CANTINA

Coinbase stablecoin liquidity

Security Review

Cantina Managed review by:

M4rio.eth, Lead Security Researcher

Jay, Security Researcher

November 30, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Base is a secure, low-cost, builder-friendly Ethereum L2 built to bring the next billion users onchain.

From Nov 12th to Nov 14th the Cantina team conducted a review of Coinbase - Stablecoin Liquidity on commit hash b9b70c7f. The team identified a total of **11** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	2	2	0
Low Risk	0	0	0
Gas Optimizations	0	0	0
Informational	9	8	1
Total	**11**	**10**	**1**

2.1 Scope

The security review had the following components in scope for Coinbase - Stablecoin Liquidity on commit hash b9b70c7f:

```
programs/scaas-liquidity
├── Cargo.toml
├── src
│   ├── constants.rs
│   ├── errors.rs
│   ├── lib.rs
│   └── state.rs
└── Xargo.toml
```

3 Findings

3.1 Medium Risk

3.1.1 Incorrect 1:1 Transfers Without Decimal Normalization Between Tokens

Severity: Medium Risk

Context: lib.rs#L167-L177

Description: The protocol currently assumes that all supported tokens share the same decimal configuration and transfers `amount_after_fee` 1:1 between `from_mint` and `to_mint` without normalizing for their respective decimals. If a token is onboarded with a different decimals value than others, the protocol's accounting becomes inconsistent: the same integer amount may represent different real-world quantities of value. This can result in users receiving too much or too little when swapping or redeeming, causing potential loss of user funds or value extraction in misconfigured pools.

Recommendation:

- Option A (preferred for flexibility): Always read decimals from both mints and normalize amounts before slippage checks, liquidity checks, and transfers (e.g., adjust by $10^{\text{to\_decimals} - \text{from\_decimals}}$ with safe math and clear rounding rules).

- Option B (simpler operationally): On token onboarding, enforce that all supported mints have identical decimals, rejecting any that differ, and document this invariant clearly in code and docs.

3.1.2 Deppeged tokens can be used to drain the other tokens

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: Swaps in the liquidity pool are implemented at a hard-coded 1:1 exchange rate between any two supported tokens, with only a fee applied on the input token:

- User provides `amount_in` of `from_mint`.
- The program computes a fee:
  - `fee_amount = amount_in * fee_rate / 10_000`.
- The net amount is:
  - `amount_after_fee = amount_in - fee_amount`.
- `amount_after_fee` of `from_mint` is transferred from the user into the `from_vault`.
- The same `amount_after_fee` of `to_mint` is transferred from the `to_vault` to the user.
- There is no pricing curve or oracle; the swap is always 1:1 on the net amount, regardless of external market prices.

At the same time, there is no mechanism to remove or disable a token once it has been added to `supported_tokens`. As long as a mint remains in the supported set, it can be used in swaps indefinitely.

This combination creates an economical risk: if one supported token depegs (e.g., a token trades under $1-f$, where `f` is the fee, on external markets), an attacker can:

- Buy the depegged token cheaply on other markets.
- Swap it in this pool at the fixed 1:1 rate for healthy tokens.
- Systematically drain the pool's good assets by repeating this trade.

Because there is no way to delist or isolate the compromised mint, the pool remains fully exposed to this drain scenario until the authority pauses the swaps and program is upgraded to introduce a way to remove that token.

Recommendation: Add a mechanism for the authority to remove or disable a token from the pool (e.g., remove it from `supported_tokens` or mark it as inactive). Because the program is upgradable this issue was marked as Medium.

3.2 Informational

3.2.1 Minor issues and typos

Severity: Informational

Context: errors.rs#L23-L24, errors.rs#L25-L27

Description:

- ☐ We use `MAX_FEE_RATE` to represent the 10% max fee, but for 100% we use 10000, consider using a constant as well `FEE_DENOMINATOR`.
- ☐ The following errors `InvalidFeeRecipient ("Invalid fee recipient")`, `InvalidVaultAccount ("Invalid vault account")`, and `ReservationOverflow ("Reservation over` are not used.
- ☐ The swap vaults accounts naming `from_vault`, `to_vault` are a bit misleading, especially in the CPI calls `from: ctx.accounts.to_vault_token_account.to_account_info(),`. Consider renaming them to `in_vault`/`out_vault`.
- ☐ The `pool` cached in the vault account is unnecessary, the pool is already in the vault's seed.

Recommendation: Consider fixing the issues mentioned above.

3.2.2 Missing Usage and Validation for Persisted Vault Account

Severity: Informational

Context: state.rs#L24

Description: The program persists `TokenVault.vault_account` as a Pubkey, but this field is never read or validated against the PDA SPL token account that the program actually uses at runtime. All instructions that operate on the vault token account instead derive and constrain the PDA directly, so changing vault_account does not currently affect behavior or funds; it is effectively unused metadata. However, this mismatch between stored state and actual behavior can mislead off-chain indexers, audits, or future code that assumes `vault_account` reflects the true vault token account.

Recommendation: To remediate this, the program should either enforce consistency or remove the field. One option is to keep `vault_account` and add explicit checks in instructions that derive the PDA vault, asserting it equals `TokenVault.vault_account`, and initialize it to the derived PDA when the vault is created. The other option is to remove vault_account entirely if it is not needed, simplifying the account state and avoiding confusion, and to add tests that confirm vault behavior relies solely on the PDA constraints rather than the persisted field.

3.2.3 Unnecessary Mutable Account in swap Context

Severity: Informational

Context: lib.rs#L425-L430

Description: The Swap instruction marks `to_vault: Account<'info, TokenVault>` as mut, but the handler only reads from this account (to check reserved_amount) and never modifies it. This causes the runtime to take a write lock on `to_vault` unnecessarily, slightly increasing compute and contention without any corresponding safety benefit. It's not a direct security risk, but it is a correctness/efficiency smell and can marginally increase the program's surface for runtime failures under heavy load.

Recommendation: In the Swap accounts struct, remove `mut` from the `to_vault` account declaration so it's read☐only instead of mutable.

3.2.4 Unrestricted initialize Allows First Caller to Seize Administrative Roles

Severity: Informational

Context: lib.rs#L26-L28

Description: The initialize instruction is currently permissionless and allows any caller to set critical administrative authorities for the pool. Because there is no restriction that initialize must be invoked by

a known, trusted signer (e.g., deployer or upgrade authority), the first caller can front run the intended deployer and set themselves as `operations_authority`, `pause_authority,` and `fee_recipient`. This effectively grants them control over pool operations (including pausing) and fee collection. While the impact may be mitigated in this deployment by the ability to redeploy a single pool, the pattern introduces unnecessary first caller risk and weakens the trust model around administrative control.

Recommendation: Restrict the initialize instruction so it can only be called by a trusted, predefined authority. Possible fixes:

- Require a specific Signer account with a hard coded or configuration stored address (e.g., deployer / upgrade authority) via an Anchor address constraint.

- Alternatively, introduce a configuration account (PDA) that stores an admin address and enforce `ctx.accounts.initializer.key() == config.admin` in initialize.

3.2.5 The `reserved_amount` is obsolete

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: From the implementation:

- TokenVault has `reserved_amount: u64`.

- In `swap`:

  - `available_liquidity = vault_token_account.amount - to_vault.reserved_amount`.

  - Require `available_liquidity >= amount_after_fee`.

- In `withdraw_liquidity`:

  - `available = vault_token_account.amount - vault.reserved_amount`.

  - Require `amount <= available`.

- In `update_reserved_amount`:

  - Only check `new_reserved_amount <= vault_token_account.amount`.

So this is used for a portion of the vault balance is made unavailable for swaps made by users and withdrawals by the operations authority.

We do not see what is the benefit of this variable, other than maybe stop the swaps to a specific token.

Recommendation: If you still want to use this variable more like a swap limit then keeping it for swaps is completely fine. We recommend deleting it from the withdraw liquidity because this is anyway a limit that can be changed by the same authority that withdraws.

3.2.6 User pays for fee recipient ATA creation

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: At the first swap for a token that was just added, the user has to pay for the ATA creation for the fee recipient:

```
#[account(
    init_if_needed,
    payer = user,
    associated_token::mint = from_mint,
    associated_token::authority = fee_recipient
)]
  pub fee_recipient_token_account: Account<'info, TokenAccount>,
```

The users should never pay for account creation of the fee recipient.

Recommendation: Consider creating this ATA when the token is first added in `add_supported_token`.

3.2.7 Fees are rounding down

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The fees are rounding down when it should round up:

```
let fee_amount = (amount_in as u128)
    .checked_mul(pool.fee_rate as u128)
    .unwrap()
    .checked_div(10000)
    .unwrap() as u64;
```

Theoretically someone can split the swaps in low amounts to avoid paying large fees but in our case as everything is 1:1, the gain is not that much to worth the fee.

Recommendation: Consider rounding up the fee.

3.2.8 Token Accounts are not checked that they correspond to the user/authority

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In `swap` and other instructions, SPL token accounts are only constrained by mint, not that their owner is the user, for example:

```
#[account(
    mut,
    token::mint = from_mint,
)]
pub user_from_token_account: Account<'info, TokenAccount>,

#[account(
    mut,
    token::mint = to_mint,
)]
pub user_to_token_account: Account<'info, TokenAccount>,
```

Similar patterns appear in other instructions (e.g. deposit/withdraw), where we only check `token::mint = ...` but not `token::authority = user` (or the respective signer / pool / operations authority).

This means the program does not enforce on-chain that these token accounts are actually controlled by the expected wallet.

Recommendation: If this is a design choice then nothing should be done, otherwise account constraints should be implemented for every token account.

3.2.9 Deposit instruction checks can be circumvented

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The deposit instruction does not provide control over how liquidity enters the vaults because the actual vault balance can be increased at any time by the operations authority (or anyone) by calling the SPL Token program directly to transfer into `vault_token_account`, completely bypassing the `pool.liquidity_paused` check.

Recommendation: We do not recommend any action regarding this issue other than documenting the fact that deposit paused check can be circumvented.