Lindy Labs Aura

Security Assessment

November 6, 2023

Prepared for:
Cristiano Teixeira
Lindy Labs

Prepared by: **Michael Colburn and Justin Jacob**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

Lindy Labs engaged Trail of Bits to review the security of its Aura smart contracts. Aura is a multicollateral protocol implemented in Cairo 1.0 that allows users to deposit a variety of collateral assets and mint a synthetic asset against them.

A team of two consultants conducted the review from June 5 to July 7, 2023, for a total of eight engineer-weeks of effort. Our testing efforts focused on identifying any accounting issues, flaws in the liquidation process, or other ways to cause funds to become trapped in the contracts or to move the contracts into unintended states. With full access to source code and documentation, we performed static testing of the smart contracts using manual processes.

Observations and Impact

Overall, the Aura codebase is quite complex but is broken down into intuitive components, so sequences of function calls are not difficult to follow. The codebase was originally written and tested against an older version of Cairo. Having access to this alternative implementation and the written specification was useful for improving our understanding of the codebase.

However, the process of porting the codebase to the most recent version of the language also introduced several issues. Further updates and changes to various components and mechanism designs (e.g., liquidations by the `Purger`) introduced significant complexity and new issues as well. The protocol has a robust access control mechanism that allows capabilities to be distributed across many discrete roles. This includes an admin role for several contracts that can exert significant control over the system, so appropriate controls to prevent its misuse (e.g., handing off this power to a decentralized autonomous organization [DAO] or, at minimum, a robust multisig in the interim) will be critical for garnering user trust.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Lindy Labs take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.

- **Continue to carefully review code that has been refactored while being ported to Cairo 1.0.** Several issues identified in the report appear to have been introduced while converting recursive calls to use loops.

- **Develop important operational guidance for monitoring and incident response, as well as the use of the admin functionality.**
- **Continue to improve test coverage and modeling of the system.** Several issues could have been caught by testing particular edge cases. Monitor the Cairo tooling ecosystem and use advanced testing techniques like fuzzing when the capabilities become available.

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	2
Medium	2
Low	4
Informational	5
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Auditing and Logging	1
Configuration	1
Data Validation	9
Undefined Behavior	2

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Jeff Braswell, Project Manager
jeff.braswell@trailofbits.com

The following engineers were associated with this project:

Michael Colburn, Consultant
michael.colburn@trailofbits.com

Justin Jacob, Consultant
justin.jacob@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
April 19, 2023	Technical onboarding call
June 1, 2023	Pre-project kickoff call
June 8, 2023	Status update meeting #1
June 15, 2023	Status update meeting #2
June 22, 2023	Status update meeting #3
June 29, 2023	Status update meeting #4
July 10, 2023	Delivery of report draft
July 10, 2023	Report readout meeting
September 1, 2023	Addition of fix review appendix
November 6, 2023	Delivery of comprehensive report with fix review

Project Goals

The engagement was scoped to provide a security assessment of the Aura smart contracts. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can there be accounting mismatches between the various components of the system?
- Is it possible to leave any trove with bad debt?
- Are the interest rates and prices updated correctly?
- Can an attacker avoid repaying flashminted yin?
- Are minimum deposits correctly enforced to prevent ERC-4626 share inflation attacks?
- When troves are opened and closed, are the correct asset amounts transferred to and from the protocol?
- Can both searcher liquidations and absorptions be prevented by a malicious actor?
- Can healthy troves be incorrectly liquidated?
- Are liquidation penalties correctly computed?
- Can the computation of shares in the `Gate` contract be manipulated?
- Are `Absorber` rewards correctly calculated and distributed, even across epochs?
- When the system is shut down, can users withdraw all their funds appropriately?
- Do redistributions correctly remove bad debt from the protocol?

Project Targets

The engagement involved a review and testing of the following target.

aura_contracts

Repository	https://github.com/lindy-labs/aura_contracts
Version	`b9cf2ae4403ee15a1fb02abd2c75ae48e91ef021` (Shrine)
	`0b46d70c58cc15437542aa65f8900d2324ec0258` (Equalizer)
	`a7fc7cf9a767cd121f62dc9b134b95c2b7fcfda0` (Gate)
	`c3de88b7f0c8a5eb9372fc859d0e047f385d3288` (Sentinel)
	`62ffdc436c59445b300b3001c5980f80cfc77f63` (Abbot)
	`696e9dd31f99d78abfdf1764e20470bf14c44f6a` (Pragma)
	`abfb496e98ba112b758a126b7fcb3705ba4b3f88` (Purger)
	`9389ec45c20eaa0ed38180d9df86307dc7484ba9` (Absorber)
Type	Cairo
Platform	Starknet

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **`Shrine`.** The `Shrine` contract is the core, non-upgradeable contract that manages the system's accounting and overall state. The `Shrine` also contains the main logic for deposits, withdrawals, and trove redistributions. We manually reviewed this contract for any issues in the accounting, such as interest accrual across multiple units of time, as well as debt and collateral being attributed correctly, both during normal operations, liquidations, and redistributions. We also reviewed the way a trove's threshold and loan-to-value (LTV) ratio are calculated, focusing on whether it is possible to bypass liquidation health checks.

- **`Purger`.** The `Purger` contract implements the liquidation mechanisms for the system. It allows a liquidator to repay an unhealthy trove's debt directly, allows the stability pool to liquidate as a group, or as a last line of defense, allows the debt to be socialized proportionally across all outstanding debt positions (redistributions). We reviewed this contract to identify any instances where a healthy trove can be liquidated or the wrong type of liquidation can be triggered. We also checked that the penalties and incentives are being calculated and applied correctly. In addition, we reviewed the redistribution logic to ensure that redistributions occur only if the stability pool cannot cover a trove's debt. Lastly, we reviewed the way collateral is transferred during a liquidation to ensure that excess collateral will not be transferred out of the system and that a trove's debt will always be backed by collateral.

- **`Flashmint`.** The `Flashmint` module is an EIP-3156–compliant module that allows the flashminting of yin from the `Shrine`. We investigated whether it is possible to avoid repaying the flashmint, whether the contract is EIP-3156–compliant, and whether the flashmint cap can be bypassed.

- **`Absorber`.** The `Absorber` contract acts as a stability pool to backstop liquidations. Users stake their synthetic asset in exchange for a share of future collateral accrued to the pool through liquidations. We looked for issues that can allow a malicious staker to gain a greater portion of the rewards than their shares would entitle them to or to bypass the unstaking cooldown and potentially earn rewards risk free. We also investigated whether shares and rewards are being correctly tracked across epochs.

- **`Pragma`.** The `Pragma` contract serves as the price feed for the system, pulling prices from the Pragma service and storing them in the `Shrine`. We reviewed this contract to check that it performs sufficient validation of the prices received from Pragma.

- **`Sentinel` and `Gate`.** Each asset supported by the system will have a corresponding `Gate` contract that provides a vault-like interface. The `Sentinel` contract acts as a router to the various `Gate`s for the other system components. While reviewing these contracts, we looked for any instances where tokens are not being handled properly, incorrect bookkeeping is being performed, or the `Sentinel` can mismanage the life cycle of a `Gate`.

- **`Equalizer` and `Allocator`.** The `Equalizer` contract is designed to mint debt surpluses to a list of users provided by the `Allocator` contract. We manually reviewed the `Equalizer`, checking that the debt surplus is correctly accounted for and minted to recipients. We also reviewed the `Allocator` to verify that allocations are correctly updated in the system.

- **`Caretaker`.** The `Caretaker` contract manages the graceful shutdown of the system. If the `Shrine` is killed, collateral can be recovered in exchange for synthetic assets, or returned directly if the system was overcollateralized. We looked for issues in this contract that could result in a user receiving an incorrect quantity of collateral back, either by accident or through malicious manipulation.

- **`Abbot`.** The `Abbot` contract is the main entry point for users to manage their troves. We manually reviewed this contract to check that it interacts properly with the `Shrine` and the `Sentinel`. We also reviewed the process by which troves are opened and closed for the correct transfer of funds and state updates.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- Commits for the contracts were finalized on an ongoing basis throughout the review as their corresponding tests were ported from Cairo 0 to Cairo 1.0. To avoid duplicating effort, components were deprioritized until these commits were received. For most contracts, changes were minimal, with the exception of `Purger`. In the initial week, we received the commit for `Shrine`. In week 2, we received the commits for `Gate`, `Equalizer`, `Allocator`, `Sentinel`, and `Abbot`. In week 3, we received commits for `Pragma`, `Absorber`, `Caretaker`, and `Purger` (though tests were still in progress).

- The `exp` function ported to Cairo 1.0 from the Balancer v2 codebase was not assessed for correctness. In addition, many of the complex mathematical operations in the system were not fully assessed to determine the maximum possible precision loss or for other numerical edge cases.

- Dynamic end-to-end testing of the system was not possible given the nascent state of the Cairo ecosystem.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The protocol heavily relies on arithmetic for tracking assets, debt, and rates over time. Precision loss is minimized through the use of Wad- and Ray-sized integers where necessary, rounding in favor of the protocol and tracking the precision loss accrual over time. Comments indicate where reverts due to overflow or underflow are possible and where the relevant check was performed upstream so the conditions cannot occur. The team also frequently models parts of the system with Desmos. We recommend further testing the arithmetic in the system to uncover numerical edge cases, especially for complex calculations such as the compounding of debt and the liquidation or absorption penalty calculations.	Satisfactory
Auditing	The contracts have appropriate event emissions in state-changing functions. Error strings indicate both the component and the reason for the error. However, the team does not have a monitoring strategy or incident response plan at this time.	Moderate
Authentication / Access Controls	The system has a fairly robust role-based access control scheme. Each contract has several discrete roles to control access to different functionalities. For example, the `Shrine` has separate roles for adding new assets, modifying system parameters, killing the `Shrine`, and many other actions. The way the access controls are laid out in each individual contract and in the `Roles` contract is very easy to understand. Additional documentation should be provided to explain which system actors have what roles and how and when these roles can be updated. However, the admin role is less intuitive. Each contract has at most one admin at a time who can	Satisfactory

	arbitrarily manage granting and revoking all the other roles. This superuser should have additional documentation explaining these powers, who controls them, and how and when they will be used.	
Complexity Management	The system is quite complex, with many moving parts, but it is divided into sensible modules. Units of time vary between components. For example, the `Shrine` works in intervals and eras, while the `Absorber` tracks epochs. Many of the contracts in the system also have their own internal accounting, which must be accurately synced among different parts of the system. Some functions also contain duplicate logic (e.g., `preview_reap`, `reap`, and `reap_internal` in `Absorber`, or `is_absorbable` and `get_absorption_penalty` in `Purger`). We recommend deduplicating this logic as much as possible and investigating ways to simplify the different internal accounting mechanisms in the system and the way time is tracked.	**Moderate**
Decentralization	For the most part, the system functions autonomously. It may periodically need intervention to update the total amount of debt in the system through the `Allocator` and `Equalizer`. As mentioned above in the access controls area, each contract with access controls has an admin superuser that can manage the other roles arbitrarily. This also includes the ability to mint synthetic assets out of thin air and seize collateral in a way that bypasses trove health checks. Access to this capability must be carefully managed. In addition, it is unclear whether the Aura protocol intends to use any sort of decentralized governance functionality.	**Moderate**
Documentation	The codebase has fairly comprehensive comment coverage and explains both what the code is doing and some of the underlying assumptions. The protocol specification does a good job of explaining the protocol at a higher level, but there are some gaps regarding more intricate protocol details. While there are diagrams about the architecture of the system as a whole, diagrams of function call sequences and common workflows (opening a trove, closing a trove, etc.) are missing.	**Satisfactory**

Front-Running Resistance	Both the `Absorber` and `Pragma` contracts have time delays that prevent immediate withdrawal of provided yin or rapid price updates, respectively. Many other operations are either privileged, such as updating protocol parameters, or user-specific, such as opening and closing a trove. However, we did find one instance of front-running (TOB-AURA-12).	**Moderate**
Low-Level Manipulation	The contracts do not use any low-level manipulation.	**Not Applicable**
Testing and Verification	The codebase has tests for most functionalities, including both unit and integration tests. However, we found issues that could have been caught with a deeper test suite. Some test cases are fairly simplistic and could be extended to reach a wider variety of scenarios. Fuzzing would be an ideal approach for this; however, the tooling landscape for Cairo is in a nascent state at this time. The Aura team has developed a list of protocol invariants, and we recommend continuously updating this list and writing end-to-end tests for the system as well.	**Moderate**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Healthy loans can be liquidated	Data Validation	High
2	block.timestamp is entirely determined by the sequencer	Undefined Behavior	Informational
3	Incorrect event emission in the Equalizer	Auditing and Logging	Low
4	Unchecked ERC-20 return values in the Absorber	Data Validation	Low
5	Incorrect loop starting index in propagate_reward_errors	Undefined Behavior	Informational
6	Redistributions may not account for accrued interest on debt	Data Validation	Medium
7	Marginal penalty may be scaled even if the threshold is equal to the absorption threshold	Data Validation	Low
8	The share conversion rate may be zero even if the Absorber is not empty	Data Validation	Medium
9	Missing safety check in the Purger's absorb function	Data Validation	Informational
10	Pair IDs are not validated to be unique	Data Validation	Informational
11	Invalid price updates still update last_price_update_timestamp	Data Validation	Low
12	Redistributions can occur even if the Shrine is killed	Data Validation	High

13	Flash fee is not taken from receiver	Configuration	**Informational**

Detailed Findings

1. Healthy loans can be liquidated	
Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-AURA-1
Target: `src/core/purger.cairo`	

Description

Due to missing validation that the loan's `threshold` is less than the LTV, an arbitrary loan can be liquidated if the threshold is between the `MAX_PENALTY_LTV` and `MAX_THRESHOLD` variables.

When loans are insolvent, liquidations via the stability pool can occur by calling the `absorb` function, which will use funds from the stability pool to cover a borrower's debt. To verify that a loan is liquidatable, the loan's LTV is checked against the `MAX_PENALTY_LTV` variable, with the rationale that the loan's threshold must be less than the LTV. This would mean that the loan's LTV is larger than the `threshold`, marking it insolvent:

```
// Performs stability pool liquidations to pay down a trove's debt in full and
transfer the
// freed collateral to the stability pool. If the stability pool does not have
sufficient yin,
// the trove's debt and collateral will be proportionally redistributed among all
troves
// containing the trove's collateral.
// - Amount of debt distributed to each collateral = (value of collateral / trove
value) * trove debt
// Reverts if the trove's LTV is not above the maximum penalty LTV
// - This also checks the trove is liquidatable because threshold must be lower than
max penalty LTV.
// Returns a tuple of an ordered array of yang addresses and an ordered array of
asset amounts
// in the decimals of each respective asset due to the caller as compensation.
#[external]
fn absorb(trove_id: u64) -> (Span<ContractAddress>, Span<u128>) {
    let shrine: IShrineDispatcher = shrine::read();
    let (trove_threshold, trove_ltv, trove_value, trove_debt) =
shrine.get_trove_info(trove_id);

    assert(trove_ltv.val > MAX_PENALTY_LTV, 'PU: Not absorbable');
```

Figure 1.1: The absorb function (`Purger.cairo#L167-L181`)

However, the threshold may not be less than the MAX_PENALTY_LTV variable. The only bound it has is the MAX_THRESHOLD variable, which is larger than MAX_PENALTY_LTV:

```
const MAX_THRESHOLD: u128 = 1000000000000000000000000000; // (ray): RAY_ONE
```

Figure 1.2: The MAX_THRESHOLD constant (Shrine.cairo#L31)

```
#[external]
fn set_threshold(yang: ContractAddress, new_threshold: Ray) {
    AccessControl::assert_has_role(ShrineRoles::SET_THRESHOLD);

    assert(new_threshold.val <= MAX_THRESHOLD, 'SH: Threshold > max');
    thresholds::write(get_valid_yang_id(yang), new_threshold);

    // Event emission
    ThresholdUpdated(yang, new_threshold);
}
```

Figure 1.3: The set_threshold function (Shrine.cairo#L449-L458)

As a result, if a loan has MAX_PENALTY_LTV < threshold < MAX_THRESHOLD and LTV < threshold, it will incorrectly be flagged for liquidation.

Exploit Scenario
Alice, a borrower, has a position with an LTV of 90% and a threshold of 95%. Eve, a malicious user, calls absorb and incorrectly liquidates Alice's position. As a result, Alice loses her funds.

Recommendations
Short term, set the MAX_THRESHOLD value to be the same as the MAX_PENALTY_LTV value.

Long term, improve unit tests to increase coverage and ensure intended behavior throughout the system.

2. block.timestamp is entirely determined by the sequencer

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-AURA-2
Target: `src/*`	

Description
The `block.timestamp` value is used throughout the codebase for validating critical operations that depend on the time. However, in Starknet, there are currently no restrictions on the return values of the `get_block_timestamp()` function. As a result, the sequencer can submit an arbitrary timestamp that may not be correct.

Exploit Scenario
The sequencer of Starknet returns an incorrect `block.timestamp`, which causes many important checks throughout the protocol to fail (e.g., oracle and interest rate updates). As a result, the Aura protocol is unusable.

Recommendations
Short term, keep parts of the codebase that depend on the timestamp to a minimum.

Long term, stay up to date with the latest Starknet documentation.

3. Incorrect event emission in the Equalizer

Severity: **Low**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-AURA-3
Target: `src/core/equalizer.cairo`	

Description

When the `equalize` function is called in the `Equalizer` contract, the function iterates over the array of addresses to be allocated to. This is implemented by popping the first element from the `recipients` and `percentages` arrays and looping until the array is empty. After all the allocations have been performed, the function emits an event that records the contents of the `recipients` array, the contents of the `percentages` array, and the total surplus that was minted. However, because the arrays are modified during execution, the values emitted in the event will be incorrect.

```
loop {
    match recipients.pop_front() {
        Option::Some(recipient) => {
            let amount: Wad = rmul_wr(surplus, *(percentages.pop_front().unwrap()));


            shrine.inject(*recipient, amount);
            minted_surplus += amount;
        },
        Option::None(_) => {
            break;
        }
    };
};

// Safety check to assert yin is less than or equal to total debt after minting
surplus
// It may not be equal due to rounding errors
let updated_total_yin: Wad = shrine.get_total_yin();
assert(updated_total_yin <= total_debt, 'EQ: Yin exceeds debt');

Equalize(recipients, percentages, minted_surplus);
```

Figure 3.1: A snippet of the `equalize` function (`equalizer.cairo#L100-L119`)

Exploit Scenario

An issue in the `Equalizer` (either due to a bug or an error when setting the `Allocator`) results in surplus being minted in the wrong proportions. This is discovered after several epochs, and the Aura team reviews event logs to trace the impact. The incorrect `Equalize` events make this effort more difficult.

Recommendations

Short term, update the `equalize` function so that it emits accurate information. This could include modifying the loop to use a counter, making a copy of the arrays before looping, or removing these arrays from the event emission.

Long term, carefully consider how the ability to log state changes could be impacted when designing loops that will process data.

4. Unchecked ERC-20 return values in the Absorber	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-AURA-4
Target: `src/core/absorber.cairo`	

Description

The `transfer_assets` function in the `Absorber` contract serves as a helper function to transfer multiple assets to a target address in one function call. Per the ERC-20 standard, calls to the `transfer` function return a Boolean indicating whether the call was successful, and developers must not assume that `false` is never returned and that the token contract will revert instead. However, the return value of the call to `transfer` in `transfer_assets` is not checked. As the tokens being transferred by this function will be third-party contracts, and given the history of differing ERC-20 implementations in the Ethereum ecosystem, validating the outcome of these function calls is strongly recommended.

```
Option::Some(asset) => {
    let asset_amt: u128 = *asset_amts.pop_front().unwrap();
    if asset_amt != 0 {
        let asset_amt: u256 = asset_amt.into();
        IERC20Dispatcher { contract_address: *asset }.transfer(to, asset_amt);
    }
},
```

Figure 4.1: A snippet from the `transfer_assets` *function (*`absorber.cairo#L893-L899`*)*

Exploit Scenario

Alice is a provider in the `Absorber`. One of the collateral tokens is briefly paused to perform an upgrade. While the token is paused, Alice attempts to claim her share of the absorbed collateral and rewards. The transfer of the paused token fails silently and Alice loses some of her share of the absorbed assets.

Recommendations

Short term, add a check to verify that the `transfer` call in `transfer_assets` was successful.

Long term, always have the code validate return values whenever possible, especially when interacting with third-party contracts.

5. Incorrect loop starting index in propagate_reward_errors

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-AURA-5
Target: `src/core/absorber.cairo`	

Description

In the `Absorber` contract, reward tokens are stored in a mapping based on their reward ID, starting at 1. Most instances where the set of reward tokens is iterated over properly start the loop at 1, but in the `propagate_reward_errors` function, the `current_rewards_id` counter is initialized to 0, contrary to the comment above the function.

```
// total number of reward tokens, starting from 1
// a reward token cannot be removed once added.
rewards_count: u8,
// mapping from a reward token address to its id for iteration
reward_id: LegacyMap::<ContractAddress, u8>,
// mapping from a reward token ID to its Reward struct:
// 1. the ERC-20 token address
// 2. the address of the vesting contract (blesser) implementing `IBlesser` for the
ERC-20 token
// 3. a boolean indicating if the blesser should be called
rewards: LegacyMap::<u8, Reward>,
```

Figure 5.1: The declaration of the rewards data structures (`absorber.cairo#L103-L112`)

```
// `current_rewards_id` should start at `1`.
fn propagate_reward_errors(rewards_count: u8, epoch: u32) {
    let mut current_rewards_id: u8 = 0;
```

Figure 5.2: A snippet of the `propagate_reward_errors` function (`absorber.cairo#L1087-L1089`)

Recommendations

Short term, update the index to start at 1.

Long term, carefully review the upper and lower bounds of loops, especially when the codebase uses both 0-indexed and 1-indexed loops.

6. Redistributions may not account for accrued interest on debt

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-AURA-6
Target: `src/core/purger.cairo`	

Description

During a stability pool liquidation, yin taken from the `Absorber` contract is used to repay a trove's debt and restore solvency. The trove's collateral is then sent back to the `Absorber` as a reward. If the `Absorber` does not have enough yin to cover all of a trove's bad debt, a redistribution occurs. During a redistribution, the debt and collateral from an insolvent trove are allocated to all the remaining troves in the system.

```
   // If absorber does not have sufficient yin balance to pay down the trove's debt
in full,
    // cap the amount to pay down to the absorber's balance (including if it is
zero).
    let purge_amt = min(max_purge_amt, absorber_yin_bal);

    // Transfer a percentage of the penalty to the caller as compensation
    let (yangs, compensations) = free(shrine, trove_id, compensation_pct, caller);

    let can_absorb_any: bool = purge_amt.is_non_zero();
    let is_fully_absorbed: bool = purge_amt == max_purge_amt;

    // Only update the absorber and emit the `Purged` event if Absorber has some yin
    // to melt the trove's debt and receive freed trove assets in return
    if can_absorb_any {
        let percentage_freed: Ray = get_percentage_freed(
            ltv_after_compensation,
            value_after_compensation,
            trove_debt,
            trove_penalty,
            purge_amt
        );

        // Melt the trove's debt using the absorber's yin directly
        shrine.melt(absorber.contract_address, trove_id, purge_amt);

        // Free collateral corresponding to the purged amount
        let (yangs, absorbed_assets_amts) = free(
            shrine, trove_id, percentage_freed, absorber.contract_address
        );
        absorber.update(yangs, absorbed_assets_amts);
        Purged(
            trove_id,
            purge_amt,
            percentage_freed,
            absorber.contract_address,
            absorber.contract_address,
            yangs,
            absorbed_assets_amts
        );
    }

    // If it is not a full absorption, perform redistribution.
    if !is_fully_absorbed {
        shrine.redistribute(trove_id);

        // Update yang prices due to an appreciation in ratio of asset to yang from
        // redistribution
        oracle::read().update_prices();
    }

    Compensate(caller, yangs, compensations);

    (yangs, compensations)
}
```

Figure 6.1: A snippet of the `absorb` function (`purger.cairo#L299-L352`)

However, the redistributed debt may not correctly accrue interest. The `redistribute` function assumes that `shrine.melt`, which uses the `charge` function to accrue interest, was called. However, if the `can_absorb_any` variable is `false`, then `shrine.melt` will never be called and the interest on the trove's debt will never be correctly accrued.

```
// Trove's debt should have been updated to the current interval via `melt` in
`Purger.purge`.
// The trove's debt is used instead of estimated debt from `get_trove_info` to
ensure that
// system has accounted for the accrued interest.
```

Figure 6.2: The comment in the `redistribute` function (`shrine.cairo#L732-L734`)

Exploit Scenario

Due to a series of liquidations, the `Absorber` has a remaining yin balance of `0`. Eve has a position that is eligible for liquidation due to a large amount of interest accrued on her debt. Eve calls the `absorb` function, and because there is no yin in the `Absorber`, the interest is never accrued. As a result, the system incorrectly has more bad debt than was redistributed.

Recommendations

Short term, modify the code to make a call to the `charge` function in the `redistribute` function.

Long term, keep track of the necessary state changes needed before and after an operation, and make sure to have the code handle edge cases where these state changes may not occur.

7. Marginal penalty may be scaled even if the threshold is equal to the absorption threshold

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-AURA-7
Target: `src/core/purger.cairo`	

Description

If a trove becomes eligible for liquidation, a penalty is applied to punish users who have insolvent positions. When computing the absorption penalty, the `get_absorption_penalty` function is used. As the comment above the function says, if the LTV exceeds the absorption threshold, the marginal penalty is scaled by the penalty scalar. However, the function will also scale the penalty if the trove's threshold is equal to the absorption threshold. This could cause a user to have to pay a higher penalty than originally intended.

```
// If LTV exceeds ABSORPTION_THRESHOLD, the marginal penalty is scaled by
`penalty_scalar`.
fn get_absorption_penalty_internal(
    threshold: Ray, ltv: Ray, ltv_after_compensation: Ray
) -> Option<Ray> {
    if ltv <= threshold {
        return Option::None(());
    }

    // It's possible for `ltv_after_compensation` to be greater than one, so we
handle this case
    // to avoid underflow. Note that this also guarantees `ltv` is lesser than one.
    if ltv_after_compensation > RAY_ONE.into() {
        return Option::Some(RayZeroable::zero());
    }

    // The `ltv_after_compensation` is used to calculate the maximum penalty that
can be charged
    // at the trove's current LTV after deducting compensation, while ensuring the
LTV is not worse off
    // after absorption.
    let max_possible_penalty = min(
        (RAY_ONE.into() - ltv_after_compensation) / ltv_after_compensation,
MAX_PENALTY.into()
    );
    if threshold >= ABSORPTION_THRESHOLD.into() {
        let s = penalty_scalar::read();
        let penalty = min(
            MIN_PENALTY.into() + s * ltv / threshold - RAY_ONE.into(),
max_possible_penalty
        );

        return Option::Some(penalty);
    }
```

Figure 7.1: The penalty calculation (`purger.cairo#L450-L478`)

Exploit Scenario

Alice, a user of the Aura protocol, opens a trove. After market conditions change, her trove becomes eligible for liquidation. However, because her trove's threshold is equal to the absorption threshold, she must pay an extra penalty beyond that intended.

Recommendations

Short term, update the conditional to the following:

```
if threshold > ABSORPTION_THRESHOLD.into()
```

Long term, improve unit test coverage to uncover edge cases and ensure intended behavior throughout the protocol.

8. The share conversion rate may be zero even if the Absorber is not empty

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-AURA-8
Target: `src/core/absorber.cairo`	

Description

After a stability pool liquidation by the `Absorber` contract, the `update` function is used to update the shares of seized collateral that stakers are entitled to. In addition, if the amount of yin in the stability pool becomes empty or goes below the `YIN_SHARE_PER_THRESHOLD` constant value, a new epoch is started. When a new epoch is started, the `epoch_share_conversion_rate` variable must be set appropriately. This variable determines the rate at which vault shares can be exchanged for underlying yin and rewards. If there is no yin in the `Absorber`, the conversion rate is 0; otherwise, the conversion rate must be 1:1.

```
if YIN_PER_SHARE_THRESHOLD > yin_per_share.val {
    let new_epoch: u32 = current_epoch + 1;
    current_epoch::write(new_epoch);


    // If new epoch's yin balance exceeds the initial minimum shares, deduct the
initial
    // minimum shares worth of yin from the yin balance so that there is at least
such amount
    // of yin that cannot be removed in the next epoch.
    if INITIAL_SHARES <= yin_balance.val {
        let epoch_share_conversion_rate: Ray = wadray::rdiv_ww(
            yin_balance - INITIAL_SHARES.into(), total_shares
        );


        epoch_share_conversion_rate::write(current_epoch,
epoch_share_conversion_rate);
        total_shares::write(yin_balance);
    } else {
        // Otherwise, set the epoch share conversion rate to 0 and total shares to
0.
        // This is to prevent an attacker from becoming a majority shareholder
        // in a new epoch when the number of shares is very small, which would
        // allow them to execute an attack similar to a first-deposit front-running
attack.
        // This would cause a negligible loss to the previous epoch's providers, but
        // partially compensates the first provider in the new epoch for the
deducted
        // minimum initial amount.
        epoch_share_conversion_rate::write(current_epoch, 0_u128.into());
        total_shares::write(0_u128.into());
    }
```

Figure 8.1: Part of the `update` function (`absorber.cairo#L596-L609`)

However, due to the incorrect conditional check, it may be possible for the epoch conversion rate to be 0, even if there is yin remaining in the `Absorber`. If the balance of yin after a stability pool liquidation is equal to the `INITIAL_SHARES` constant value, then the `epoch_share_conversion_rate` will be set to 0 while the `total_shares` variable will be set to 1,000 (the value of `INITIAL_SHARES`). This could lead to unexpected behavior and incorrect protocol accounting downstream.

Exploit Scenario

After a stability pool liquidation, the `Absorber` is left with 1,000 yin. The `update` function incorrectly sets the `epoch_share_conversion` rate to 0 after the liquidation. When Alice, a staker in the `Absorber`, tries to withdraw some of her rewards for the epoch, the protocol incorrectly calculates her reward balance as 0. As a result, Alice loses some of her rewards.

Recommendations

Short term, update the conditional to the following:

```
if INITIAL_SHARES < yin_balance.val
```

Long term, improve unit test coverage to uncover edge cases and ensure intended behavior throughout the protocol.

9. Missing safety check in the Purger's absorb function

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-AURA-9
Target: `src/core/purger.cairo`	

Description

The `Purger` has separate entry point functions for performing searcher liquidations (`liquidate`) and for triggering absorptions and redistributions (`absorb`). For the most part, both functions follow a comparable sequence of steps with one exception: after updating the trove's debt and seizing the collateral, `liquidate` includes an extra safety check that will revert if the trove's LTV somehow increased as a result of the liquidation, while `absorb` omits any such check. This should not be possible in normal operations, but consistent application of this check could block a future bug or edge case in liquidation logic from being exploitable.

```
shrine.melt(funder, trove_id, purge_amt);

// Free collateral corresponding to the purged amount
let (yangs, freed_assets_amts) = free(shrine, trove_id, percentage_freed,
recipient);

// Safety check to ensure the new LTV is lower than old LTV
let (_, updated_trove_ltv, _, _) = shrine.get_trove_info(trove_id);
assert(updated_trove_ltv <= trove_ltv, 'PU: LTV increased');
```

Figure 9.1: The extra safety check (`purger.cairo#L245-L252`)

Recommendations

Short term, add the missing check to the `absorb` function to ensure its LTV never increases.

Long term, review functionalities that perform similar operations and ensure they follow comparable steps.

10. Pair IDs are not validated to be unique	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-AURA-10
Target: `src/external/pragma.cairo`	

Description

When adding a yang to the `Pragma` contract, a `pair_id` is specified. This `pair_id` acts as a unique identifier that determines the price feed used by Pragma. For example, the `pair_id` for the ETH/USD price feed would be the `felt252` representation of the string ETH/USD. However, there is no check that the `pair_id` is unique when adding a yang. This could lead to a different yang using an incorrect price feed instead of the intended one.

```
#[external]
fn add_yang(pair_id: u256, yang: ContractAddress) {
    AccessControl::assert_has_role(PragmaRoles::ADD_YANG);
    assert(pair_id != 0, 'PGM: Invalid pair ID');
    assert(yang.is_non_zero(), 'PGM: Invalid yang address');
    assert_new_yang(yang);

    // doing a sanity check if Pragma actually offers a price feed
    // of the requested asset and if it's suitable for our needs
    let response: PricesResponse =
oracle::read().get_data_median(DataType::Spot(pair_id));
    // Pragma returns 0 decimals for an unknown pair ID
    assert(response.decimals != 0, 'PGM: Unknown pair ID');
    assert(response.decimals <= 18_u256, 'PGM: Too many decimals');

    let index: u32 = yangs_count::read();
    let settings = YangSettings { pair_id, yang };
```

Figure 10.1: The add_yang function (`pragma.cairo#L178-L193`)

Exploit Scenario

Alice, the admin of the contracts, accidentally uses the ETH/USD `pair_id` when adding wBTC as a yang in the `Pragma` contract. Despite ETH being already added, the price feed for wBTC will incorrectly use the ETH/USD feed, resulting in a completely incorrect price.

Recommendations

Short term, have the code store the used `pair_ids` in a mapping and validate that a `pair_id` has not been used when adding a new yang.

Long term, review functionalities that perform similar operations and ensure they follow comparable steps.

11. Invalid price updates still update last_price_update_timestamp	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-AURA-10
Target: `src/external/pragma.cairo`	

Description

The Aura contracts rely on price updates from the Pragma oracle to provide necessary price data to the system. Price updates take place periodically based on the `update_frequency` state variable. In addition, invalid price updates (i.e., price updates that do not meet the minimum requirements for the number of sources aggregated or that are too old) are rejected by the contract. If there is an invalid price update, no state variables are updated and an `InvalidPriceUpdate` event is emitted instead.

However, even if every price update is invalid, the `last_price_update_timestamp` variable is updated. This could potentially cause delays when computing price updates.

```
        // if we receive what we consider a valid price from the oracle, record it in
the Shrine,
        // otherwise emit an event about the update being invalid
        if is_valid_price_update(response, asset_amt_per_yang) {
            shrine::read().advance(settings.yang, price * asset_amt_per_yang);
        } else {
            InvalidPriceUpdate(
                settings.yang,
                price,
                response.last_updated_timestamp,
                response.num_sources_aggregated,
                asset_amt_per_yang
            );
        }

        idx += 1;
    };

    // record and emit the latest prices update timestamp
    last_price_update_timestamp::write(block_timestamp);
    PricesUpdated(block_timestamp, get_caller_address());
}
```

Figure 11.1: Part of the `update_prices` function (`pragma.cairo#L232-L252`)

Exploit Scenario

Due to an issue with an off-chain data provider, the Pragma oracle uses a lower number of aggregated sources than the threshold defined by the contract. As a result, every price update is considered invalid, but the `last_price_update_timestamp` variable is updated regardless. When the off-chain data provider resumes working, `update_prices` cannot be called for a larger delay than intended, leading to stale prices in the system.

Recommendations

Short term, modify the code to track the number of invalid price updates that occur when `update_prices` is called. If this number is equal to the yang count (i.e., no price update was performed), then `last_price_update_timestamp` should not be updated.

Long term, keep track of the necessary preconditions needed for state updates. Have the code validate that state updates take place only if these preconditions are met.

12. Redistributions can occur even if the Shrine is killed

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-AURA-12
Target: `src/core/absorber.cairo`	

Description

The Aura protocol has steps to shut down gracefully. First, the `Caretaker::shut` function is invoked, which kills the `Shrine` and withdraws collateral from the `Gate` contract to the `Caretaker` contract. The amount of collateral withdrawn is enough to back the total system yin at a 1:1 ratio. During a shutdown, the `Caretaker` contract will allow trove owners to burn their yin to claim back their collateral. Notably, when the `Shrine` is killed, its `forge` and `melt` functions revert.

However, there is nothing stopping a redistribution from occurring when the `Shrine` is killed as long as the `Absorber`'s yin balance is emptied. Similar to finding 6, if the `can_absorb_any` variable is `false`, then `shrine.melt` will never be called and the call to the `absorb` function will not revert. If a trove is eligible for absorption, then before the trove owner can call the `release` function to withdraw their excess collateral, an attacker can front-run them and call `absorb`, triggering a redistribution.

```
    // Only update the absorber and emit the `Purged` event if Absorber has some yin
    // to melt the trove's debt and receive freed trove assets in return
    if can_absorb_any {
        let percentage_freed: Ray = get_percentage_freed(
            ltv_after_compensation,
            value_after_compensation,
            trove_debt,
            trove_penalty,
            purge_amt
        );

        // Melt the trove's debt using the absorber's yin directly
        shrine.melt(absorber.contract_address, trove_id, purge_amt);

        // Free collateral corresponding to the purged amount
        let (yangs, absorbed_assets_amts) = free(
            shrine, trove_id, percentage_freed, absorber.contract_address
        );

        absorber.update(yangs, absorbed_assets_amts);
        Purged(
            trove_id,
            purge_amt,
            percentage_freed,
            absorber.contract_address,
            absorber.contract_address,
            yangs,
            absorbed_assets_amts
        );
    }

    // If it is not a full absorption, perform redistribution.
    if !is_fully_absorbed {
        shrine.redistribute(trove_id);

        // Update yang prices due to an appreciation in ratio of asset to yang from
        // redistribution
        oracle::read().update_prices();
    }

    Compensate(caller, yangs, compensations);

    (yangs, compensations)
}
```

Figure 12.1: Part of the `absorb` function (`purger.cairo#L309-L352`)

Exploit Scenario

The `Shrine` is killed and trove owners pull their yin out of the `Absorber` contract to reclaim their collateral from the `Caretaker`. Collateral prices fall and Bob's trove is eligible for absorption. Before Bob can call `release` to withdraw the excess collateral from a trove, Eve, a malicious user, front-runs him and calls `absorb`. This forces a redistribution, and as a result, Bob loses his excess collateral permanently.

Recommendations

Short term, make sure redistributions revert if the `Shrine` is killed.

Long term, improve unit test coverage to uncover edge cases and ensure intended behavior throughout the protocol.

13. Flash fee is not taken from receiver

Severity: **Informational**	Difficulty: **Low**
Type: Configuration	Finding ID: TOB-AURA-13
Target: `src/core/flashmint.cairo`	

Description

The `Flashmint` module allows users to mint a percentage of the total yin supply at once as long as they repay the yin at the end of the transaction. In addition, a user must also pay a fee for this flashmint, given by the FLASH_FEE constant. The `Flashmint` module is intended to be EIP-3156–compliant, and as such, it implements the appropriate functions and callbacks. Per EIP-3156, the `flash_loan` function must receive the flashloaned amount plus the flash fee from the callback; however, `shrine.eject` is called with only `amount` as a parameter. Currently, the flash fee is set to 0 so no funds will be lost, but this does break EIP-3156 compliance.

```
    shrine.inject(receiver, amount_wad);

let initiator: ContractAddress = starknet::get_caller_address();

let borrower_resp: u256 = IFlashBorrowerDispatcher {
    contract_address: receiver
}.on_flash_loan(initiator, token, amount, FLASH_FEE, call_data);

assert(borrower_resp == ON_FLASH_MINT_SUCCESS, 'FM: on_flash_loan failed');

// This function in Shrine takes care of balance validation
shrine.eject(receiver, amount_wad);
```

Figure 13.1: Part of the `flash_mint` function (`flashmint.cairo#L98-L109`)

Recommendations

Short term, make sure the code includes the FLASH_FEE when calling `shrine.eject`.

Long term, carefully monitor EIPS and ensure that the protocol meets every requirement from the specification.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories

Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories

Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

Purger

- **Update incorrect comments:**

```
 // Performs searcher liquidations that requires the caller address to supply the
amount of debt to repay
// and the recipient address to send the freed collateral to.
// Reverts if:
// - the trove is not liquidatable (i.e. LTV > threshold).
// - the repayment amount exceeds the maximum amount as determined by the close
factor.
// - if the trove's LTV is worse off than before the liquidation (should not be
possible, but as a precaution)
// Returns a tuple of an ordered array of yang addresses and an ordered array of
freed collateral amounts
// in the decimals of each respective asset due to the recipient for performing the
liquidation.
```

Figure C.1: The comments above the `liquidate` function (`purger.cairo#L214-L220`)

```
// Handling the case where `ltv > 1` to avoid underflow
if ltv >= RAY_ONE.into() {
    return Option::Some(RayZeroable::zero());
}
```

Figure C.2: The comment in the `get_liquidation_penalty_internal` function (`purger.cairo#L433-L436`)

Shrine

- **Update incorrect comments:**

```
 // Returns the average multiplier over the specified time period, including
`end_interval` but NOT including `start_interval`
// - If `start_interval` is the same as `end_interval`, return the multiplier value
at that interval.
// - If `start_interval` is different from `end_interval`, return the average.
// Return value is a tuple so that function can be modified as an external view for
testing
fn get_avg_multiplier(start_interval: u64, end_interval: u64) -> Ray {
```

Figure C.3: The comment above the `get_avg_multiplier` function (`shrine.cairo#L1317-L1321`)

Absorber

- **Remove unused parameters.** Here, `total_shares` is never used:

```
    let total_shares: Wad = total_shares::read();

    // NOTE: both `get_absorbed_assets_for_provider_internal` and
`get_provider_accumulated_rewards`
    // contain early returns of empty arrays if `provision.shares` is zero.

    // Loop over absorbed assets and transfer
    let (absorbed_assets, absorbed_asset_amts) =
get_absorbed_assets_for_provider_internal(
        provider, provision, provider_last_absorption_id, current_absorption_id
    );
    transfer_assets(provider, absorbed_assets, absorbed_asset_amts);

    // Loop over accumulated rewards, transfer and update provider's rewards
cumulative
    let (reward_assets, reward_asset_amts) = get_provider_accumulated_rewards(
        provider, provision, current_epoch, rewards_count
    );
    transfer_assets(provider, reward_assets, reward_asset_amts);

    // NOTE: it is very important that this function is called, even for a new
provider.
    // If a new provider's cumulative rewards are not updated to the current epoch,
    // then they will be zero, and the next time `reap_internal` is called, the
provider
    // will receive all of the cumulative rewards for the current epoch, when they
    // should only receive the rewards for the current epoch since the last time
    // `reap_internal` was called.
    //
    // NOTE: We cannot rely on the array of reward addresses returned by
    // `get_provider_accumulated_rewards` because it returns an empty array when
    // `provision.shares` is zero. This would result in a bug where the reward
cumulatives
    // for new providers are not updated to the latest epoch's values and start at
0. This
    // wrongly entitles a new provider to receive rewards from epoch 0 up to the
    // latest epoch's values, which would eventually result in an underflow when
    // transferring rewards during a `reap_internal` call.
    update_provider_cumulative_rewards(provider, current_epoch, rewards_count);

    Reap(provider, absorbed_assets, absorbed_asset_amts, reward_assets,
reward_asset_amts);
}
```

Figure C.4: Part of the `reap_internal` function (`absorber.cairo#L792-L826`)

D. Incident Response Plan Recommendations

This section provides recommendations on formulating an incident response plan.

- **Identify the parties (either specific people or roles) responsible for implementing the mitigations when an issue occurs (e.g., deploying smart contracts, pausing contracts, upgrading the front end, etc.).**
- **Document internal processes for addressing situations in which a deployed remedy does not work or introduces a new bug.**
    - Consider documenting a plan of action for handling failed remediations.
- **Clearly describe the intended contract deployment process.**
- **Outline the circumstances under which the Aura protocol will compensate users affected by an issue (if any).**
    - Issues that warrant compensation could include an individual or aggregate loss or a loss resulting from user error, a contract flaw, or a third-party contract flaw.
- **Document how the team plans to stay up to date on new issues that could affect the system; awareness of such issues will inform future development work and help the team secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.**
    - Identify sources of vulnerability news for each language and component used in the system, and subscribe to updates from each source. Consider creating a private Discord channel in which a bot will post the latest vulnerability news; this will provide the team with a way to track all updates in one place. Lastly, consider assigning certain team members to track news about vulnerabilities in specific components of the system.
- **Determine when the team will seek assistance from external parties (e.g., auditors, affected users, other protocol developers, etc.) and how it will onboard them.**
    - Effective remediation of certain issues may require collaboration with external parties.
- **Define contract behavior that would be considered abnormal by off-chain monitoring solutions.**

It is best practice to perform periodic dry runs of scenarios outlined in the incident response plan to find omissions and opportunities for improvement and to develop “muscle memory.” Additionally, document the frequency with which the team should perform dry runs of various scenarios, and perform dry runs of more likely scenarios more regularly. Create a template to be filled out with descriptions of any necessary improvements after each dry run.

E. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On September 1, 2023, Trail of Bits reviewed the fixes and mitigations implemented by the Lindy Labs team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In addition to the fixes for issues identified in this report, we also reviewed fixes for two issues that were identified by the Lindy Labs team concurrent to our review. The first issue concerned a situation where the system could accumulate inaccessible collateral and bad debt as a result of the initial yang amount not being attributed to a trove. This was fixed in PR #348, which implements changes to prevent this initial sharing from accruing debt or collateral and includes additional logic for handling redistributions when a yang is being used only by the trove that is being redistributed. The second issue concerns an edge case where a trove that could be partially absorbed would instead be redistributed, which would be an excessive penalty. This was fixed in PR #375 and PR #396.

In summary, of the 13 issues described in this report, Lindy Labs has resolved nine issues, has partially resolved one issue, and has not resolved the remaining three issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Healthy loans can be liquidated	Resolved
2	block.timestamp is entirely determined by the sequencer	Unresolved
3	Incorrect event emission in the Equalizer	Resolved
4	Unchecked ERC-20 return values in the Absorber	Unresolved
5	Incorrect loop starting index in propagate_reward_errors	Resolved
6	Redistributions may not account for accrued interest on debt	Resolved

7	Marginal penalty may be scaled even if the threshold is equal to the absorption threshold	Resolved
8	The share conversion rate may be zero even if the Absorber is not empty	Resolved
9	Missing safety check in the Purger’s absorb function	Resolved
10	Pair IDs are not validated to be unique	Resolved
11	Invalid price updates still update last_price_update_timestamp	Partially Resolved
12	Redistributions can occur even if the Shrine is killed	Resolved
13	Flash fee is not taken from receiver	Unresolved

Detailed Fix Review Results

TOB-AURA-1: Healthy loans can be liquidated
Resolved in PR #335. The conditions for triggering an absorption were updated as part of a larger refactor of the liquidation penalty calculations. Absorptions can no longer be triggered under any circumstances unless the trove's LTV exceeds the trove's threshold.

TOB-AURA-2: block.timestamp is entirely determined by the sequencer
Unresolved. The Lindy Labs team acknowledges the issue.

TOB-AURA-3: Incorrect event emission in the Equalizer
Resolved in PR #343 and PR #412. The loop now iterates over copies of the `recipients` and `percentages` arrays to allow the original arrays to be emitted in the event.

TOB-AURA-4: Unchecked ERC-20 return values in the Absorber
Unresolved. The Lindy Labs team is aware of this issue but does not plan to address it via code changes. The team is planning to develop a playbook for evaluating potential tokens to onboard and will consider this issue as part of that evaluation.

TOB-AURA-5: Incorrect loop starting index in propagate_reward_errors
Resolved in PR #327. The loop was updated to start from 1 instead of 0. Additionally, in PR #370 and PR #382, the loop start and end indices were updated across the entire codebase to be treated in a consistent way.

TOB-AURA-6: Redistributions may not account for accrued interest on debt
Resolved in PR #351. The call to `shrine.melt` is now performed before the `if can_absorb_any` conditional block so that interest is properly accrued.

TOB-AURA-7: Marginal penalty may be scaled even if the threshold is equal to the absorption threshold
Resolved in PR #352. The comparison to `ABSORPTION_THRESHOLD` was updated to use the greater than symbol [>] instead of the greater than or equal to symbol [>=].

TOB-AURA-8: The share conversion rate may be zero even if the Absorber is not empty
Resolved in PR #358. The comparison of `INITIAL_SHARES` to the new epoch's yin balance was updated to use the less than symbol [<] instead of the less than or equal to symbol [<=].

TOB-AURA-9: Missing safety check in the Purger's absorb function
Resolved in PR #361. The missing safety check was added to `purger.absorb`.

TOB-AURA-10: Pair IDs are not validated to be unique
Resolved in PR #362. The `assert_new_yang` function called by `add_yang` now also checks that the pair ID has not already been associated with another yang.

TOB-AURA-11: Invalid price updates still update last_price_update_timestamp
Partially resolved in PR #363. The team acknowledges this issue but does not plan to address it fully as this variable is also used to manage how frequently the Yagi keepers can attempt to trigger price updates. To help clarify this, the team has renamed the variable to `last_update_prices_call_timestamp` and modified the `update_prices` function to emit a `PricesUpdated` event only if at least one price had a valid update. The team is also exploring integrating a fallback oracle to minimize the impact of a Pragma outage.

TOB-AURA-12: Redistributions can occur even if the Shrine is killed
Resolved in PR #351. The call to `shrine.melt` is now performed before the `if can_absorb_any` conditional block, which will cause the transaction to revert if the `Shrine` is not in a live state.

TOB-AURA-13: Flash fee is not taken from receiver
Unresolved. The team acknowledges this issue but designed the protocol with zero-fee flash loans in mind.

F. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	Description
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.