`; and
3. The program updates rates using the foreign model parameters.

Recommendation

Enforce account binding to prevent mismatched reads and unintended updates for both `view_pair_data` and `view_user_position_data` instructions.

Go back to Findings Summary

W2: Initialize Accepts Self-Pair Without Distinct Token Check

Impact:	Warning	Likelihood:	N/A
Target:	programs/omnipair/src/instructions/liquidity/initialize.rs	Type:	Data validation

Description

The `initialize` instruction's validation omits a distinctness check between `token0_mint` and `token1_mint`. If both mints are equal, both vaults are derived as the same Associated Token Account (ATA) for the pair Program Derived Address (PDA) and that mint, because ATAs are unique per `(owner, mint)`.

Since vault accounts use `init_if_needed`, this configuration is accepted during initialization, producing a self-pair where both sides of the pool share one token account. Core flows (for example, swaps) expect distinct vaults and fail validation, effectively bricking the pool.

Listing 6. Excerpt from initialize

```
108 #[account(
109     init_if_needed,
110     payer = deployer,
111     associated_token::mint = token0_mint,
112     associated_token::authority = pair,
113 )]
114 pub token0_vault: Box<InterfaceAccount<'info, TokenAccount>>,
115
116 #[account(
117     init_if_needed,
118     payer = deployer,
119     associated_token::mint = token1_mint,
120     associated_token::authority = pair,
121 )]
122 pub token1_vault: Box<InterfaceAccount<'info, TokenAccount>>,
```

Exploit scenario

Alice a deployer who wants to create a self-pair.

1. Alice sets `token0_mint` and `token1_mint` to the same mint `X` and calls the `initialize` instruction;
2. The program derives both `token0_vault` and `token1_vault` to the same ATA for `(pair_pda, X)` and accepts them due to `init_if_needed`;
3. Alice attempts a swap operation, which expects distinct input and output vault accounts; and
4. The swap validation fails because both roles map to the same account.

Recommendation

Add strict distinctness in initialization by checking that `token0_mint` and `token1_mint` are different.

Go back to Findings Summary

W3: CommonAdjustPosition Context Accepts Non-Canonical Pair-Owned Token Vaults

Impact:	Warning	Likelihood:	N/A
Target:	programs/omnipair/src/instructions/lending/common.rs	Type:	Data validation

Description

Instructions that use the `CommonAdjustPosition` accounts context accept any pair-owned SPL `TokenAccount` as the vault for either token mint. The constraint only checks that `token_vault.owner == pair.key()` and that the mint matches one of the pair tokens, without enforcing that the vault is the canonical pool vault.

As a result, users can introduce additional, non-canonical pair-owned token accounts and pass them as `token_vault`. The program then authorizes transfers from these accounts using the pair seeds. This fragments liquidity across multiple vaults, complicating accounting.

Listing 7. Excerpt from common

```
63 #[account(
64     mut,
65     constraint = token_vault.mint == pair.token0 || token_vault.mint ==
   pair.token1,
66     constraint = token_vault.owner == pair.key() @ ErrorCode::InvalidVaultIn,
67 )]
68 pub token_vault: Box<InterfaceAccount<'info, TokenAccount>>,
69
70 #[account(
71     mut,
72     constraint = user_token_account.mint == pair.token0 ||
   user_token_account.mint == pair.token1,
73     token::authority = user,
74 )]
75 pub user_token_account: Box<InterfaceAccount<'info, TokenAccount>>,
76
```

```
77 #[account(address = token_vault.mint)]
78 pub vault_token_mint: Box<InterfaceAccount<'info, Mint>>,
```

Recommendation

Bind `token_vault` to the canonical pool vault and ensure consistent usage across instructions.

Go back to Findings Summary

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain Security, Audit Report | Omnipair: Oracle-less Lending, 27.11.2025.

Appendix B: Trident Findings

This section details the results and methodology of fuzz testing performed using the Trident framework during the audit.

B.1. Implementation Details

Fuzz testing with Trident executes the actual Solana program logic without approximations or simplifications. The programs are compiled in their production form and deployed to TridentSVM, ensuring that test results accurately reflect real-world behavior.

Execution Flows represent the core testing strategy:

- **Revision 1.0**: Used predefined sequences of Solana instructions executed during the fuzz testing process, following specific user scenarios and administrative workflows

B.2. Fuzzing

The following table lists all implemented execution flows in the Trident fuzz testing framework.

ID	Flow	Added
F1	Initialize futarchy authority with revenue distribution configuration	1.0
F2	Distribute tokens to treasury recipients according to BPS allocation	1.0
F3	Claim protocol fees from pair to futarchy authority	1.0
F4	Initialize new trading pair with initial liquidity bootstrap	1.0
F5	Add liquidity to existing pair with proportional or imbalanced amounts	1.0

ID	Flow	Added
F6	Remove liquidity from pair with slippage protection	1.0
F7	Swap tokens with slippage protection and fee collection	1.0
F8	Add collateral to user position	1.0
F9	Borrow tokens against collateral	1.0
F10	Repay borrowed tokens	1.0
F11	Remove collateral from position	1.0
F12	Liquidate undercollateralized positions	1.0
F13	Execute flashloan with fee repayment	1.0

Table 4. Trident fuzzing flows

The following table lists all implemented invariant checks in the Trident fuzz testing framework.

ID	Invariant	Added	Status
IV1	Recipients are set correctly	1.0	Success
IV2	Revenue distribution BPS sum to maximum	1.0	Success
IV3	Revenue share BPS are within valid range	1.0	Success
IV4	Authority receives exactly the claimed amounts	1.0	Success
IV5	Protocol revenue reserves decrease by exactly the claimed amounts	1.0	Success
IV6	Vaults hold at least reserves plus collateral minus debt with tolerance for rounding	1.0	Success
IV7	Pair core state does not change when claiming protocol fees	1.0	Success
IV8	Pair identity does not change	1.0	Success

ID	Invariant	Added	Status
IV9	Distribution leaves no dust in source token account when pre-amount > 0	1.0	Success
IV10	Pair reserves match amounts deposited	1.0	Success
IV11	Pair total supply includes minimum liquidity that is locked permanently	1.0	Success
IV12	Pair parameters match initialization arguments	1.0	Success
IV13	Vault balances are at least pair reserves	1.0	Success
IV14	Authority WSOL balance increases by at least pair creation fee	1.0	Success
IV15	Initialization fails when token0_mint equals token1_mint	1.0	**Fail** (W2)
IV16	Pair reserves increase by at least the deposited amounts	1.0	Success
IV17	Pair total_supply increases by liquidity minted	1.0	Success
IV18	User receives the expected liquidity tokens	1.0	Success
IV19	User receives at least min_liquidity_out	1.0	Success
IV20	Vault balances are at least pair reserves	1.0	Success
IV21	User receives at least minimum amount out for both tokens	1.0	Success
IV22	Pair total supply decreases by liquidity burned	1.0	Success
IV23	User LP balance decreases by liquidity amount in	1.0	Success
IV24	Vault balances are at least reserves plus collateral	1.0	Success

ID	Invariant	Added	Status
IV25	User balance decreases by exactly the collateral amount	1.0	Success
IV26	User position collateral increases by exactly the amount added	1.0	Success
IV27	Pair total collateral increases by exactly the amount added	1.0	Success
IV28	Vault balance is at least reserves plus collateral minus debt	1.0	Success
IV29	User position is properly initialized with correct owner and pair	1.0	Success
IV30	User receives borrowed tokens	1.0	Success
IV31	Pair total debt increases by at least the borrow amount	1.0	Success
IV32	User position debt shares increase correctly	1.0	Success
IV33	Vault balance is at least reserves plus collateral minus debt	1.0	Success
IV34	User position ownership does not change	1.0	Success
IV35	Collateral amounts do not change during borrow	1.0	Success
IV36	token_vault is the canonical ATA for (pair, vault_token_mint)	1.0	**Fail** (W3)
IV37	User pays tokens for repayment	1.0	Success
IV38	When repaying all debt, repay amount equals initial debt	1.0	Success
IV39	User position debt shares decrease correctly	1.0	Success
IV40	When repaying all debt, final debt shares are zero	1.0	Success

ID	Invariant	Added	Status
IV41	Vault balance is at least reserves plus collateral minus debt	1.0	Success
IV42	User position ownership does not change	1.0	Success
IV43	Collateral amounts do not change during repay	1.0	Success
IV44	User token balance increases by exactly the withdraw amount	1.0	Success
IV45	User position collateral decreases by exactly the withdraw amount	1.0	Success
IV46	Pair total collateral decreases by exactly the withdraw amount	1.0	Success
IV47	When withdrawing all available collateral, final collateral is zero	1.0	Success
IV48	Vaults hold at least reserves plus total collateral minus debt	1.0	Success
IV49	User position ownership does not change	1.0	Success
IV50	Debt amounts do not change during collateral removal	1.0	Success
IV51	Non-withdrawn collateral remains unchanged	1.0	Success
IV52	Liquidator receives liquidation incentive	1.0	Success
IV53	User position collateral decreases	1.0	Success
IV54	Pair total collateral decreases by the seized amount	1.0	Success
IV55	Collateral seized is at least the liquidation incentive	1.0	Success
IV56	Debt shares decrease during liquidation	1.0	Success

ID	Invariant	Added	Status
IV57	Collateral reserve increases by seized collateral minus incentive	1.0	Success
IV58	Vault balances are at least reserves plus collateral minus debt	1.0	Success
IV59	Position ownership remains unchanged	1.0	Success
IV60	Non-seized collateral remains unchanged	1.0	Success
IV61	Vault balances increase by at least the fee	1.0	Success
IV62	Vault balances increase when fees are collected	1.0	Success
IV63	Vaults hold at least reserves plus collateral minus debt	1.0	Success
IV64	Pair identity does not change	1.0	Success
IV65	Collateral and debt shares do not change during flashloan	1.0	Success
IV66	Total supply does not change during flashloan	1.0	Success
IV67	User pays exactly amount in	1.0	Success
IV68	User receives at least minimum amount out	1.0	Success
IV69	Constant product remains positive	1.0	Success
IV70	Reserves remain positive after swap	1.0	Success
IV71	Authority receives futarchy fee	1.0	Success
IV72	Vault balances are at least pair reserves	1.0	Success
IV73	user_position belongs to the given pair	1.0	**Fail** (W1)
IV74	rate_model equals the pair's configured rate_model	1.0	**Fail** (W1)

Table 5. Trident fuzzing invariants

ackee
blockchain security

Thank You

Ackee Blockchain a.s.

Rohanske nabrezi 717/4
186 00 Prague
Czech Republic

hello@ackee.xyz