HashEye

// SMART CONTRACT SECURITY AUDIT

Opus

Security assessment by HashEye · prepared for Lindy Labs

HASHEYE AUDITED

PROJECT	Opus
PERFORMED BY	HashEye Security Review Team - Sergey, Brian
RESEARCH	Sergey, Brian
CLIENT	Lindy Labs
CATEGORY	Blockchain
PUBLISHED	December 1, 2023
REPORT ID	research-opus-2023-12-01-1ku17t

This report was produced by HashEye's security review team under the HashEye review process, combining source review, deployment-state checks, and senior manual verification. Full report detail is available at hasheye.io/audits/research-opus-2023-12-01-1ku17t.

Opus Security Assessment February 29, 2024 Prepared for: Opus Team Lindy Labs Prepared by: Michael Colburn and Justin Jacob

About HashEye Founded in 2012 and headquartered in New York, HashEye 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/hasheye-io/publications , with links to papers, presentations, public audit reports, and podcast appearances. In recent years, HashEye 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. HashEye 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 hasheye on Twitter and explore our public repositories at https://github.com/hasheye-io . To engage us directly, visit our "Contact" page at https://www.hasheye.io/contact , or email us at info@hasheye.io . HashEye, Inc. 497 Carroll St., Space 71, Seventh Floor Brooklyn, NY 11215 https://www.hasheye.io info@hasheye.io HashEye 1 Opus Contracts Security Assessment PUBLIC

Notices and Remarks Copyright and Distribution © 2024 by HashEye, Inc. All rights reserved. HashEye hereby asserts its right to be identified as the creator of this report in the United Kingdom. This report is considered by HashEye to be public information; it is licensed to Lindy Labs under the terms of the project statement of work and has been made public at Lindy Labs's request. Material within this report may not be reproduced or distributed in part or in whole without the express written permission of HashEye. The sole canonical source for HashEye publications is the HashEye Publications page . Reports accessed through any source other than that page may have been modified and should not be considered authentic. Test Coverage Disclaimer All activities undertaken by HashEye 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. HashEye 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. HashEye 2 Opus Contracts Security Assessment PUBLIC

Table of Contents About HashEye 1 Notices and Remarks 2 Table of Contents 3 Project Summary 4 Executive Summary 5 Project Goals 7 Project Targets 8 Project Coverage 9 Codebase Maturity Evaluation 12 Summary of Findings 15 Detailed Findings 16 1. Redistributed debt does not accrue interest until next trove action 16 2. Incorrect starting index in the bestow function 18 3. MAX_YANG_RATE is set lower than intended 20 4. LOWER_UPDATE_FREQUENCY_BOUND is set much lower than Starknet block time 21 5. The absorb function can still be called even if the Absorber is killed 23 6. The get_shrine_health function does not account for interest or recovery mode threshold 25 7. Yin cannot be pulled out of the Absorber if the Shrine is killed 27 8. Exceptionally redistributed yangs are not included in compensation for absorptions 29 A. Vulnerability Categories 32 B. Code Maturity Categories 34 C. Opus System Invariants 36 D. Fix Review Results 38 Detailed Fix Review Results 39 E. Fix Review Status Categories 40 HashEye 3 Opus Contracts Security Assessment PUBLIC

Project Summary Contact Information The following project manager was associated with this project: Jeff Braswell , Project Manager jeff .braswell@hasheye.io The following engineering director was associated with this project: Josselin Feist , Engineering Director, Blockchain josselin.feist@hasheye.io The following consultants were associated with this project: Michael Colburn , Consultant Justin Jacob , Consultant michael.colburn@hasheye.io justin.jacob@hasheye.io Project Timeline The significant events and milestones of the project are listed below. Date Event November 30, 2023 Pre-project kickoff call December 12, 2023 Status update meeting #1 December 20, 2023 Status update meeting #2 December 27, 2023 Status update meeting #2 January 2, 2024 Delivery

of report draft January 2, 2024 Report readout meeting January 12, 2024 Delivery of comprehensive report February 29, 2024 Delivery of updated report and fix review appendix HashEye 4 Opus Contracts Security Assessment PUBLIC

Executive Summary Engagement Overview Lindy Labs engaged HashEye to review the security of its Opus smart contracts. Opus is a multicollateral protocol implemented in Cairo 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 December 4 to December 29, 2023, for a total of eight engineer-weeks of effort. Our testing eff orts focused on changes made to the codebase since our previous review in June of 2023, particularly the new Controller and Seer modules, a new library for SignedWad numbers, a new recovery mode state, the ability to delist collateral assets, explicit accounting for budget surpluses and defi cits for future protocol features, and updates to the liquidation and redistribution processes. With full access to the source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes. Observations and Impact Overall, the Opus codebase is quite complex but is broken down into intuitive components, so sequences of function calls are not difficult to follow. The codebase is well documented, with plenty of inline comments that explain the design decisions and intentions of the developer. However, updates and changes to various components and mechanism designs (e.g., the inability to accrue interest over every trove in the system) introduced significant complexity, new issues, and some discrepancies between the codebase and the specification in the provided technical documentation. Many of the modules have different data structures that store accounting information across uniquely different time periods. Verifying system invariants about these data structures (e.g., if the yin per share never goes below the threshold, the epoch will never increase) are crucial for maintaining system security. The system would benefi t from stateful fuzz testing around these particular areas. In particular, the various internal accounting mechanisms for both ordinary and exceptional redistributions are quite complex and, as a result, are difficult to exhaustively review manually, so they would greatly benefi t from additional further review via automated testing. However, given the nascent state of tooling in the Cairo ecosystem, implementing such testing would be rather challenging. Recommendations Based on the codebase maturity evaluation and findings identified during the security review, HashEye 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. HashEye 5 Opus Contracts Security Assessment PUBLIC

• Continue to develop operational guidance for criteria to determine when to delist collateral types. • Add further tests that encompass a variety of edge cases. In addition to having both positive and negative testing, try to test for a variety of different scenarios that encompass various collateral types, time periods, and liquidations/redistributions. Make sure that system invariants hold after every possible trove action or series of actions. Finding Severities and Categories The following tables provide the number of findings by severity and category. EXPOSURE ANALYSIS Severity Count High 1 Medium 2 Low 2 Informational 3 Undetermined 0 CATEGORY BREAKDOWN Category Count Data Validation 6 Timing 2 HashEye 6 Opus Contracts Security Assessment PUBLIC

Project Goals The engagement was scoped to provide a security assessment of the Opus Cairo smart contracts. Specifically, we sought to answer the following non-exhaustive list of questions: • Can an attacker avoid accruing interest on their debt? • Are the access controls implemented correctly? • Are there any gaps or edge cases in both ordinary and exceptional redistributions? • Are liquidation penalties applied correctly? • Does the Caretaker contract gracefully shut down the system? • Does the Shrine contract gracefully transition in and out of recovery mode, and does the bookkeeping take this into account consistently? • Is it possible to bypass or abuse the yang suspension mechanism? • Does the budget properly track system surpluses and defi cits? • Is it possible to prevent unhealthy positions from being liquidated? • How are rewards handled and distributed based on the current health of the Shrine contract? • Can an attacker manipulate any share or asset calculations? HashEye 7 Opus Contracts Security Assessment PUBLIC

Project Targets The engagement involved a review and testing of the following target. Opus Contracts Repository https://github.com/lindy-labs/opus_contracts Version b4c08d2e3c59b22e87847dd6a68721a1b86ba156 Type Cairo Platform Starknet HashEye 8 Opus Contracts Security Assessment PUBLIC

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. It 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. We checked that debt and collateral are attributed correctly during normal operations, liquidations, and redistributions, as well as in the part of

the budget that tracks debt surpluses and defi cits. 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. • Controller : The Controller contract algorithmically updates the interest rate multiplier in the Shrine contract to help minimize how far the yin price is able to drift from the target. We reviewed this contract to check that it implements the expected formula provided in the system's documentation and that it returns values within the range expected by the Shrine . • 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 (redistribution). We reviewed this contract to identify any instances in which a healthy trove can be liquidated or the wrong type of liquidation can be triggered. We also checked that the penalties and incentives are 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. • 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 correctly tracked across epochs. • 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 HashEye 9 Opus Contracts Security Assessment PUBLIC

avoid repaying a flashmint, whether the contract is EIP-3156-compliant, and whether the flashmint cap can be bypassed. We also reviewed the process by which the Shrine 's debt ceiling is updated to account for any budget defi cit or surplus during a flashmint operation. • 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. Lastly, we verified that budget defi cits are correctly decremented when yin is burned. • 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 contracts for the other system components. While reviewing these contracts, we looked for any instances in which tokens are not handled properly (including during the yang suspension process), incorrect bookkeeping is performed, access controls could be bypassed, or the Sentinel could mismanage the life cycle of a Gate . • Seer and Pragma : The Seer contract serves as the price feed for the system, pulling prices from the Pragma service and storing them in the Shrine . In addition, the Seer uses keeper services to automatically update price information. We reviewed this contract to check that it performs sufficient validation of the prices received from Pragma and that keeper credits could not be maliciously consumed. • 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. • 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 is overcollateralized. We looked for issues in this contract that could cause a user to receive an incorrect quantity of collateral back, either by accident or through malicious manipulation. • utils : The system includes various utility contracts in the utils folder. These contracts provide the logic for access control checks, reentrancy guards, and exponentiation and signed and unsigned arithmetic operations on wad- and ray-sized fixed point numbers. We reviewed these libraries to better understand how they are used throughout the codebase but did not exhaustively test them in isolation. HashEye 10 Opus Contracts Security Assessment PUBLIC

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: • Dynamic end-to-end testing of the system was not possible given the nascent state of the Cairo ecosystem. • Redistributions (both ordinary and exceptional) involve many different data structures to be updated based on a variety of possible scenarios. While we did manually review the redistribution logic, further testing and review, focusing particularly on edge cases based on different trove collateral compositions, is recommended. • We only briefl y reviewed the signed WadRay implementation, spending most of our

focus on the modules that encompass the protocol’s main logic. HashEye 11 Opus Contracts Security Assessment PUBLIC

Codebase Maturity Evaluation HashEye 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. Defi ciencies 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 is performed upstream so that 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, the liquidation or absorption penalty calculations, and the accounting done for redistributions. We also recommend further testing the mathematical libraries such as SignedWad , focusing on boundary case values (e.g., 0, 1, -1, felt252::MAX, etc.). Satisfactory Auditing The state-changing functions in the contracts emit appropriate events. Error strings indicate both the component and the reason for the error. In addition, the technical documentation provides a detailed incident response plan outlining the steps to be taken in the event of an emergency. The Opus team also uses Hypernative to monitor the contracts; however, the scope and extent of this monitoring is unknown at this time. Satisfactory 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 Satisfactory HashEye 12 Opus Contracts Security Assessment PUBLIC

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 arbitrarily manage the granting and revoking of all the other roles. The system documentation includes a note that the protocol requires trusting the admin to not abuse these capabilities. 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 with different parts of the system. Several of the ERC-4626 vault-style modules, like the Shrine , Gate , and Absorber , have unique processes to deal with the initial 1,000 wei deposit. These procedures may introduce unexpected edge cases and should be tested very thoroughly. Lastly, the redistribution logic is quite complex and could be split up into smaller building blocks. 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 section, each contract with access controls has an admin superuser that can arbitrarily manage the other roles. This also includes the ability to mint synthetic assets out of thin air and seize collateral in a way that bypasses trove health checks (normally used during flashmints and liquidations, respectively). Access to this capability must be carefully managed. In addition, the Opus team plans to migrate the contracts to a DAO; however, the timeline for doing so is still undetermined. Moderate Documentation The codebase has fairly comprehensive comment coverage and explains both the code’s workings and Satisfactory HashEye 13 Opus Contracts Security Assessment PUBLIC

some of the underlying assumptions. The documentation includes diagrams about the architecture of the system as a whole and those that detail the flow of funds and external calls through some operations. Given the complexity of some of the liquidation flows, more diagrams of the internal call sequences may be useful for these as well. In addition, the protocol would benefi t from more user-centric documentation. Most of the existing documentation is quite technical and appears to be targeted toward developers. 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 Opus 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 Transaction Ordering 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. We did not note any transaction ordering issues, but more complex scenarios may be possible. Further Investigation Required HashEye 14 Opus Contracts Security Assessment PUBLIC

Summary of Findings The table below summarizes the findings of the review, including type and severity details. ID Title Type Severity 1 Redistributed debt does not accrue interest until next trove action Timing Low 2 Incorrect starting index in the bestow function Data Validation Informational 3 MAX_YANG_RATE is set lower than intended Data Validation Informational 4 LOWER_UPDATE_FREQUENCY_BOUND is set much lower than Starknet block time Data Validation Informational 5 The absorb function can still be called even if the Absorber is killed Data Validation Medium 6 The get_shrine_health function does not account for interest or recovery mode threshold Data Validation Low 7 Yin cannot be pulled out of the Absorber if the Shrine is killed Data Validation High 8 Exceptionally redistributed yangs are not included in compensation for absorption s Timing Medium HashEye 15 Opus Contracts Security Assessment PUBLIC

Detailed Findings 1. Redistributed debt does not accrue interest until next trove action Severity: Low Difficulty: Low Type: Timing Finding ID: TOB-OPUS-1 Target: src/core/shrine.cairo Description When a trove becomes eligible for liquidation and the Absorber does not have enough yin to cover the trove’s debt, a redistribution occurs. A redistribution will take all of the debt and collateral from an unhealthy trove and evenly distribute it among the remaining troves in the system. This has the eff ect of removing the trove’s bad debt at the expense of splitting it across other trove owners. When a trove owner takes an action, such as depositing or withdrawing from the system, the charge function is called. This function accrues interest on the trove’s debt and pulls redistributed debt into the trove: let compounded_trove_debt: Wad = self .compound(trove_id, trove, current_interval); // Pull undistributed debt and update state let trove_yang_balances: Span <YangBalance> = self .get_trove_deposits(trove_id); let (updated_trove_yang_balances, compounded_trove_debt_with_redistributed_debt) = self .pull_redistributed_debt_and_yangs(trove_id, trove_yang_balances, compounded_trove_debt); Figure 1.1: A snippet of the charge function in shrine.cairo#L1412-L1417 However, self.compound is called before the redistributed debt has been pulled into the trove. As a result, the redistributed debt is not charged any interest for the period from the last redistribution to the next trove action . Exploit Scenario A large trove becomes insolvent, and its debt becomes redistributed across all of the troves in the system. Eve waits a significant amount of time before attempting to withdraw some of her assets. If the interest were accrued on the redistributed debt in her trove, her position would have been insolvent after withdrawal, and she would not have been able to HashEye 16 Opus Contracts Security Assessment PUBLIC

withdraw her assets. However, because the interest was not accrued, she can withdraw her funds successfully. Recommendations Short term, implement a function that calculates the average redistributed debt during a given time frame, and have the system charge interest on said debt. This will ensure that the system accounts for interest on redistributed debt. Long term, improve the unit test coverage to uncover edge cases and ensure that the system behaves as intended. In addition, have the system validate all interest accruals and ensure the principal before compounding is accurate. HashEye 17 Opus Contracts Security Assessment PUBLIC

2. Incorrect starting index in the bestow function Severity: Informational Difficulty: Low Type: Data Validation Finding ID: TOB-OPUS-2 Target: src/core/absorber.cairo Description The Absorber module acts as a stability pool to help protect trove solvency. Users can stake their yin in the Absorber to earn rewards in the form of absorbed assets (i.e., user collateral received from a liquidation) or excess yin from debt surpluses. The bestow function is used to calculate and allocate these rewards; however, the current_rewards_id starts at 0 instead of LOOP_START . Currently, this is not an issue because the rewards mapping starts at 1 , and, as a result, a current_rewards_id of 0 will be skipped. However, if the reward calculation logic were ever refactored or redesigned, the incorrect index could become problematic. fn bestow (ref self : ContractState) { // Defer rewards until at least one provider deposits let total_shares: Wad = self .total_shares.read(); if !is_operational_helper(total_shares) { return ; } // Trigger issuance of active rewards let total_recipient_shares: Wad = total_shares - INITIAL_SHARES.into(); let epoch: u32 = self .current_epoch.read(); let mut blessed_assets: Array <AssetBalance> = ArrayTrait::new(); let mut current_rewards_id: u8 = 0 ; let loop_end: u8 = self .rewards_count.read() + REWARDS_LOOP_START; loop { if current_rewards_id == loop_end { break ; } let reward: Reward = self .rewards.read(current_rewards_id); if !reward.is_active { current_rewards_id += 1 ; continue ; } [...] Figure 2.1: Part of the bestow function in absorber.cairo#L912-L936 HashEye 18 Opus Contracts Security Assessment PUBLIC

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. HashEye 19 Opus Contracts Security Assessment PUBLIC

3. MAX_YANG_RATE is set lower than intended Severity: Informational Difficulty: Low Type: Data Validation Finding ID: TOB-OPUS-3 Target: src/core/shrine.cairo Description The MAX_YANG_RATE constant represents the maximum interest rate that can be set for any yang used in the system. This value is stored as a fixed-point number with 27 decimals (i.e., a ray unit). The comment above the declaration states that this value should be 1 ray, but the actual declaration sets it to 0.1 ray (i.e., 1e26). As a result, the system does not support base interest rates above 10%. Note that the USE_PREV_BASE_RATE constant, which the code comments defi ne relative to MAX_YANG_RATE , is set to the correct expected value (i.e., 1e27 + 1). // Maximum interest rate a yang can have (ray): RAY_ONE const MAX_YANG_RATE: u128 = 100000000000000000000000000 ; // Flag for setting the yang's new base rate to its previous base rate in `update_rates` // (ray): MAX_YANG_RATE + 1 const USE_PREV_BASE_RATE: u128 = 100000000000000000000000001 ; Figure 3.1: The declaration of the MAX_YANG_RATE and USE_PREV_BASE_RATE constants in shrine.cairo#L63-68 Recommendations Short term, correct the MAX_YANG_RATE constant to be initialized to the correct value. Long term, improve the unit test coverage to ensure that the system behaves as intended. HashEye 20 Opus Contracts Security Assessment PUBLIC

4. LOWER_UPDATE_FREQUENCY_BOUND is set much lower than Starknet block time Severity: Informational Difficulty: High Type: Data Validation Finding ID: TOB-OPUS-4 Target: src/core/seer.cairo Description To monitor and determine the frequency with which price feeds for a given yang in the Shrine can be updated per block (using the update_price function), the Seer module has an update_frequency state variable. Currently, the lower bound on the update frequency is set to 15 seconds, and the surrounding comments state that the goal is that the price can be updated at least every block. However, the current block time is typically much higher than that (around 2–3 minutes), and the Starknet documentation states that the average block time is 3 minutes. const LOWER_UPDATE_FREQUENCY_BOUND: u64 = 15 ; // seconds (approx. Starknet block prod goal) Figure 4.1: The declaration of the LOWER_FREQUENCY_UPDATE_BOUND constant in seer.cairo#L29 fn set_update_frequency (ref self : ContractState , new_frequency: u64) { self .access_control.assert_has_role(seer_roles::SET_UPDATE_FREQUENCY); assert(LOWER_UPDATE_FREQUENCY_BOUND ≤ new_frequency && new_frequency ≤ UPPER_UPDATE_FREQUENCY_BOUND, ' SEER: Frequency out of bounds '); [...] Figure 4.2: The frequency update bounds check in the set_update_frequency function in seer.cairo#L156-L161 Exploit Scenario Bob, a developer at Lindy Labs, sets the update_frequency variable to the lowest possible value, 15 seconds, with the purpose of letting the keeper update the price every single block. However, this update frequency allows the keeper to call update_frequency many times per block, consuming keeper credits. HashEye 21 Opus Contracts Security Assessment PUBLIC

Recommendations Short term, set the LOWER_UPDATE_FREQUENCY_BOUND variable to 180 seconds, which is in line with the expected Starknet block production time. Long term, keep up to date with the Starknet documentation, and ensure that constants and system parameters are in accordance with it. HashEye 22 Opus Contracts Security Assessment PUBLIC

5. The absorb function can still be called even if the Absorber is killed Severity: Medium Difficulty: High Type: Data Validation Finding ID: TOB-OPUS-5 Target: src/core/purger.cairo Description The Absorber module is responsible for being a backstop against unhealthy troves. Similar to the Shrine , the Absorber module can be killed by the admin of the Opus system. When it is killed, providing yin to the Absorber becomes impossible; the only possible actions stakers can take are reaping collateral rewards, requesting to remove their provided yin, or removing their yin from the Absorber entirely. However, liquidations using the stability pool are still possible even when the Absorber is killed, as the absorb function does not check that the Absorber is live: fn absorb (ref self : ContractState , trove_id: u64) → Span <AssetBalance> { [...] // If the absorber is operational, cap the purge amount to the absorber's balance // (including if it is zero). let purge_amt = if absorber.is_operational() { min(max_purge_amt, shrine.get_yin(absorber.contract_address)) } else { WadZeroable::zero() }; [...] Figure 5.1: Part of the absorb function in purger.cairo#L265-L291 #[inline(always)] fn is_operational_helper (total_shares : Wad) → bool { total_shares ≥ MINIMUM_SHARES.into() } Figure 5.2: The is_operational_helper function in absorber.cairo#L1102-L1106 Exploit Scenario Due to a bug in the Absorber contract, the admin of the Opus system kills the Absorber , with the intention to migrate the yin and rewards to a newly deployed Absorber contract that patches the issue. However, Eve calls the absorb function, which triggers a redistribution on an unhealthy trove’s debt and uses up the Absorber ’s yin. In addition, this also updates the epoch, allowing yin providers to earn extra rewards than intended. HashEye 23 Opus Contracts Security Assessment PUBLIC

Recommendations Short term, enforce that the absorb function cannot be called unless the Absorber is live. Long term, improve the unit test coverage to uncover edge cases and ensure that the system behaves as intended. Ensure that all system features are appropriately disabled during component

shutdowns, and document invariants that should hold when the system components are not live. HashEye 24 Opus Contracts Security Assessment PUBLIC

6. The get_shrine_health function does not account for interest or recovery mode threshold Severity: Low Difficulty: Low Type: Data Validation Finding ID: TOB-OPUS-6 Target: src/core/shrine.cairo Description The Shrine is the main accounting module in the system. It keeps track of open troves, the total yang deposited in the system, and the weighted yang threshold. To get the total system health, the get_shrine_health function is used: // Returns a Health struct comprising the Shrine's threshold, LTV, value and debt; fn get_shrine_health (self : @ContractState) → Health { let (threshold, value) = self .get_threshold_and_value(self .get_shrine_deposits(), now()); let debt: Wad = self .total_troves_debt.read(); // If no collateral has been deposited, then shrine's LTV is // returned as the maximum possible value. let ltv: Ray = if value.is_zero() { BoundedRay::max() } else { wadray::rdiv_ww(debt, value) }; Health { threshold, ltv, value, debt } } Figure 6.1: The get_shrine_health function in shrine.cairo#L512-L526 However, this function fails to account for any interest or pulled redistributed debt, as the charge function, which accrues interest, is never called. In addition, the get_threshold_and_value function, which determines the weighted threshold, does not check whether recovery mode is active. Exploit Scenario The Shrine enters recovery mode, and the Caretaker attempts to gracefully shut down the system. Alice, the admin of the Opus contracts, calls shut . However, the lack of interest accrued on the debt causes trove owners to be able to withdraw more collateral than originally intended. HashEye 25 Opus Contracts Security Assessment PUBLIC

Recommendations Short term, have the get_shrine_health function call the charge function for every trove to ensure that interest is accrued. Long term, improve the unit test coverage to uncover edge cases and ensure that the system behaves as intended. HashEye 26 Opus Contracts Security Assessment PUBLIC

7. Yin cannot be pulled out of the Absorber if the Shrine is killed Severity: High Difficulty: Medium Type: Data Validation Finding ID: TOB-OPUS-7 Target: src/core/absorber.cairo Description When the system is shut down, the Caretaker module is responsible for withdrawing as much collateral as possible to back the yin supply as close to 1:1 as possible. To do this, the shut function is called. It will mint any final budget surpluses, calculate the amount of yang needed to back the yin supply, and pull the collateral from the Sentinel . It will then proceed to irreversibly kill the Shrine . Notably, the shut function does not kill the Absorber , in order to allow stakers to withdraw their yin. Trove owners can then call the release function to reclaim any remaining yang in a trove: // Loop over yangs deposited in trove and transfer to trove owner loop { match yangs_copy.pop_front() { Option :: Some (yang) ⇒ { let deposited_yang: Wad = shrine.get_deposit(*yang, trove_id); let asset_amt: u128 = if deposited_yang.is_zero() { 0 } else { let exit_amt: u128 = sentinel.exit(*yang, trove_owner, trove_id, deposited_yang); // Seize the collateral only after assets have been // transferred so that the asset amount per yang in Gate // does not change and user receives the correct amount shrine.seize(*yang, trove_id, deposited_yang); exit_amt }; released_assets.append(AssetBalance { address: *yang, amount: asset_amt }); }, Option :: None ⇒ { break; }, }; }; Figure 7.1: Seizing of collateral from the Shrine in caretaker.cairo#L285-L304 However, as a result of shutting down the system and withdrawing yang from the Shrine , the Shrine ’s loan-to-value (LTV) ratio increases, and there is a high chance that recovery mode will be enabled. This is problematic because withdrawing yin from the Absorber is HashEye 27 Opus Contracts Security Assessment PUBLIC

not possible when recovery mode is active; furthermore, since the system’s total yin supply is fixed at shutdown, it is impossible to turn recovery mode off: fn assert_can_remove (self : @ContractState, request: Request) { assert (! self .shrine.read().is_recovery_mode(), 'ABS: Recovery Mode active'); assert (request .timestamp.is_non_zero(), 'ABS: No request found'); assert (!request .has_removed, 'ABS: Only 1 removal per request'); let current_timestamp: u64 = starknet::get_block_timestamp(); let removal_start_timestamp: u64 = request .timestamp + request .timelock; assert (removal_start_timestamp ⩽ current_timestamp , 'ABS: Request is not valid yet'); assert (current_timestamp ⩽ removal_start_timestamp + REQUEST_VALIDITY_PERIOD, 'ABS: Request has expired'); } Figure 7.2: The assert_can_remove helper function in absorber.cairo#L896-L906 Exploit Scenario The Caretaker shuts down the system, and trove owners call release to reclaim any leftover collateral. As a result, the Shrine enters recovery mode because the system’s total LTV surpassed the 70% system threshold. Alice, a staker in the Absorber , goes to remove her yin in order to call the reclaim function and get back a proportion of system collateral; however, because recovery mode is active, she is prevented from withdrawing her yin and loses her assets permanently. Recommendations Short term, allow users to remove their yin when the Shrine is killed regardless of whether recovery mode is active. Long term, improve the unit test coverage to uncover edge cases and ensure that the system behaves as intended. Ensure that all system features are

appropriately disabled or enabled during component shutdowns, and document invariants that should hold when the system components are not live. HashEye 28 Opus Contracts Security Assessment PUBLIC

8. Exceptionally redistributed yangs are not included in compensation for absorptions Severity: Medium Difficulty: Medium Type: Timing Finding ID: TOB-OPUS-8 Target: src/core/absorber.cairo Description During a redistribution, if the redistributed trove has yangs that no other troves have deposited as collateral, then that yang will be split evenly among all of the other yangs. This is known as an exceptional redistribution. Because an exceptional redistribution changes the collateral composition of a trove, the pull_redistributed_debt_and_yangs function is used to return an array of updated yang amounts for a trove. The get_trove_health function then uses these values to calculate a trove's health after an exceptional redistribution: // Calculate debt let compounded_debt: Wad = self .compound(trove_id, trove, interval); let (updated_trove_yang_balances, compounded_debt_with_redistributed_debt) = self .pull_redistributed_debt_and_yangs(trove_id, trove_yang_balances, compounded_debt); if updated_trove_yang_balances.is_some() { let (new_threshold, new_value) = self .get_threshold_and_value(updated_trove_yang_balances.unwrap(), interval); threshold = self .scale_threshold_for_recovery_mode(new_threshold); value = new_value; } let ltv: Ray = wadray::rdiv_ww(compounded_debt_with_redistributed_debt, value); Health { threshold, ltv, value, debt: compounded_debt_with_redistributed_debt } Figure 8.1: The calculation done by the get_trove_health function in shrine.cairo#L1068-L1082 During an absorption, the return values of the get_trove_health function are used to calculate the percentage of compensation given to the caller. This percentage is then used in the free function to remove collateral from the trove and send said collateral to the caller: HashEye 29 Opus Contracts Security Assessment PUBLIC

let trove_health: Health = shrine.get_trove_health(trove_id); let (trove_penalty, max_purge_amt, pct_value_to_compensate, _, ltv_after_compensation, value_after_compensation) = self .preview_absorb_internal(trove_health) .expect('PU: Not absorbable'); let caller: ContractAddress = get_caller_address(); let absorber: IAbsorberDispatcher = self .absorber.read(); // If the absorber is operational, cap the purge amount to the absorber's balance // (including if it is zero). let purge_amt = if absorber.is_operational() { min(max_purge_amt, shrine.get_yin(absorber.contract_address)) } else { WadZeroable::zero() }; // Transfer a percentage of the penalty to the caller as compensation let compensation_assets: Span < AssetBalance > = self .free(shrine, trove_id, pct_value_to_compensate, caller); // Melt the trove's debt using the absorber's yin directly // This needs to be called even if `purge_amt` is 0 so that accrued interest // will be charged on the trove before `shrine.redistribute`. // This step is also crucial because it would revert if the Shrine has been killed, thereby // preventing further liquidations. shrine.melt(absorber.contract_address, trove_id, purge_amt); Figure 7.2: The call to free before the call to charge during an absorption in absorber.cairo#L269-L302 However, although the yang amounts including exceptional redistributions are used in the calculation, the trove's deposits are not correctly updated to refl ect their inclusion. The charge function (called by shrine.melt) is responsible for updating the trove's deposits; however, it is called after the compensation is given to the caller. As a result, the caller gets less compensation than intended. Exploit Scenario After an exceptional redistribution, Alice's trove is eligible for absorption. Bob goes to call absorb ; however, because the exceptionally redistributed yangs are not pulled into the trove, Bob does not receive them as compensation. HashEye 30 Opus Contracts Security Assessment PUBLIC

Recommendations Short term, move the call to shrine.melt before the call to the free function so that the trove's deposits are correctly updated. Long term, improve the unit test coverage to uncover edge cases and ensure that the system behaves as intended. Evaluate the correct order of operations for each call sequence and validate that necessary state updates are performed before important calculations or transference of funds. HashEye 31 Opus Contracts Security Assessment PUBLIC

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 Undefi ned behavior triggered within the system HashEye 32 Opus Contracts Security Assessment PUBLIC

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 defi ned 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 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 Transaction Ordering The system’s resistance to transaction-ordering attacks

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. Opus System Invariants During our review, we identified system invariants that should be true at all times and that were not included alongside the system properties described by the Opus team in the provided documentation. We recommend adding these properties to the documentation and including tests to ensure they hold. Shrine • The budget is always positive or equal to zero. • The lowest threshold that a non-suspended yang can have is 50% of the set threshold. • Yang price updates happen at most once per interval. • Depositing and withdrawing yang immediately does not result in any profit. • System debt always increases when yin is forged. • The budget defi cit never decreases when yin is forged or melted. • Redistribution IDs are monotonically increasing. • Exceptional redistributions can occur only if there is less than 1 wad of yang in another trove. • The initial 1,000 wei of deposits can rebase in value but will not accrue redistributed debt. • Redistributions always decrease the redistributed trove’s debt. • Yang values can only positively rebase if the total yang supply is decremented. Absorber • Epochs and absorption IDs are monotonically increasing. • Every new epoch has at least INITIAL_SHARES (1,000 wei) worth of shares at the start. • Rewards are available to be distributed only once per epoch. • If there is a redistribution in an epoch, stakers are eligible for rewards only before said epoch. • Rewards do not accrue to the initial shares provided at the start of the epoch. • If the Shrine is in recovery mode, providing and removing yin always reverts.

• Providing and removing yin atomically always reverts. • Yin is melted from the Absorber only if it is operational or there is a redistribution. Purger • A trove’s LTV always decreases after a liquidation or absorption. • The close amount is always less than or equal to the trove’s debt. • The penalty for an unhealthy trove increases monotonically up to the maximum. • The percentage of freed collateral is capped to 100% of the trove’s collateral. • Troves are absorbable only if the trove’s threshold is greater than the absorption threshold. • Self liquidations are not profitable.

D. Fix Review Results When undertaking a fix review, HashEye 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 February 20, 2024, HashEye reviewed the fixes and mitigations implemented by the Opus team for the issues identified in this report. We reviewed each fix to determine its eff ectiveness in resolving the associated issue. In summary, of the eight issues described in this report, Opus has resolved five

issues and has not resolved the remaining three issues. For additional information, please see the Detailed Fix Review Results below. ID Title Status 1 Redistributed debt does not accrue interest until next trove action Unresolved 2 Incorrect starting index in the bestow function Resolved 3 MAX_YANG_RATE is set lower than intended Resolved 4 LOWER_UPDATE_FREQUENCY_BOUND is set much lower than Starknet block time Unresolved 5 The absorb function can still be called even if the Absorber is killed Resolved 6 The get_shrine_health function does not account for interest or recovery mode threshold Unresolved 7 Yin cannot be pulled out of the Absorber if the Shrine is killed Resolved 8 Exceptionally redistributed yangs are not included in compensation for absorptions Resolved HashEye 38 Opus Contracts Security Assessment PUBLIC

Detailed Fix Review Results TOB-OPUS-1: Redistributed debt does not accrue interest until next trove action Unresolved. The client provided the following context for this finding’s fix status: It’s working as designed, a proper solution of accruing interest on redistributed debt would be computationally expensive, we’re fine. TOB-OPUS-2: Incorrect starting index in the bestow function Resolved in PR #513 . The starting index for bestow has been updated to 1 . TOB-OPUS-3: MAX_YANG_RATE is set lower than intended Resolved in PR #512 . The MAX_YANG_RATE constant has been updated to be 1 ray. TOB-OPUS-4: LOWER_UPDATE_FREQUENCY_BOUND is set much lower than Starknet block time Unresolved. The client provided the following context for this finding’s fix status: Due to the lack of a working keeper service on Starknet, we’ll be running the price update bot ourselves for now; if one comes along (or Yagi launches), we’ll be sure to assess if reverted TXs don’t eat up keeper credit. TOB-OPUS-5: The absorb function can still be called even if the Absorber is killed Resolved in PR #518 . Rewards are no longer distributed when the Absorber is killed; however, absorptions are still possible when the Absorber is killed. TOB-OPUS-6: The get_shrine_health function does not account for interest or recovery mode threshold Unresolved. The client provided the following context for this finding’s fix status: We’re fine with the loss of protocol income from unaccrued interest in this case. TOB-OPUS-7: Yin cannot be pulled out of the Absorber if the Shrine is killed Resolved in PR #521 . The Absorber now checks whether the Shrine is killed and, if so, allows withdrawals even if recovery mode is active. TOB-OPUS-8: Exceptionally redistributed yangs are not included in compensation for absorptions Resolved in PR #529 . The charge function is now called before the call to free , which correctly pulls exceptionally redistributed yangs into a trove. HashEye 39 Opus Contracts Security Assessment PUBLIC

E. 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. HashEye 40 Opus Contracts Security Assessment PUBLIC