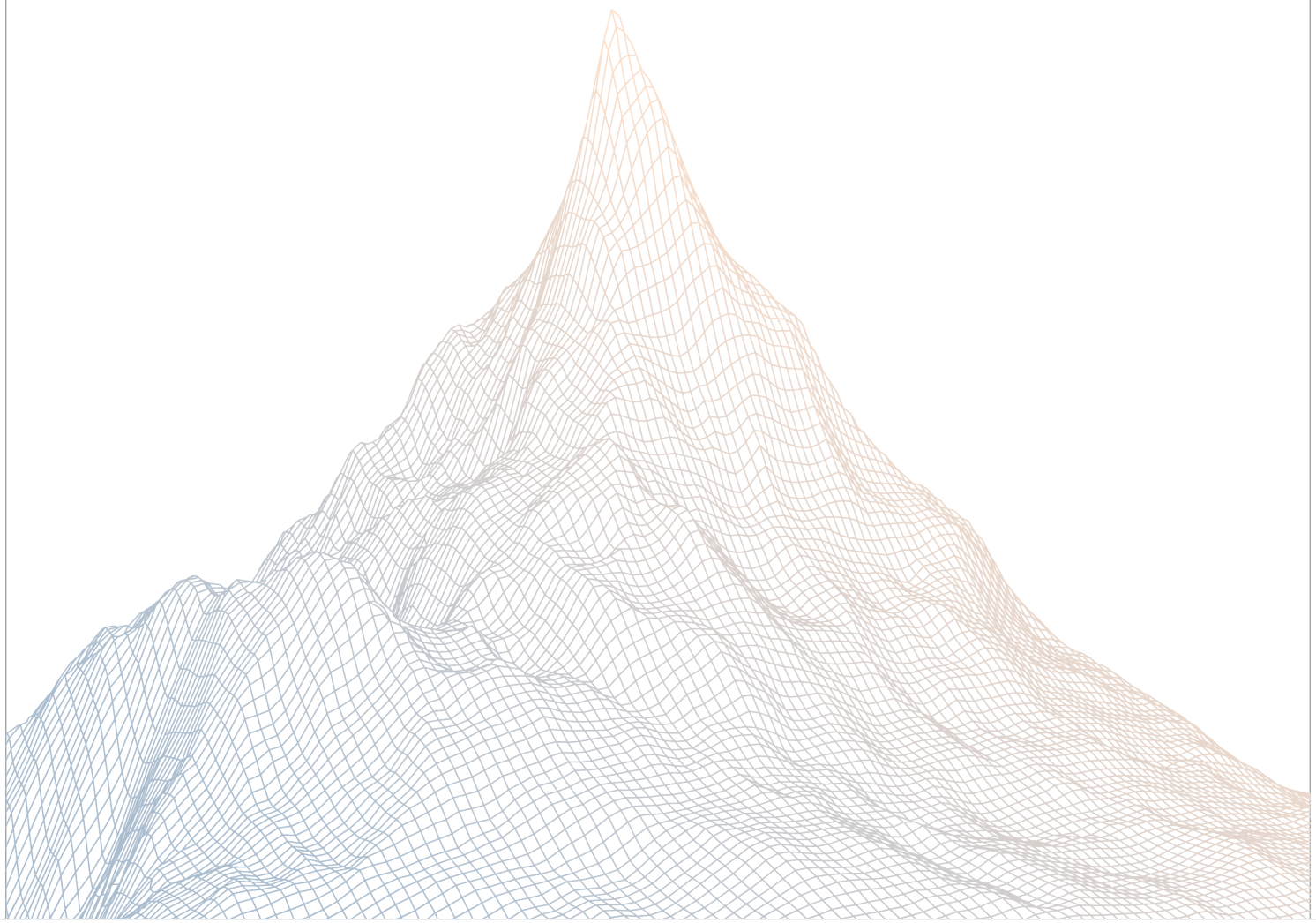
Zenith

Talus

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: March 19th to March 31st, 2025

AUDITED BY: Matte
Christian Vari

Contents

1 Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Talus

Talus has a vision of AI agents powering a new age of user-centric internet. We stand to serve the user. That is our goal. However, we realize that achieving this mission will require collaboration. Therefore we are building applications directly for users, but also the infrastructure and tooling that will empower an ecosystem of developers to create best-in-class AI-enhanced applications for the user.

To that end, Talus is the onchain platform for AI agents while Nexus is the developer framework for building these agents.

The AI agent space is broad and competitive, but we see the unique market opportunity to augment AI agents with the unique value proposition blockchains offer. A distinctive approach, the Talus way. For our community of users, by our community of developers, together with our partners.

2.2 Scope

The engagement involved a review of the following targets:

Target	iao
Repository	https://github.com/Talus-Network/iao
Commit Hash	`01adb8b6177f23722f6d2db5531db541dfaf3047`

Files	`iao/sources/iao.rs`
	`utils/sources/bonding_curve.rs`
	`utils/sources/oracle.rs`
	`utils/sources/minimal_fixed_point.rs`
	`utils/sources/governance.rs`

2.3 Audit Timeline

March 19th, 2025	Audit start
March 31st, 2025	Audit end
April 29th, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	2
High Risk	8
Medium Risk	13
Low Risk	10
Informational	4
Total Issues	**37**

3

Findings Summary

ID	Description	Status
C-1	Incorrect coin supply accounting leads to incorrect prices	Resolved
C-2	Duplicate AssetAdminCap forging using a malicious AssetConfig	Resolved
H-1	Calculation for Fixed bonding curve will violate invariant	Resolved
H-2	Certain Trigger Metrics could prevent curve from completing	Resolved
H-3	Liquidity pool can be created ahead causing graduation to fail	Resolved
H-4	Missing slippage check for liquidity_pool::swap	Resolved
H-5	It is not possible to graduate IAOs	Resolved
H-6	Incorrect price calculation	Resolved
H-7	It is possible to perform swaps if the protocol or the targeted IAO is paused	Resolved
H-8	Incorrect config access in update_parameters prevents parameter updates	Resolved
M-1	burn_tokens() will burn tokens instantly instead of scheduling	Resolved
M-2	buy_back_burn_bd and buy_back_burn_lp fail to reset last_burn_timestamp_ms	Resolved
M-3	Incorrect check in can_graduate()	Resolved
M-4	buy_back_burn_lp can be griefed by pool manipulation	Resolved
M-5	revenue::total_collected is incorrectly increased in buy_back_burn_lp() and buy_back_burn_bd()	Resolved
M-6	Missing IAO pause check in buy-back and burn functions	Resolved
M-7	Inconsistent use of global AdminCap instead of AssetAdminCap for IAO operations	Resolved
M-8	Public access to the initialize function of the liquidity_pool module could lead to inconsistent states	Resolved

ID	Description	Status
M-9	Static custom trigger metrics break level progression mechanism	Resolved
M-10	Excessive oracle data age threshold can lead to price manipulation risk	Resolved
M-11	Cetus oracle allows arbitrary operator Input	Resolved
M-12	Frontrunning in oracle updates	Resolved
M-13	Price inconsistencies and risks due to token minting	Resolved
L-1	Fee can be bypassed due to rounding down of fees	Resolved
L-2	Sufficient supply buffer should be provided	Resolved
L-3	buy_back_burn_lp should set by_amount_in = true	Resolved
L-4	Unrestricted access to graduate function may cause inconsistent state	Resolved
L-5	Missing validation on supply_threshold_bps	Resolved
L-6	Missing input validation for update_config of the pools module	Resolved
L-7	Missing pause check in protocol fee collection	Resolved
L-8	Missing fee validation in init_config of the iao module	Resolved
L-9	Lack of operator revocation mechanism in oracle module enables persistent malicious activity	Resolved
L-10	Unbounded iteration in level evaluation	Resolved
I-1	Consider locking the LP positions upon graduation	Resolved
I-2	Orphaned graduation records	Resolved
I-3	Lack of event emission on pause resume	Resolved

ID	Description	Status
I-4	Inconsistent event identifier usage	Resolved

4

Findings

4.1 Critical Risk

A total of 2 critical risk findings were identified.

[C-1] Incorrect coin supply accounting leads to incorrect prices

SEVERITY: Critical	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- sui/iao/sources/iao.move

Description:

The current implementation of the `buy_back_burn_bd` function in the bonding curve mechanism presents a vulnerability due to an incomplete update of internal state following token burns. Specifically, while `CoinY` tokens acquired through `revenue_manager.withdraw` and subsequent `bonding_curve::swap` operations are successfully destroyed via `burn_manager.burn`, the total token `supply` tracked by the bonding curve remains unchanged.

Since price calculations, such as those in `calculate_current_price`, depend on the token `supply`, this oversight leads to pricing based on an inflated `supply`. As a result, users executing swaps receive artificially lowered prices. This mispricing exposes the protocol to potential financial exploitation, creates arbitrage opportunities, and distorts the protocol's economic model.

The same vulnerability is observed in the `execute_burn` function, where `burn_manager.execute_burn is used`, yet the bonding curve's token `supply` is not reduced accordingly. This systemic inconsistency compromises the integrity and accuracy of price-related computations.

Recommendations:

We recommend updating the implementations of `burn_manager.burn` and `burn_manager.execute_burn` to ensure that every token burn operation triggers a corresponding deduction in the bonding curve's internal token `supply` record.

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in

commit 033d0c95.

Zenith: Verified

[C-2] Duplicate `AssetAdminCap` forging using a malicious `AssetConfig`

SEVERITY: Critical	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- sui/iao/sources/iao.move

Description:

An attacker can exploit the current registration process by invoking the `register_asset` function with a coin pair that hasn't been used before while passing an `AssetConfig` that contains the same `asset_id` as a valid, already registered asset. Since the uniqueness check in the registry is based solely on the token pair (derived from the coin types) rather than the `asset_id`, this malicious registration bypasses the duplicate asset check and results in issuing a new `AssetAdminCap` for the same asset.

Steps to reproduce:

1. Identify a valid registration:

A legitimate asset is registered with a specific `asset_id` using a particular coin pair (e.g., `CoinX` and `CoinY`).

2. Craft a malicious registration:

An attacker creates an `AssetConfig` containing the same `asset_id` as the valid asset but uses a different coin pair (by altering the type parameters).

3. Call `register_asset`:

The attacker invokes `register_asset` with the malicious `AssetConfig` (with the duplicate `asset_id`) and a coin pair that differs from the valid one

4. Outcome:

The registry's uniqueness check (based on the token pair) does not detect the duplicate `asset_id`, and the function successfully registers the asset again, issuing a new `AssetAdminCap` for the same asset.

Multiple `AssetAdminCap` instances for the same asset mean that the attacker can obtain administrative control over asset-specific operations (e.g., fee collection, token burns).

Recommendations:

We recommend modifying the registration process to include a check that ensures an asset with a given `asset_id` can only be registered once, regardless of the coin pair.

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit `5471395c`.

Zenith: Verified.

4.2 High Risk

A total of 8 high risk findings were identified.

[H-1] Calculation for Fixed bonding curve will violate invariant

SEVERITY: High	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: High

Target

- bonding_curve.move#L668
- bonding_curve.move#L685

Description:

The `Fixed` bonding curve performs the buy/sell using the constant product formula (`x * y ⩽ k`), where the invariant `k = curve.liquidity_reserve * curve.curve_tokens`.

However, in `calculate_fixed_buy_output()`, the calculation `new_curve_token = (old_liquidity_reserve * old_curve_tokens)/new_liquidity_reserve` incorrectly rounds down. As `new_curve_token` decreases due to the rounding down, it causes `k` to decrease too and violates the constant product invariant (`x * y ⩽ k`).

The same issue also applies for `calculate_fixed_sell_output()`.

In the extreme case, this could cause the `curve_token` reserve amount to decrease progressively and get depleted.

Example Scenario

- curve_tokens = 120,
- liquidity_reserve = 100,
- k = 120*100 = 12000

Now, if we swap 19 CoinX for CoinY,

- new_liquidity_reserve = 100 + 19 = 119
- new_curve_tokens = 12000 / 119 = 100 (rounded down from ~100.84)
- token_out = 120 - 100 = 20

- new_k = 119 * 100 = 11900

Now this violates the variant as `new_k` is lower the the previous `k`.

```
/// Calculate fixed curve buy output
fun calculate_fixed_buy_output<CoinX, CoinY>(
    curve: &BondingCurve<CoinX, CoinY>,
    input_amount: u64
): (u64, u64) {
    let old_liquidity_reserve = balance::value(&curve.liquidity_reserve);
    let old_curve_tokens = balance::value(&curve.curve_tokens);

    let new_liquidity_reserve = old_liquidity_reserve + input_amount;
    let new_curve_tokens =
    minimal_fixed_point::mul_div(old_liquidity_reserve, old_curve_tokens,
    new_liquidity_reserve, false);
    let token_out = old_curve_tokens - new_curve_tokens;

    assert!(token_out > 0, EInvalidAmount);
    (token_out, 0) // No refund in fixed curve
}

/// Calculate fixed curve sell output
fun calculate_fixed_sell_output<CoinX, CoinY>(
    curve: &BondingCurve<CoinX, CoinY>,
    input_amount: u64
): u64 {
    let old_liquidity_reserve = balance::value(&curve.liquidity_reserve);
    let old_curve_tokens = balance::value(&curve.curve_tokens);
    assert!(old_liquidity_reserve > 0, EInsufficientLiquidity);

    let new_curve_tokens = old_curve_tokens + input_amount;
    let new_liquidity_reserve =
    minimal_fixed_point::mul_div(old_curve_tokens, old_liquidity_reserve,
    new_curve_tokens, false);
    let x_out = old_liquidity_reserve - new_liquidity_reserve;

    assert!(x_out > 0, EInsufficientLiquidity);
    x_out
}
```

Recommendations:

```
/// Calculate fixed curve buy output
fun calculate_fixed_buy_output<CoinX, CoinY>(
    curve: &BondingCurve<CoinX, CoinY>,
    input_amount: u64
): (u64, u64) {
    let old_liquidity_reserve = balance::value(&curve.liquidity_reserve);
    let old_curve_tokens = balance::value(&curve.curve_tokens);

    let new_liquidity_reserve = old_liquidity_reserve + input_amount;
    let new_curve_tokens = minimal_fixed_point::mul_div(old_liquidity_
        reserve, old_curve_tokens, new_liquidity_reserve, false);
    let new_curve_tokens = minimal_fixed_point::mul_div(old_liquidity_
        reserve, old_curve_tokens, new_liquidity_reserve, true);
    let token_out = old_curve_tokens - new_curve_tokens;

    assert!(token_out > 0, EInvalidAmount);
    (token_out, 0) // No refund in fixed curve
}

/// Calculate fixed curve sell output
fun calculate_fixed_sell_output<CoinX, CoinY>(
    curve: &BondingCurve<CoinX, CoinY>,
    input_amount: u64
): u64 {
    let old_liquidity_reserve = balance::value(&curve.liquidity_reserve);
    let old_curve_tokens = balance::value(&curve.curve_tokens);
    assert!(old_liquidity_reserve > 0, EInsufficientLiquidity);

    let new_curve_tokens = old_curve_tokens + input_amount;
    let new_liquidity_reserve = minimal_fixed_point::mul_div(old_curve_
        tokens, old_liquidity_reserve, new_curve_tokens, false);
    let new_liquidity_reserve = minimal_fixed_point::mul_div(old_curve_
        tokens, old_liquidity_reserve, new_curve_tokens, true);
    let x_out = old_liquidity_reserve - new_liquidity_reserve;

    assert!(x_out > 0, EInsufficientLiquidity);
    x_out
}
```

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit [9811a30a](9811a30a).

Zenith: Verified.

[H-2] Certain Trigger Metrics could prevent curve from completing

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- level.move#L30-L37

Description:

Bonding curve can be configured with levels based on a trigger metric, that will allow it to be completed when triggered.

For Pool trigger metric, if the curve is configured with Pool trigger metric, it will not be completable even when all the available supply are bought. That is because the Pool trigger metric (curve_tokens) will decreases over time as the curve_tokens get bought. And the trigger is checked using `current_value ⩾ trigger_value`. This will prevent the curve from completing as it will not get triggered.

For Price trigger metric, if the price level is set too high, it is possible for it to be uncompletable, if it cannot be triggered, despite depleting the supply.

For TradeVolume trigger, it is also possible for the volume to be low over a long period and yet deplete the supply.

Recommendations:

Remove Pool trigger metric as Supply should be used instead.

Consider allowing the curve to be mark completed when a specified target supply is minted, for the case of Price and TradeVolume triggers.

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit `75db61b6`.

Zenith: Verified.

[H-3] Liquidity pool can be created ahead causing graduation to fail

SEVERITY: High	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: High

Target

- liquidity_pool.move#L78
- liquidity_pool.move#L26

Description:

It is possible for the Cetus liquidity pools to be created by an attacker using the tokens bought from the bonding curve.

This will prevent the bonding curve's `graduate()` from proceeding due to an existing liquidity pool for the token pair and tick-spacing. The result is that the funds (liquidity reserve and curve tokens) will be stuck within the bonding curve. That will then require the protocol admin to withdraw the funds.

The same issue also applies for `liquidity_pool::initialize()`.

Recommendations:

Use `create_pool_v2_with_creation_cap` when there is an existing pool for the token pair.

Cetus allows the coin issuers to reclaim the ability to create pool using `create_pool_v2_with_creation_cap`.

Talus Network: Resolved with @7042e442a9b...

Zenith: Verified.

[H-4] Missing slippage check for liquidity_pool::swap

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- liquidity_pool.move#L134

Description:

`liquidity_pool::swap` uses `pool::flash_swap` and `pool::repay_flash_swap` to perform a swap in the Cetus liquidity pool.

However, it is missing slippage check, and this could cause the swap to be affected by price impact. In the worst case, an attack could perform a sandwich attack to cause the swap to receive significantly less output amount than expected.

Slippage check is supposed to be implemented for `flash_swap` as stated in Cetus docs as follows:

> In the Cetus CLMM contract, both the flash swap and the repay flash swap functions do not care for slippage. If you integrate these functions into your contract, you must implement your own slippage checks.

Recommendations:

Implement slippage check for `liquidity_pool::swap` by introducing a `min_out_amount` parameter that can be checked again the `out_amount`.

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit `18e36d16`.

Zenith: Verified.

[H-5] It is not possible to graduate IAOs

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- sui/iao/sources/iao.move

Description:

The `graduate` function in the IAO module includes an assertion intended to prevent the re-execution of the graduation process. However, the current check is incorrectly implemented as:

```
assert!(iao_r1.graduation_status ≠ GraduationStatus::NOT_STARTED,
    EIAOAlreadyGraduated);
```

This logic erroneously allows graduation to proceed only if the status is not `NOT_STARTED`, which implies it has either already begun or completed. As a result, the function will fail before any graduation has occurred, exactly the opposite of the intended behavior.

The intended logic should block execution after graduation has been initiated or completed. Consequently, it would not be possible to actually graduate IAOs.

Recommendations:

We recommend correcting the assertion logic to:

```
assert!(iao_r1.graduation_status ≠ GraduationStatus::NOT_STARTED,
    EIAOAlreadyGraduated);
assert!(iao_r1.graduation_status = GraduationStatus::NOT_STARTED,
    EIAOAlreadyGraduated);
```

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit 810e4ec3.

Zenith: Verified

[H-6] Incorrect price calculation

SEVERITY: High	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: High

Target

- sui/pools/sources/bonding_curve.move

Description:

The bonding curve module is designed to maintain an accurate record of the current token price through the `last_price` field. However, the current implementation calculates `last_price` before the bonding curve's token balance is updated to reflect tokens that have been split off for a trade. As a result, the computed price does not represent the post-swap state.

Specifically, the `buy` function executes the `update_curve_state` before splitting the `token_out` balance from the `curve.curve_tokens`.

Recommendations:

We recommend that the ordering of operations be reviewed so that the `curve.curve_tokens` balance used in the `last_price` calculation reflects the state after the swap is fully processed.

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit `3a1ee990`.

Zenith: Verified

[H-7] It is possible to perform swaps if the protocol or the targeted IAO is paused

SEVERITY: High	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: High

Target

- sui/iao/sources/iao.move

Description:

The current implementation of the IAO `swap` function directly calls the bonding curve's `swap` function without performing pause or configuration checks. This omission means that:

- The global configuration pause state is not verified.
- The `CoinX` specific configuration pause is not enforced.
- The IAO's own pause state is not checked.

As a result, users could execute token swaps on the bonding curve even when the system or the specific IAO should be paused. Moreover, the underlying `bonding_curve::swap` function is not restricted to ensure these checks are enforced.

Recommendations:

We recommend adding explicit checks in the IAO swap function to verify that:

- The global configuration is not paused.
- The configuration for CoinX is active.
- The IAO itself is not paused.

Modify the bonding curve `swap` function or restrict its visibility so that it can only be called after these validations have been performed at the IAO level.

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit `736f8855`.

Zenith: Verified

[H-8] Incorrect config access in `update_parameters` prevents parameter updates

SEVERITY: High	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: High

Target

- sui/iao/sources/config.move

Description:

The update_parameters function directly accesses the Config<CoinX> object using:

```
let config_coin_x = dof::borrow_mut<ConfigKey<CoinX>,
    Config<CoinX>>(&mut config.id, ConfigKey<CoinX> {});
```

This assumes that the `Config<CoinX>` dynamic object field is attached directly to `GlobalConfig`, which contradicts the architecture used throughout the module. In all other functions, `Config<CoinX>` is stored inside an `ObjectBag` within `GlobalConfigR1`, which is itself stored as a dynamic field under the `GlobalConfig` object at the key `VERSION`.

This incorrect access path will cause a runtime error when the function fails to locate the expected `Config<CoinX>` at the wrong level of the object hierarchy.

Recommendations:

We recommend applying the following diff:

```
let config_coin_x = dof::borrow_mut<ConfigKey<CoinX>, Config<CoinX>>(&mut
    config.id, ConfigKey<CoinX> {});
let global_config_r1
 = df::borrow_mut<u16, GlobalConfigR1>(&mut config.id, VERSION);
let config_coin_x =

    object_bag::borrow_mut<ConfigKey<CoinX>, Config<CoinX>>(&mut global_config_
    r1.configs,ConfigKey<CoinX> {});
```

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit `9fec8435`.

Zenith: Verified

4.3 Medium Risk

A total of 13 medium risk findings were identified.

[M-1] burn_tokens() will burn tokens instantly instead of scheduling

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- iao.move#L669

Description:

The comments for `burn_tokens()` indicates that it is supposed to schedule the burn (i.e. transfer to `burn_vault`) and then `execute_burn()` will be used to perform the actual burn on a periodic basis (i.e. burn the tokens in `burn_vault`).

However, it calls `burn_manager.burn()`, which will actually burn the tokens immediately, instead of scheduling it.

```
/// Burn tokens
/// This just schedules the burn, the actual burn is done in the execute_burn
    function
public fun burn_tokens<CoinX, CoinY>(
    config: &mut GlobalConfig,
    iao: &mut IAO<CoinX, CoinY>,
    coin: Coin<CoinY>,
    clock: &Clock,
) {
    config.check_if_global_paused();
    config.check_if_config_paused<CoinX>();

    let iao_r1 = df::borrow_mut<u16, IAO_R1<CoinX, CoinY>>(&mut iao.id,
    VERSION);
    assert!(iao_r1.is_paused == false, ETradingPaused);

    let iao_r1 = df::borrow_mut<u16, IAO_R1<CoinX, CoinY>>(&mut iao.id,
```

```
    VERSION);
    let asset_id = iao_r1.asset.asset_id;
    //@audit this will burn the tokens instantly
    iao_r1.burn_manager.burn(&mut iao_r1.treasury_cap, coin, asset_id,
    clock);

}
```

Recommendations:

Update `burn_tokens()` to schedule the burn by transferring to `burn_vault`.

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit 16673cdc.

Zenith: Verified.

[M-2] buy_back_burn_bd and buy_back_burn_lp fail to reset last_burn_timestamp_ms

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- iao.move#L390
- iao.move#L466

Description:

The functions `buy_back_burn_bd` and `buy_back_burn_lp` are used to perform buyback of `CoinY` using deposited `CoinX` revenue, and then burn the bought `CoinY`. It performs the check `revenue_manager.check_if_can_burn` to ensure that it can only be trigger after `min_burn_interval_ms`.

However, both `buy_back_burn_bd` and `buy_back_burn_lp` fail to reset `last_burn_timestamp_ms` by calling `burn_revenue()`. This will allow the buybacks to be performed anytime, instead of on a periodic basis.

Recommendations:

Reset the `last_burn_timestamp_ms` in `buy_back_burn_bd` and `buy_back_burn_lp`.

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit `e98aa0c9`.

Zenith: Verified.

[M-3] Incorrect check in `can_graduate()`

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- bonding_curve.move#L879-L886

Description:

`can_graduate()` is incorrect as when the curve is completed, supply and liquidity_reserve will be non-zero and certain case curve_tokens will be non-zero.

This will prevent `liquidity_pool::initialize()` from proceeding for a completed bonding curve.

```
public fun can_graduate<CoinX, CoinY>(config: &GlobalConfig): bool {
    let curve = borrow_bonding_curve<CoinX, CoinY>(config);
    // Trading fee may still be left in the fee manager
    curve.state == BondingCurveState::Completed
    && curve.supply == 0
    && balance::value(&curve.liquidity_reserve) == 0
    && balance::value(&curve.curve_tokens) == 0
}
```

Recommendations:

Update `can_graduate()` to just check for completed.

Talus Network: Resolved with @e603368226...

Zenith: Verified.

[M-4] buy_back_burn_lp can be griefed by pool manipulation

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- iao.move#L466

Description:

`buy_back_burn_lp`will buyback CoinY using deposited CoinX revenue via `liquidity_pool::swap`, and then burn the bought CoinY. There is a burn interval check to only allow `buy_back_burn_lp` to be conducted after the interval delay.

However, it is possible for the buyback `liquidity_pool::swap` to be successful even if there are no CoinY bought from the swap. That means `buy_back_burn_lp` will be completed despite buyback did not occur.

That makes it possible for an attacker to grief `buy_back_burn_lp` by sandwiching it with swaps to manipulate the pool price such that the swap for the buyback will result in zero CoinY swapped.

This issue could also occur when there are insufficient liquidity in pool to fulfill the buyback.

Recommendations:

Add a min amount check to ensure that the `liquidity_pool::swap` has bought a certain amount of CoinY, before marking the buyback successful and reset the interval timer.

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit `cf0f47ed`.

Zenith: Verified

[M-5] revenue::total_collected is incorrectly increased in buy_back_burn_lp() and buy_back_burn_bd()

SEVERITY: Medium	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Medium

Target

- iao.move#L451
- iao.move#L525

Description:

`buy_back_burn_lp()` and `buy_back_burn_bd()` will withdraw CoinX from revenue vault to perform the buyback and burn. Any leftover CoinX that were not used for the buyback will be deposited back into the revenue vault.

However, when `iao_r1.revenue_manager.deposit()` is called again for the leftover CoinX, it will increase `revenue_manager.total_collected` and double counted the total collection amount, that was already counted when it was first deposited. This will inflate the `total_collected` value over time, resulting in an inaccuracy of the tracking.

```
/// Buy-back and burn for liquidity pool
public fun buy_back_burn_lp<CoinX, CoinY>(
    _admin_cap: &AdminCap,
    config: &mut GlobalConfig,
    iao: &mut IAO<CoinX, CoinY>,
    cetus_config: &cetus_clmm_config::GlobalConfig,
    pool: &mut Pool<CoinX, CoinY>,
    sqrt_price_limit: u128,
    by_amount_in: bool,
    mut amount: u64,
    clock: &Clock,
    ctx: &mut TxContext
) {
    ...
    //@audit leftover CoinX is re-deposited and this will increase the
    total_collected again
    iao_r1.revenue_manager.deposit<CoinX>(
        iao_r1.asset.asset_id,
        payment,
```

```
        clock
    );
```

Recommendations:

Consider adding a flag for `revenue::deposit()` that specify whether to increase the `total_collected` value. This flag can then be set to false when depositing via `buy_back_burn_lp()` and `buy_back_burn_bd()`.

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit `4cdbc050`.

Zenith: Verified.

[M-6] Missing IAO pause check in buy-back and burn functions

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- sui/iao/sources/iao.move

Description:

The `buy_back_burn_bd` and `buy_back_burn_lp` functions of the IAO module do not validate the IAO's paused state before execution. While they correctly check for global and configuration-level pauses, they omit the explicit assertion:

```
assert!(iao_r1.is_paused == false, ETradingPaused);
```

which is present in other critical functions such as `deposit`, `execute_burn`, and `collect_trading_fees_bonding_curve`.

This omission enables the execution of buy-back and burn operations even when the IAO is in a paused state, which could lead to inconsistent or unintended state transitions during maintenance or emergency scenarios.

Recommendations:

We recommend adding an explicit assertion to both `buy_back_burn_bd` and `buy_back_burn_lp` to ensure the IAO is not paused before proceeding.

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit `533fd0c5`.

Zenith: Verified

[M-7] Inconsistent use of global `AdminCap` instead of `AssetAdminCap` for IAO operations

`SEVERITY:` Medium	`IMPACT:` Medium
`STATUS:` Resolved	`LIKELIHOOD:` Medium

Target

- sui/iao/sources/iao.move

Description:

The IAO module currently uses the global `AdminCap` in functions such as `mint`, `iao_pause`, and `iao_resume` for administrative operations on an IAO.

However, in practice, it is unlikely that the same entity will hold both the global `AdminCap` and the `IAO`. Instead, the asset-specific administrative authority should be derived from the `AssetAdminCap`, which is returned upon asset registration, since it is explicitly tied to the asset's identity.

Recommendations:

We recommend modifying all functions that perform asset-specific administrative tasks (e.g., `mint`, `iao_pause`, `iao_resume`, and others where applicable) to require a reference to `AssetAdminCap` rather than the global `AdminCap`.

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit `34ced411`.

Zenith: Verified

[M-8] Public access to the `initialize` function of the `liquidity_pool` module could lead to inconsistent states

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- sui/pools/sources/liquidity_pool.move

Description:

The `initialize` function within the `liquidity_pool` module is declared with `public` visibility, allowing it to be executed by any external actor. This breaks the expected control flow where initialization is intended to be coordinated via the `register_asset` process.

Direct invocation of `initialize` without this coordination leads to a state where the liquidity pool is created but not properly linked to the corresponding IAO.

As a result, data such as the `position` built on cetus and the IAO's `graduation_status` will remain unset, leaving the module in an inconsistent state. This could hinder the correct functioning of the IAO lifecycle and disrupt the broader asset registration pipeline.

Recommendations:

We recommend ensuring the `initialize` function can only be invoked as part of the controlled register_asset flow.

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit `736f8855`.

Zenith: Verified.

[M-9] Static custom trigger metrics break level progression mechanism

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- sui/pools/sources/level.move

Description

The level management module contains a critical logical flaw in how it handles custom trigger metrics that can fundamentally break the intended progressive nature of bonding curve levels.

The issue centers on the `TriggerMetric::Custom(u64)` variant, which was likely designed to allow for specialized level progression criteria. Examining the implementation in the `get_trigger_value()` function reveals that when this variant is selected, the function simply returns the statically defined value without any dynamic context:

“ match (metric) { // Other metrics use dynamic state values... TriggerMetric::Custom(value) => value, } “

This static value is then used in the `should_trigger_level()` function where it's compared against the level's trigger threshold to determine if a level should activate. The root cause of the vulnerability is that this comparison uses a fixed value rather than one that changes based on protocol state or external conditions.

This implementation creates two problematic scenarios that undermine the bonding curve's level progression mechanism:

1. **Permanent Stagnation**: If the custom value is configured below the trigger threshold, the level will never activate regardless of other protocol conditions or the passage of time. This effectively creates "dead levels" that permanently stall bonding curve progression.

2. **Immediate Unintended Activation**: Conversely, if the custom value is set above the threshold, the level will activate immediately upon creation or as soon as the evaluation occurs, potentially triggering premature state transitions without meeting the intended economic or utilization conditions.

The highest impact scenario occurs when the protocol relies on level progressions to implement critical economic adjustments (such as fee changes, reserve ratio modifications, or price curve adjustments). In such cases, stalled levels could prevent the protocol from adapting to changing market conditions, while premature transitions could introduce economic imbalances or unexpected behavior that impacts user funds.

This issue stems from a fundamental misalignment between the static nature of the custom trigger implementation and the dynamic progression model that bonding curves are typically designed to follow.

Recommendations

The custom trigger metric should be redesigned to incorporate dynamic values rather than static ones.

Talus Network: Resolved with @142767de5bf...

Zenith: Verified

[M-10] Excessive oracle data age threshold can lead to price manipulation risk

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- sui/pools/sources/oracle.move

Description

The oracle module in the pools package implements a data freshness verification mechanism that is inadequate for protecting against market manipulation, particularly in volatile or low-liquidity markets.

The root cause of this vulnerability is located in the declaration of the `MAX_DATA_AGE_MS` constant, which is set to 300,000 milliseconds (5 minutes). This constant establishes the maximum allowed age for oracle data before it's considered stale:

“ const MAX_DATA_AGE_MS: u64 = 300000; // 5 minutes in milliseconds “

This threshold is used in critical data retrieval functions such as `get_trade_volume()`, `get_last_price()`, and `get_liquidity()`, where the system verifies data freshness by comparing the current timestamp against the last update timestamp:

“ let current_timestamp_ms = clock::timestamp_ms(clock); assert!(current_timestamp_ms - oracle_data.last_update_ms <= MAX_DATA_AGE_MS, EStaleData); “

The highest impact scenario involves price manipulation through strategic timing:

1. An attacker observes a significant price movement in the actual market (e.g., a 20% price drop)
2. The oracle data becomes increasingly stale but remains within the 5-minute window
3. The attacker executes trades against the protocol at favorable terms based on the stale oracle price
4. After extracting value, the attacker may then update the oracle or wait for it to be updated

The issue is exacerbated by the lack of any mechanisms to adjust the threshold based on

market conditions or volatility, creating a static and overly permissive window regardless of market dynamics.

Recommendations

Reduce the maximum data age threshold to a more appropriate value or

```
const MAX_DATA_AGE_MS: u64 = 300000; // 5 minutes in milliseconds
const MAX_DATA_AGE_MS: u64 = 60000;  // 1 minute in milliseconds
```

Talus Network: Resolved with @346675ae09...

Zenith: Verified.

[M-11] Cetus oracle allows arbitrary operator Input

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- sui/pools/sources/oracle.move

Description:

The `update_oracle_data_cetus` function in the oracle module allows a designated operator to submit updated reserve balances, price data, liquidity, and other metrics for a specific Cetus liquidity pool.

The oracle does not directly query data from the Cetus module, instead, it blindly accepts the values provided by the operator, relying solely on the check that the transaction sender matches the authorized operator address.

This creates a dependency on operator honesty and accuracy. A malicious operator, or one whose key has been compromised, can submit arbitrary or falsified data.

Recommendations:

We recommend retrieving data directly from the cetus module

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit `346675ae`.

Zenith: Verified

[M-12] Frontrunning in oracle updates

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- sui/pools/sources/oracle.move

Description:

The oracle module is vulnerable to frontrunning during oracle data updates, particularly when multiple operators submit updates within the same block. The functions `update_oracle_data_bonding_curve` and `update_oracle_data_cetus` overwrite the existing `OracleData` with the most recent submission, without aggregating concurrent inputs.

As a result, the final stored value is determined solely by transaction ordering. This creates an opportunity for a malicious operator to observe the mempool and strategically submit a transaction to override prior values, effectively dictating the final oracle state.

Recommendations:

We recommend implementing an aggregation mechanism within the update functions. All oracle updates submitted within a block (or defined short window) should be collected and consolidated using an aggregation method such as a median or weighted average.

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit `346675ae`.

Zenith: Verified

[M-13] Price inconsistencies and risks due to token minting

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- sui/iao/sources/iao.move

Description:

The iao module allows administrative minting of tokens using the `mint` function, bypassing the bonding curve's internal accounting mechanisms. Minted tokens are transferred directly to the caller without adjusting key bonding curve parameters such as total supply or the curve token reserve. This creates a discrepancy between the real token supply and the bonding curve's perceived state.

As a result, pricing calculations, derived from the curve's relationship between liquidity reserves, token balances, and supply, can become inaccurate.

This inconsistency may be exploited for arbitrage, triggering inflationary attacks or mispriced swaps. The vulnerability undermines the design and pricing guarantees of the bonding curve mechanism.

Recommendations:

We recommend minting tokens to the bonding curve instead of the caller.

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit `f0522760`.

Zenith: Verified

4.4 Low Risk

A total of 10 low risk findings were identified.

[L-1] Fee can be bypassed due to rounding down of fees

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- fees.move#L53-L67
- config.move#L155

Description:

`calculate_buy_fee()` and `calculate_sell_fee()` will round down the fees for buying/selling.

This can be used to bypass fee by bundling multiple dust buy/sell with zero fee due to rounding down.

The same issue applies for `calculate_protocol_fee()`.

```
/// Calculate buy fee amount
public fun calculate_buy_fee<CoinType>(
    fee_manager: &FeeManager<CoinType>,
    amount: u64
): u64 {
    minimal_fixed_point::mul_div(amount, fee_manager.trading_fee_buy_bps
    as u64, BPS as u64, false)
}

/// Calculate sell fee amount
public fun calculate_sell_fee<CoinType>(
    fee_manager: &FeeManager<CoinType>,
    amount: u64
): u64 {
    minimal_fixed_point::mul_div(amount, fee_manager.trading_fee_sell_bps
    as u64, BPS as u64, false)
```

```
}
```

```
public fun calculate_protocol_fee<CoinX>(
    global_config: &GlobalConfig,
    amount: u64
): u64 {
    let protocol_fee = get_protocol_fee<CoinX>(global_config);
    minimal_fixed_point::mul_div(amount, protocol_fee, PRECISION, false)
}
```

Recommendations:

Round up for `calculate_buy_fee()`, `calculate_sell_fee()`, `calculate_protocol_fee()`.

Talus Network: Resolved with @ff9da593ab9...

Zenith: Verified.

[L-2] Sufficient supply buffer should be provided

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- bonding_curve.move#L384

Description:

The bonding curve can be configured to be completed when the final level supply metric is triggered.

However, for a linear curve type, there is no check that to ensure there is a sufficient buffer between the initial supply and the final level's supply value.

If the initial supply == final level's supply value, there will not be curve tokens left to graduate to the liquidity pool.

The final level's supply value should be much lower than initial supply, to provide a sufficient buffer so that there are curve tokens left for graduation.

The buffer will also mitigate the risk of griefing, where the last buyer's tx fails as it exceed the actual supply when an attacker frontruns the it with a dust amount purchase. The last buyer refers to the user that buys all the remaining supply such that the curve can be marked completed and then graduated. If there is sufficient buffer, the last buyer's tx cannot be griefed by dust purchase.

Recommendations:

During level creation using supply metric, ensure that there is sufficient buffer between the initial supply and the last level.

Talus Network: Resolved with @e22e3952522...

Zenith: Verified.

[L-3] buy_back_burn_lp should set by_amount_in = true

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- iao.move#L511

Description:

`buy_back_burn_lp` is used to perform buyback of CoinY using deposited CoinX revenue, and then burn the bought CoinY. It does so using `liquidity_pool::swap()`.

However, it allows `by_amount_in` to be specified by the IAO Admin. It should instead be true as the `amount` passed in refers to the amount in. The buyback will be wrong when IAO Admin sets `buy_amount_in = false`.

```
/// Buy-back and burn for liquidity pool
public fun buy_back_burn_lp<CoinX, CoinY>(
    _admin_cap: &AdminCap,
    config: &mut GlobalConfig,
    iao: &mut IAO<CoinX, CoinY>,
    cetus_config: &cetus_clmm_config::GlobalConfig,
    pool: &mut Pool<CoinX, CoinY>,
    sqrt_price_limit: u128,
    by_amount_in: bool,
    mut amount: u64,
    clock: &Clock,
    ctx: &mut TxContext
) {
...
    liquidity_pool::swap<CoinX, CoinY>(
        cetus_config,
        pool,
        &mut payment,
        &mut coin_y,
        true,
        by_amount_in,
        amount,
        sqrt_price_limit,
```

```
        clock,
        ctx
    );
```

Recommendations:

```
    liquidity_pool::swap<CoinX, CoinY>(
        cetus_config,
        pool,
        &mut payment,
        &mut coin_y,
        true,
        by_amount_in,
     true
        amount,
        sqrt_price_limit,
        clock,
        ctx
    );
```

Talus Network: Resolved with @47e3b316a5..

Zenith: Verified.

[L-4] Unrestricted access to `graduate` function may cause inconsistent state

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- sui/pools/sources/liquidity_pool.move

Description:

The `graduate` function in the `liquidity_pool` module is publicly accessible, allowing it to be invoked independently of the intended flow managed by the `iao::graduate` function. This introduces the risk of bypassing critical state updates, such as persisting the position within the IAO object, updating the `graduation_status`, and registering the migration in the IAO registry. Improper invocation may lead to an inconsistent or partially updated system state, undermining the integrity of the IAO lifecycle and causing synchronization issues between the IAO and its registry.

Recommendations:

We recommend restricting access to the `graduate` function so that it can only be executed internally or by the `iao::graduate` function.

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit `9df2cd80`.

Zenith: Verified

[L-5] Missing validation on `supply_threshold_bps`

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: High

Target

- sui/pools/sources/configs.move

Description:

The `create_launcher_manager` function initializes a `LauncherManager` struct, including the `supply_threshold_bps` field. However, there is no validation to ensure this value is within the expected bounds (e.g., `0 < supply_threshold_bps ⩽ 10_000`).

Without such checks, it is possible to instantiate the manager with a zero or overly high threshold, which could cause incorrect premium calculations.

Recommendations:

We recommend enforcing validation on the `supply_threshold_bps` parameter in the `create_launcher_manager` function.

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit `8b8f3bbb`.

Zenith: Verified

[L-6] Missing input validation for `update_config` of the `pools` module

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- sui/pools/sources/configs.move

Description:

The `update_config` function permits the `AdminCap` owner to update configuration parameters for a given coin, including `deployment_fee` and `platform_fee`. However, these fee parameters are not properly validated, specifically, they are not enforced to remain within the valid range of `[0, PRECISION]` as in the `init_config`. As a result, a malicious or negligent owner of the `AdminCap` could configure fees that exceed the maximum allowed or the total permissible range.

Recommendations:

We recommend implementing strict validation logic within `update_config` to ensure both `deployment_fee` and `platform_fee` within the inclusive bounds of `[0, PRECISION]`.

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit `a8acc805`.

Zenith: Verified

[L-7] Missing pause check in protocol fee collection

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- sui/iao/sources/config.move

Description:

The `collect_protocol_fees` function allows an `AdminCap` holder to withdraw accumulated fees from the protocol vault for a specific coin.

However, the function does not validate whether the associated configuration is currently paused (`is_paused = true`). This omission allows protocol fees to be collected even when the config is explicitly paused, which may contradict the intended semantics of a paused state typically signaling halted operations or an emergency.

Recommendations:

We recommend introducing an explicit check at the beginning of the `collect_protocol_fees` function to assert that the configuration for the specified coin is not paused.

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit `69238c70`.

Zenith: Verified

[L-8] Missing fee validation in `init_config` of the `iao` module

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- sui/iao/sources/config.move

Description:

The `init_config` function permits the `AdminCap` owner to initialize configuration parameters for a given coin, including `deployment_fee` and `protocol_fee`. However, `protocol_fee` is not properly validated, specifically, it is not enforced to remain within the valid range of `[0, PRECISION]`, and `deployment_fee` is not checked against `MAX_DEPLOYMENT_FEE`. As a result, a malicious or negligent owner of the `AdminCap` could configure fees that exceed the maximum allowed or the total permissible range.

Recommendations:

We recommend implementing strict validation logic within `init_config` to ensure `protocol_fee` remains within the inclusive bounds of `[0, PRECISION]`. Additionally, enforce a check to ensure that `deployment_fee` does not exceed `MAX_DEPLOYMENT_FEE`.

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit `0dc035d1`.

Zenith: Verified

[L-9] Lack of operator revocation mechanism in oracle module enables persistent malicious activity

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- sui/pools/sources/oracle.move

Description:

The oracle module lacks functionality to revoke or disable oracle operator privileges once granted. Operators are assumed to act honestly and are not subject to any administrative removal or slashing mechanism. This design flaw allows malicious or compromised operators to persist indefinitely, even if they are observed submitting fraudulent or manipulative data.

In scenarios where oracle accuracy is critical, an unremovable malicious operator poses a significant risk. Their continuous ability to submit oracle updates can be exploited to corrupt market data, interfere with protocol logic, or facilitate economic attacks.

Recommendations:

We recommend implementing a mechanism to remove oracle operators.

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit 346675ae.

Zenith: Verified

[L-10] Unbounded iteration in level evaluation

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- sui/pools/sources/level.move

Description:

The `check_and_update_level` and `get_current_level_index` functions perform linear iteration over the levels vector to identify the active level and evaluate transitions.

However, while the design assumes a small number of levels under normal conditions, there is no enforced upper bound on the vector's size. If the number of levels grows significantly, either through misconfiguration or malicious input, these unbounded loops can lead to excessive gas consumption, potentially resulting in denial of service (DoS) via out-of-gas failures.

Recommendations:

We recommend enforcing a maximum number of levels that can be added to the `LevelManager`. This can be achieved via a hard cap or configurable protocol parameter.

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit `a204477b`.

Zenith: Verified

4.5 Informational

A total of 4 informational findings were identified.

[I-1] Consider locking the LP positions upon graduation

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- liquidity_pool.move#L51
- liquidity_pool.move#L105

Description:

When the bonding curve is graduated to a Cetus liquidity pool, it will bootstrap the pool liquidity using the bonding curve's reserves. A LP position NFT for the provisioned liquidity is minted and stored in the registry. As ownership of the LP position is not relinquished, there is still a possibility that it can be used to withdraw those liquidity.

However, the users and community might prefer to see the initial liquidity to be permanent locked and not withdrawable. In that case, it would better to use the Cetus LP Burn mechanism to relinquish control of the LP position,

Recommendations:

Consider using Cetus LP Burn to permanently lock the LP position for both `liquidity_pool::graduate()` and `liquidity_pool:initialize()`.

Talus Network: Resolved with @363f4b3c6a...

Zenith: Verified.

[I-2] Orphaned graduation records

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- sui/pools/sources/configs.move

Description:

When a pool is removed via `remove_pool`, any corresponding graduation record is not automatically removed or flagged. This might lead to orphaned records that no longer have a corresponding active pool.

Recommendations:

We recommend removing graduation records if present.

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit `05fdf652`.

Zenith: Verified

[I-3] Lack of event emission on pause resume

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- sui/pools/sources/configs.move

Description:

The `emergency_pause` function emits a `PauseStatusChangedEvent`, but the complementary `resume_trading` function does not emit any event upon resuming trading.

Without an event for resuming trading, off-chain monitoring and logging might lose visibility into when trading resumes.

Recommendations:

We recommend emitting a corresponding event when resuming trading.

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit `dc156faf`.

Zenith: Verified

[I-4] Inconsistent event identifier usage

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- sui/pools/sources/configs.move

Description:

The `update_config` function permits the `AdminCap` owner to update configuration parameters.

However, while in `init_config` and the pause functions (e.g., `emergency_pause`), events are emitted using the ID from the coin-specific configuration (via `object::id(config)` where config is of type `Configuration<CoinX>`), the `update_config` emits an event using the global configuration's ID (from the `GlobalConfig` parameter).

This inconsistency can lead to confusion when monitoring events and tracing which configuration was affected.

Recommendations:

We recommend ensuring consistency in the event payloads, either always emit the coin-specific configuration ID

Talus Network: This issue has been acknowledged by Talus, and a fix was implemented in commit `effb8b0c`.

Zenith: Verified