Adevar Labs

Prepared for:
Valiant

December 03, 2025

Audit Report

Version: Audit report after fix review

Table of Contents

1. Executive Summary

About Valiant

Valiant ships with two tightly linked components: Cyclone is the launchpad layer that runs a bonding-curve sale with fixed liquidity, tick bounds, and completion price so new tokens can bootstrap supply in a controlled way, while Vortex is the concentrated-liquidity AMM that takes over once the sale graduates, offering full CLMM trading, fee accounting, and reward emissions for the newly launched pool.

Audit Summary

- Number of Findings: 5 total
  - Critical: 1
  - High: 0
  - Medium: 1
  - Low: 3
- Number of Enhancement Opportunities: 2

Risk Profile

The following table summarizes the distribution of identified vulnerabilities by risk level:

Risk Level	Count	Fixed	Acknowledged
Critical	1	1	0
High	0	0	0
Medium	1	1	0
Low	3	3	0

Overall Security Posture

After-fix Review

All identified issues have been fixed. The key changes are:

- Added an admin check so arbitrary configs and fee tiers can’t be created.
- Set the update authority to be immutable.
- Enforced minimum and maximum sqrt price bounds based on Cyclone boundaries instead of Vortex boundaries.
- Removed the unnecessary reload call.
- Added a check to ensure that when using a session, the funder is not equal to the owner.
- Validated that the graduated lower tick index is divisible by the configured tick spacing.

Before-fix Review

Cyclone's launch math reuses Orca's proven token constants and liquidity helpers, the graduation instruction carefully checks every seed, vault, fee account, and tick window before touching balances, and core structs and math helpers ship with unit tests that exercise serialization layouts and price/liquidity conversions. Within this otherwise solid design, the following issues were identified:

- Anyone could create arbitrary configs and fee tiers and potentially take over Cyclone and Vortex pools using those configs.
- The update authority could be permanently lost when using a FOGO session.
- The max and min sqrt price range taken from Vortex was out of bounds for Cyclone.
- When using sessions, there was no validation that the funding key must not be equal to the owner.
- graduation_lower_tick_index wasn't explicitly validated to be divisible by tick_spacing during initialization.
- A vault reload was performed in update_and_swap_cyclone even though it wasn't needed.

Launch Recommendations

After-fix Review

All five issues have been fixed, cyclone and vortex are safe to launch in production now.

Before-fix Review

Strongly Recommended:

• Fix the five identified issues

• Document trust assumptions and market owner responsibilities in protocol documentation

Audit Scope

- **Repository:** Valiant Dex
- **Commit Hash:** c3e5f5fdd52689c57ce7e48d6c7cfc0bf17660c1
- **Files/Modules in Scope:**
  - programs/vortex
- **Fix Review Commit Hashes**
  - 2d29436b940c57ddd386613a9628158a21b04c68
  - 8826d13993583232fe500044f831cce1cb9b21f5
  - dd336aa3ca854c230d8df312a0b3c83962b76f84
  - c5446d417884e0140bf6e968f8b19afda14e68c6
  - f92bd01cc8d1e26babb5961417b185cad3348baa
  - 06b17f053c8386d2108fb8c2dbe904cca0e0b517

2. Assumptions and Considerations

Trust Assumptions

1. Admins always pre-validate Cyclone parameters (liquidity, token threshold, tick bounds, completion price) so the bonding curve can actually finish where intended.
2. Keepers and frontends assume SOL legs are wrapped/unwrapped correctly, since the program treats SOL as SPL tokens and never syncs native balances for you.
3. Authority and fee accounts are assumed to be correct when passed in–there's no fallback if, for example, protocol fee vaults point to the wrong owner.

3. Severity Definitions

Each issue identified in this report is assigned a severity level based on two dimensions: **Impact** and **Likelihood**. These dimensions help project our team's understanding of both the potential consequences of a vulnerability and how likely a vulnerability is to be discovered and exploited in the real world.

Note: Enhancements represent non-blocking improvements–typically usability, observability, or defense-in-depth tweaks that do not pose an immediate asset risk but would improve the product's reliability and/or user experience if implemented.

Impact

Impact reflects the potential consequences of the issue–particularly on **project funds**, **user funds**, and the **availability or integrity** of the protocol.

- **High Impact**: Successful exploitation could result in a complete loss of user or protocol funds, disruption of core protocol functionality, or permanent loss of control over critical components.
- **Medium Impact**: Exploitation could cause significant disruption or partial loss of funds, but not a total compromise. May impact some users or non-core functionality.
- **Low Impact**: The issue has minor or negligible consequences. It may affect edge cases, expose metadata, or degrade performance slightly without putting funds or core logic at serious risk.

Likelihood

Likelihood reflects how easy a vulnerability is to discover and exploit by an attacker, as well as how economically attractive the exploit is to an attacker.

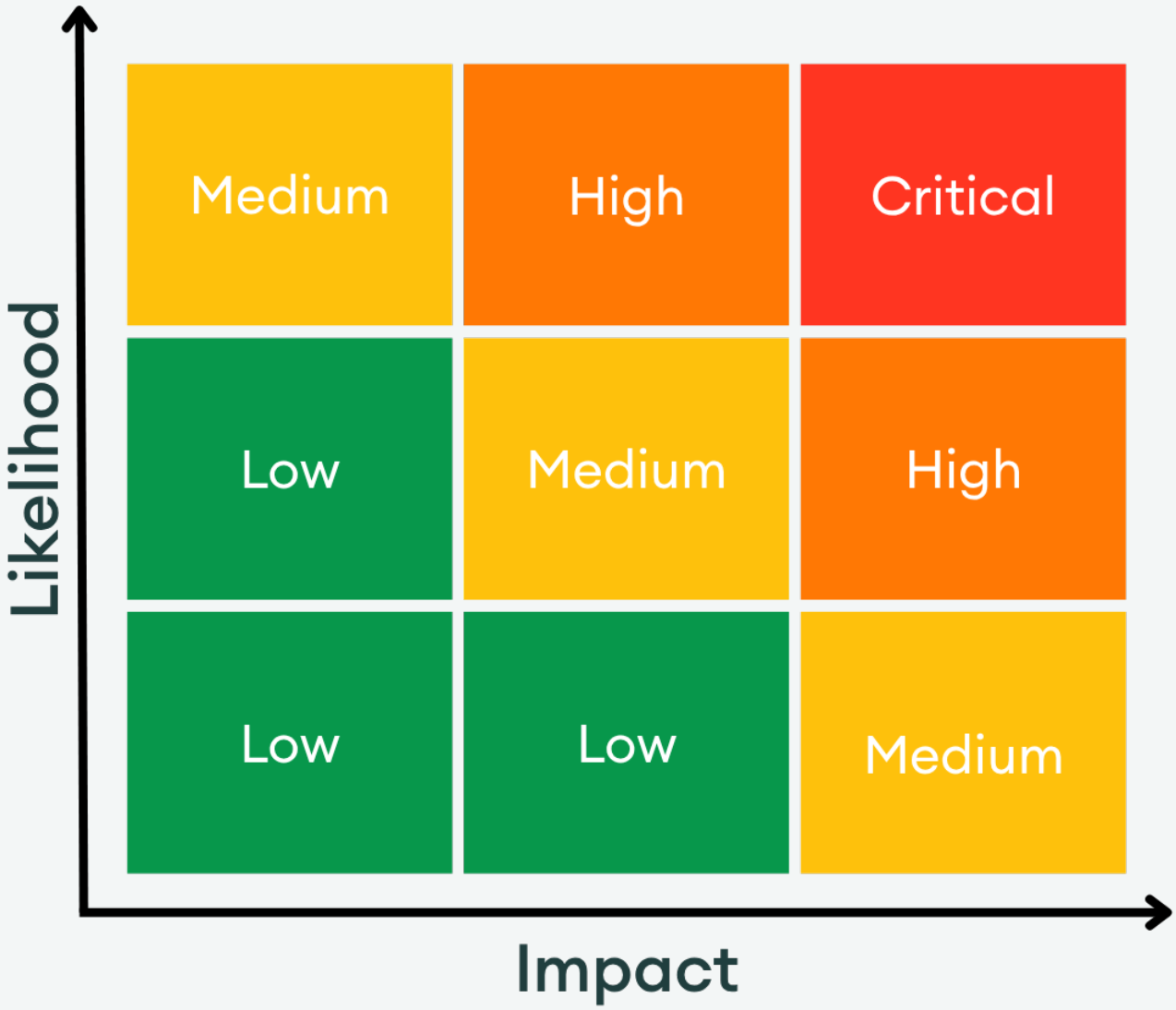
- **High Likelihood**: The vulnerability is trivially exploitable. This means it can be exploited by a wide range of actors without privileged access rights, with minimal capital requirements and low financial risks.
- **Medium Likelihood**: This type of vulnerability can be found and exploited with moderate effort. It might require a significant capital investment, but with manageable financial risk.
- **Low Likelihood**: Exploitation of these vulnerabilities is often technically unfeasible or requires highly specialized conditions. They may require extraordinary effort or a significant financial risk for an attacker, with a high chance of failure and minimal potential return.

Severity Classification Matrix

By combining **Impact** and **Likelihood**, we assign a severity level using the matrix below:

- **Critical**: High impact + high likelihood (e.g. a bug that could allow anyone to drain a substansial amount of protocol funds with minimal effort)
- **High**: High impact with medium likelihood, or medium impact with high likelihood
- **Medium**: Moderate impact and/or discoverability
- **Low**: Minimal impact or unlikely to be exploited

This structured approach helps teams prioritize fixes and mitigate the most dangerous threats first.



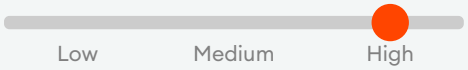
Severity Matrix

4. Findings

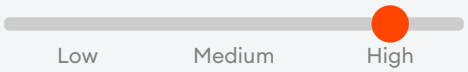
C01: Anyone can create arbitrary configs and fee tier and takeover the cyclone and vortex pools that use those configs

Status: Resolved

Impact:



Likelihood:



Severity: Critical

Location: https://github.com/AdevarLabs/vortex-reference/blob/15cde54f71166abe9aa91ea70340a66ea428b2dd/programs/vortex/src/instructions/initialize_config.rs#L5-L31

>_ initialize_config.rs RUST

```rust
use crate::state::*;

#[derive(Accounts)]
pub struct InitializeConfig<'info> {
    #[account(init, payer = funder, space = VortexConfig::LEN)]
    pub config: Account<'info, VortexConfig>,

    #[account(mut)]
    pub funder: Signer<'info>,

    pub system_program: Program<'info, System>,
}

pub fn handler(
    ctx: Context<InitializeConfig>,
    fee_authority: Pubkey,
    collect_protocol_fees_authority: Pubkey,
    reward_emissions_super_authority: Pubkey,
    default_protocol_fee_rate: u16,
) -> Result<()> {
    let config = &mut ctx.accounts.config;

    config.initialize(
        fee_authority,
        collect_protocol_fees_authority,
        reward_emissions_super_authority,
        default_protocol_fee_rate,
    )
}
```

Description:

```rust
pub fn handler(
    ctx: Context<InitializeConfig>,
    fee_authority: Pubkey,
    collect_protocol_fees_authority: Pubkey,
    reward_emissions_super_authority: Pubkey,
    default_protocol_fee_rate: u16,
) -> Result<()> {
    let config = &mut ctx.accounts.config;

    config.initialize(
        fee_authority,
        collect_protocol_fees_authority,
        reward_emissions_super_authority,
        default_protocol_fee_rate,
    )
}
```

There is no check to prevent an arbitrary user from creating configs and fee tiers.

If you can create arbitrary config where a malicious user can set variable of it's own for collect_protocol_fees_authority, reward_emissions_super_authority, default_protocol_fee_rate, fee_authority , the claim for the authority and fee would be held by the attacker.

Subsequently that user can create arbitrary fee tiers too both for the vortex and the cyclone

```rust
#[derive(Accounts)]
#[instruction(tick_spacing: u16)]
pub struct InitializeCycloneFeeTier<'info> {
    pub config: Box<Account<'info, VortexConfig>>,

    #[account(init,
      payer = funder,
      seeds = [b"cyclone_fee_tier", config.key().as_ref()],
      bump,
      space = CycloneFeeTier::LEN)]
    pub cyclone_fee_tier: Account<'info, CycloneFeeTier>,

    #[account(mut)]
    pub funder: Signer<'info>,

    #[account(address = config.fee_authority)]
    pub fee_authority: Signer<'info>,

    pub system_program: Program<'info, System>,
}
```

```rust
#[derive(Accounts)]
#[instruction(tick_spacing: u16)]
pub struct InitializeFeeTier<'info> {
    pub config: Box<Account<'info, VortexConfig>>,
```

... continued

```rust
    #[account(init,
      payer = funder,
      seeds = [b"fee_tier", config.key().as_ref(),
               tick_spacing.to_le_bytes().as_ref()],
      bump,
      space = FeeTier::LEN)]
    pub fee_tier: Account<'info, FeeTier>,

    #[account(mut)]
    pub funder: Signer<'info>,

    #[account(address = config.fee_authority)]
    pub fee_authority: Signer<'info>,

    pub system_program: Program<'info, System>,
}
```

The only check in both is

```rust
#[account(address = config.fee_authority)]
```

which will be bypassed since the attacker would set himself as the fee authority in the config.

Now any user trading in these pools can be locked out, might be forced to pay more fee and lose funds since the attacker controls all the following variables

```rust
default_fee_rate,
        default_protocol_fee_rate,
        initial_cyclone_tick_index,
        final_cyclone_tick_index,
        cyclone_liquidity,
        tokens_sold_threshold,
        graduation_lower_tick_index,
        default_graduation_fee_rate,
        default_graduation_creator_fee_rate,
```

```rust
#[account]
pub struct FeeTier {
    pub vortex_config: Pubkey,
    pub tick_spacing: u16,
    pub default_fee_rate: u16,
}
```

Recommendation:

Add the admin check for the config initialization

Developer Response:

Added an admin check that allows only the admin to add new configs and fee tiers.

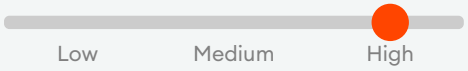
Fixed on commit: **2d29436b940c57ddd386613a9628158a21b04c68**

M01: Access to the update authority is permanently lost if we use fogo session

Status: Resolved

Impact:

Likelihood:



Severity: Medium

Location: https://github.com/AdevarLabs/vortex-reference/blob/15cde54f71166abe9aa91ea70340a66ea428b2dd/programs/vortex/src/instructions/initialize_cyclone_with_liquidity.rs#L114-L116

>_ initialize_cyclone_with_liquidity.rs RUST

```
112:            metadata: &ctx.accounts.metadata,
113:            mint: &ctx.accounts.token_mint_a.to_account_info(),
114:            mint_authority: &ctx.accounts.funder.to_account_info(),
115:            payer: &ctx.accounts.funder.to_account_info(),
116:            update_authority: (&ctx.accounts.funder.to_account_info(), true),
117:            system_program: &ctx.accounts.system_program.to_account_info(),
118:            rent: Some(&ctx.accounts.rent.to_account_info()),
```

Description:

During metadata initialisation in the init_cyclone_with_liqudity the mint_authority/payer/update_authority are set to the funder account.

However, if the funder signs via a Fogo session, the session PDA ends up recorded in the metadata.

Once that session expires, no one–not even the real user–can submit Metaplex UpdateMetadata instructions, because only the recorded authority can sign them.

Recommendation:

Return actual user pub key from the validate_session and continue assigning this value into the metadata..

Developer Response:

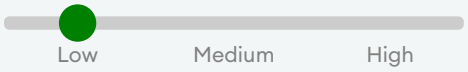
The metadata is not meant to be changed after token creation and will therefore be set to is_mutable = false, so neither the session nor the regular creator can modify it.

Fixed on commit: **8826d13993583232fe500044f831cce1cb9b21f5**

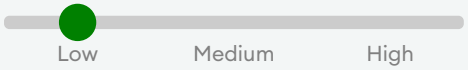
L01: Using the max and min sqrt price range from vortex is out of bounds of cyclone

Status: Resolved

Impact:



Likelihood:



Severity: Low

Location: https://github.com/AdevarLabs/vortex-reference/blob/15cde54f71166abe9aa91ea70340a66ea428b2dd/programs/vortex/src/manager/swap_manager.rs#L46-L54

>_ swap_manager.rs RUST

```
44:     }
45:
46:     let adjusted_sqrt_price_limit = if sqrt_price_limit == NO_EXPLICIT_SQRT_PRICE_LIMI
        T {
47:         if a_to_b {
48:             MIN_SQRT_PRICE_X64
49:         } else {
50:             MAX_SQRT_PRICE_X64
51:         }
52:     } else {
53:         sqrt_price_limit
54:     };
55:
56:     if !(MIN_SQRT_PRICE_X64..=MAX_SQRT_PRICE_X64).contains(&adjusted_sqrt_price_limit)
         {
```

Description:

In the swap_cyclone function there is logic that is copied from swap to make the swap math feasible, which is

RUST

```
let adjusted_sqrt_price_limit = if sqrt_price_limit == NO_EXPLICIT_SQRT_PRICE_LIMIT {
        if a_to_b {
            MIN_SQRT_PRICE_X64
        } else {
            MAX_SQRT_PRICE_X64
        }
```

but this is wrong in the context of cyclone since cyclone operates in very small price bounds relative to vrotex.

Price bounds for the cyclone are lower_tick_index and upper_tick_index which are set in the cyclone_fee_tier

Recommendation:

There can be two solutions, create two new constants for cyclone whose sqrt price correspond to the cyclones lower and upper ticks.

Or it can be completely removed too since after few lines we call this function

```rust
let (next_tick_sqrt_price, sqrt_price_target) =
        get_next_sqrt_prices(next_tick_index, adjusted_sqrt_price_limit, a_to_b);
```

and inside the get_next_sqrt_prices function if the sqrt price would be too extreme relative the sqrt price limit it will switch it to next tick price

```rust
fn get_next_sqrt_prices(
    next_tick_index: i32,
    sqrt_price_limit: u128,
    a_to_b: bool,
) -> (u128, u128) {
    let next_tick_price = sqrt_price_from_tick_index(next_tick_index);
    let next_sqrt_price_limit = if a_to_b {
        sqrt_price_limit.max(next_tick_price)
    } else {
        sqrt_price_limit.min(next_tick_price)
    };
    (next_tick_price, next_sqrt_price_limit)
}
```

Developer Response:

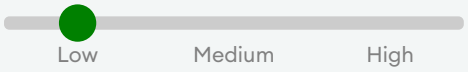
The code now calculates the minimum and maximum sqrt price from the Cyclone tick bounds.

Fixed on commit: **dd336aa3ca854c230d8df312a0b3c83962b76f84**

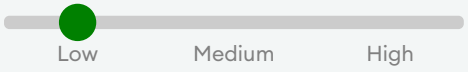
L02: When using sessions, there should be a validation that the key funding should not be equal to owner.

Status: Resolved

Impact:



Likelihood:



Severity: Low

Location: https://github.com/AdevarLabs/vortex-reference/blob/89775e79930d959a828e205a9749bbd92263214d/programs/vortex/src/instructions/open_position.rs#L11-L47

>_ open_position.rs RUST

```rust
#[derive(Accounts)]
pub struct OpenPosition<'info> {
    #[account(mut)]
    pub funder: Signer<'info>,

    /// CHECK: safe, the account that will be the owner of the position can be arbitrary
    pub owner: UncheckedAccount<'info>,

    #[account(init,
      payer = funder,
      space = Position::LEN,
      seeds = [b"position".as_ref(), position_mint.key().as_ref()],
      bump,
    )]
    pub position: Box<Account<'info, Position>>,

    #[account(init,
        payer = funder,
        mint::authority = vortex,
        mint::decimals = 0,
    )]
    pub position_mint: Account<'info, Mint>,

    #[account(init,
      payer = funder,
      associated_token::mint = position_mint,
      associated_token::authority = owner
    )]
    pub position_token_account: Box<Account<'info, TokenAccount>>,

    pub vortex: Box<Account<'info, Vortex>>,

    #[account(address = token::ID)]
    pub token_program: Program<'info, Token>,
    pub system_program: Program<'info, System>,
    pub rent: Sysvar<'info, Rent>,
    pub associated_token_program: Program<'info, AssociatedToken>,
```

>_ open_position.rs RUST

```
47: }
48:
49: /*
```

Description:

Consider the following code snippet from open_position

RUST

```rust
pub struct OpenPosition<'info> {
    #[account(mut)]
    pub funder: Signer<'info>,

    /// CHECK: safe, the account that will be the owner of the position can be arbitrary
    pub owner: UncheckedAccount<'info>,

    #[account(init,
      payer = funder,
      space = Position::LEN,
      seeds = [b"position".as_ref(), position_mint.key().as_ref()],
      bump,
    )]
    pub position: Box<Account<'info, Position>>,
```

The commet says that it can be arbitrary but there should be a check here that checks that it is not equal to funder if we are using a fogo session, in that case once the session is over the access to the owner key would be lost any way.

Recommendation:

Add a constratint that checks that the owner should not be equal to funder - and this check should be places at other places where fogo sessions are used too.

Developer Response:

Added a new validation and error code to ensure that the funder is not equal to the owner when using a session.

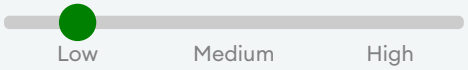
Fixed on commit: **06b17f053c8386d2108fb8c2dbe904cca0e0b517**

L03: graduation_lower_tick_index isn't explicitly validated to be divisible by tick_spacing during initialisation

Status: Resolved

Impact:

Likelihood:



Severity: Low

Location: https://github.com/AdevarLabs/vortex-reference/blob/15cde54f71166abe9aa91ea70340a66ea428b2dd/programs/vortex/src/state/cyclone_fee_tier.rs#L99

>_ cyclone_fee_tier.rs RUST

```
    97:         liquidity: u128,
    98:         tokens_sold_threshold: u64,
    99:         graduation_lower_tick_index: i32,
   100:     ) -> Result<()> {
   101:         if initial_cyclone_tick_index >= final_cyclone_tick_index
```

Description:

During **cyclone graduation**, the logic checks whether the **graduation lower tick** is divisible by the **tick spacing**:

RUST

```rust
fn get_tick(&self, tick_index: i32, tick_spacing: u16) -> Result<&Tick> {
    if !self.check_in_array_bounds(tick_index, tick_spacing)
        || !Tick::check_is_usable_tick(tick_index, tick_spacing)
    {
        return Err(ErrorCode::TickNotFound.into());
    }
}
```

However, during the **bonding curve setup**, the graduation_lower_tick_index isn't explicitly validated to ensure it's divisible by the expected **tick spacing**.

If an improper tick is provided, the fee initialisation and consequent cyclone bonding curve state could take place, meanwhile the **graduation process** could revert.

Recommendation:

During the set bonding curve, execute the check which will ensure that the graduation lower tick is divisible by the tick_spacing.

Developer Response:

Validation is added to ensure **graduation lower tick** is divisible by the **tick spacing**

Fixed on commit: **c5446d417884e0140bf6e968f8b19afda14e68c6**

5. Enhancement Opportunities

E01: Vault reload is not needed in the update_and_swap_cyclone function

Location: https://github.com/AdevarLabs/vortex-reference/blob/15cde54f71166abe9aa91ea70340a66ea428b2dd/programs/vortex/src/util/swap_utils.rs#L46

>_ swap_utils.rs RUST

```
44:     )?;
45:
46:     token_vault_a.reload()?;
47:
48:     Ok(())
```

Description:

in the update_and_swap_cyclone there is a vault state reload but this is not needed since the state for the vault is not accessed or read after the reload, we call this function and emit event after it and that evernt doesn't read from the vault state

RUST

```
update_and_swap_cyclone(
        cyclone,
        &ctx.accounts.token_authority,
        &ctx.accounts.token_owner_account_a,
        &ctx.accounts.token_owner_account_b,
        &mut ctx.accounts.token_vault_a,
        &mut ctx.accounts.token_vault_b,
        &ctx.accounts.token_program,
        swap_update,
        a_to_b,
        ctx.accounts.program_signer.as_ref(),
    )?;

    emit!(Traded {
        vortex: cyclone.key(),
        a_to_b,
        pre_sqrt_price,
        post_sqrt_price: cyclone.sqrt_price,
        input_amount,
        output_amount,
        input_transfer_fee: 0,
        output_transfer_fee: 0,
        lp_fee,
        protocol_fee,
    });
```

Potential Benefit:

removing the redundant code will lead to more CU efficiency.

Recommendation:

remove the reload line.

Developer Response:

The reload call at the end of update_and_swap_cyclone has been removed.

E02: Additional check must be enforced that bonding_curve_complete_sqrt_price isn't higher than bonding_curve_upper_tick_index

Location: https://github.com/AdevarLabs/vortex-reference/blob/15cde54f71166abe9aa91ea70340a66ea428b2dd/programs/vortex/src/instructions/initialize_cyclone_with_liquidity.rs#L195-L196

>_ initialize_cyclone_with_liquidity.rs RUST

```
193:         ctx.accounts.token_vault_b.key(),
194:         bonding_curve_lower_tick_index,
195:         bonding_curve_upper_tick_index,
196:         bonding_curve_complete_sqrt_price,
197:         graduation_tick_lower_index,
198:         graduation_tick_upper_index,
```

Description:

Valid admin init cyclone_fee_tier, and set:

- initial_cyclone_tick_index
- final_cyclone_tick_index

However, during initialisation it never checks that final_cyclone_tick_index isn't lesser than the bonding_curve_complete_sqrt_price

The following scenario could occur, that fee tier would be initialised with following state:

RUST

```
bonding_curve_complete_sqrt_price > final_cyclone_tick_index
```

In such scenario, when user would execute swap, he could never achieve complete sqrt price, since it'll always hit this check.

RUST

```
if (a_to_b && next_tick_sqrt_price >= curr_sqrt_price)
    || (!a_to_b && next_tick_sqrt_price <= curr_sqrt_price) {
    return Err(ErrorCode::InvalidSqrtPriceLimitDirection.into());
}
```

```
if (a_to_b && next_tick_sqrt_price >= curr_sqrt_price)
    || (!a_to_b && next_tick_sqrt_price <= curr_sqrt_price) {
    return Err(ErrorCode::InvalidSqrtPriceLimitDirection.into());
}
```

If the "completion" price exceeds the upper tick, curr_sqrt_price reaches the tick before the completion condition ever turns true. From that point on, the check above rejects every swap–effectively DoS'ing further trades while the cyclone still believes it's unfinished.

Potential Benefit:

Additional check would prevent the potential DoS scenario

Recommendation:

During the fee tier initialisation, ensure that bonding_curve_complete_sqrt_price produced based on liquidity/threshold isn't higher than bonding_curve_upper_tick_index

Developer Response:

Validation is added to ensure that the sqrt_price_at_final_cyclone_tick_index correspond to the bonding_curve_complete_sqrt_price

Fixed on commit: **f92bd01cc8d1e26babb5961417b185cad3348baa**

About Us

About Adevar Labs

Adevar Labs is a boutique blockchain security firm specializing in web3 audits.

Built by a mix of experienced professionals in traditional enterprise and crypto natives who have contributed to some of the most critical projects in blockchain infrastructure.

Our team's background spans companies like Bitdefender, Asymmetric Research, Quantstamp, Chainproof, and Juicebox, and includes experience securing smart contracts, bridges, and L1 and L2 protocols across ecosystems like Solana, Ethereum, Polkadot, Cosmos, and MultiversX.

With over 100 audits completed and a portfolio that includes custom fuzzers, exploit modeling, and runtime testing frameworks, Adevar Labs brings both depth and precision to every engagement.

Our auditors have discovered critical vulnerabilities, built high-impact tooling, and placed in top positions in premier audit competitions including Code4rena and Sherlock.

Team members hold distinctions such as PhDs in software protection, and key roles at Fortune 500 companies, and leadership of flagship conferences like ETH Bucharest.

With team members having publications with over 1,100 academic citations and trusted by projects securing over $500M in on-chain value, Adevar Labs blends elite technical rigor with real-world security impact.

We also collaborate with some of the best independent security researchers in the web3 space.

Projects may optionally request specific contributors to be part of their audit.

Audit Methodology

Our audit methodology is specialized to provide thorough security assessments of Solana programs. We focus explicitly on rigorous manual analysis, detailed threat modeling, and careful validation of implemented fixes to ensure your programs operate securely and as intended.

1. Program Context and Architecture Analysis

Our auditors begin by deeply examining your Solana program's documentation, intended functionality, and account design. We meticulously map out program interactions, instruction processing flows, and state management logic. Special attention is given to understanding how your program interfaces with critical Solana system programs, such as the SPL Token Program, Stake Program, and System Program. Last but not least we check external integrations with other Solana projects and verify if the inputs and return values are handled properly.

2. Threat Modeling

We conduct targeted threat modeling tailored specifically for Solana's execution environment. We carefully define attacker capabilities and identify potential vulnerabilities that may arise from Solana-specific issues, including but not limited to:

- Unauthorized account data manipulation
- Improper ownership or signer verification
- Misuse of Program-Derived Addresses (PDAs)

- Incorrect use of Cross-Program Invocations (CPI)
- Failure to adequately handle account privileges or account states
- Risks stemming from rent-exemption and account initialization logic

3. In-depth Manual Security Review

Our experienced auditors perform an extensive manual security review of your Rust-based Solana programs. This involves a comprehensive line-by-line inspection of source code, focusing on common Solana vulnerabilities including, but not limited to:

- Missing or insufficient ownership checks
- Inadequate signer checks
- Incorrect handling of CPI calls and invocation privileges
- Arithmetic and integer overflow or underflow errors
- Unsafe deserialization and serialization of account data structures
- Improper token transfers and SPL-token logic issues
- Logic flaws in financial operations or state transitions
- Edge-case handling in instruction input validation
- Potential denial-of-service vectors related to transaction execution and account handling

During this stage, we clearly document any discovered vulnerabilities, including detailed descriptions, precise severity ratings, and recommendations for secure implementation. In situations where it is not clear how the vulnerability might be exploited we may also include a detailed proof-of-concept exploit including code snippets and instructions on how the exploit could be performed.

4. Detailed Fix Review and Validation

After the initial audit and your team's subsequent remediation efforts, we perform a comprehensive fix review to ensure the vulnerabilities identified have been effectively resolved.

We verify each fix individually, confirming that:

- Corrections effectively eliminate the security risks
- Changes do not inadvertently introduce new vulnerabilities or regressions
- The fixes align closely with Solana best practices and secure coding guidelines

Our detailed validation ensures that security improvements are robust, complete, and aligned with best practices specific to the Solana development ecosystem.

Confidentiality Notice

This report, including its content, data, and underlying methodologies, is subject to the confidentiality and feedback provisions in your agreement with Adevar Labs. These materials are not to be disclosed, extracted, copied, or distributed except to the extent expressly authorized by Adevar Labs.

Legal Disclaimer

The review and this report are provided by Adevar Labs on an as-is, where-is, and as-available basis. To the fullest extent permitted by law, Adevar Labs disclaims all warranties, expressed or implied, in

connection with this report, its content, and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

You agree that access to and/or use of the report and other results of the review, including any associated services, products, protocols, platforms, content, and materials, will be at your sole risk.

FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE. This report is based on the scope of materials and documentation provided for a limited review at the time provided.

You acknowledge that blockchain technology remains under development and is subject to unknown risks and flaws. Adevar Labs does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party, or any open-source or third-party software, code, libraries, materials, or information accessible through the report. As with the purchase or use of a product or service in any environment, you should use your best judgment and exercise caution where appropriate.