HashEye

// SMART CONTRACT SECURITY AUDIT

Sandclock

Security assessment by HashEye · prepared for Lindy Labs

HASHEYE AUDITED

PROJECT	Sandclock
CLIENT	Lindy Labs
CATEGORY	Blockchain
PUBLISHED	July 1, 2023
REPORT ID	research-sandclock-2023-07-01-1hs9wa

This report was produced under HashEye's layered review process — **automated detection**, **pattern correlation**, and **senior manual verification** — with every finding signed off by a human reviewer. Full findings detail and on-chain attestation are available on the report page at hasheye.io/audits/research-sandclock-2023-07-01-1hs9wa.

Lindy Labs Sandclock Security Assessment August 23, 2023 Prepared for: Cristiano Teixeira Lindy Labs Prepared by: Bo Henderson and Guillermo Larregay

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. 228 Park Ave S #80688 New York, NY 10003 https://www.hasheye.io info@hasheye.io HashEye 1 Lindy Labs Sandclock Security Assessment PUBLIC

 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 Lindy Labs Sandclock Security Assessment PUBLIC

Table of Contents HashEye 3 Lindy Labs Sandclock Security Assessment PUBLIC

Executive Summary Engagement Overview Lindy Labs engaged HashEye to review the security of its Sandclock product. Sandclock is a collection of ERC-4626 vaults that provide managed investment strategies, primarily using Lido and Liquity. A team of two consultants conducted the review from June 12 to July 12, 2023, for a total of four engineer-weeks of effort. Our testing efforts focused on the flow of funds and interactions with a variety of third-party financial service providers. With full access to the source code and documentation, we performed static and dynamic testing of the Sandclock vaults, using automated and manual processes. Observations and Impact Sandclock is a mature project with well-organized logic and sensible design decisions, such as those related to the use of adapters and library contracts to manage third-party interactions. However, the documentation is still a work in progress. Both the system's unit and fuzz tests provide thorough

coverage of the core functionality, and security tools such as Slither are integrated into automated CI processes. Few issues were discovered in the core business logic, and those that were discovered include a minor arithmetic issue that would delay user withdrawals only in certain edge cases (TOB-SANDCLOCK-3). The protocol extensively uses third-party financial services, including but not limited to Balancer, Lido, Aave, Chainlink, and Uniswap. Some of these integrations introduce more severe risks than others. For example, Balancer flash loans may begin charging fees at some point in the future, but the possibility of fees is not accounted for in the Sandclock system. If fees for flash loans were enabled, then core functionality of the Sandclock vaults would be irreparably broken, leading to severe loss of user funds (TOB-SANDCLOCK-1). Less severe integration issues include the acceptance of potentially stale price data from Chainlink oracles (TOB-SANDCLOCK-6). 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. • Write comprehensive documentation. Apart from inline code comments, only very high-level documentation was provided. Building out comprehensive documentation will help onboard new developers more quickly, help future auditors

perform more effective reviews, and help users interact more confidently with Sandclock services. We recommend focusing on the following areas while writing this documentation. ○ How-to guides: Provide a list of the different types of users (e.g., USDC investors, QUARTZ stakers, etc.) and user stories describing the actions they can take (e.g., calling `mint` , `withdraw` , `depositIntoStrategy` , etc.), the process for taking those actions, the effects they have, and the risks associated with them. Similarly, describe how the keepers and other administrative roles should interact with the system; this documentation will increase transparency, provide a valuable reference for future system administrators, and help future auditors ramp up on the business logic more quickly. ○ NatSpec comments: Many of the system’s functions already have NatSpec comments describing them, the parameters they receive, and the values they return; however, not all of the functions have these comments. Add NatSpec comments to all of the Sandclock functions, including those that are internal and access-controlled. Additionally, review all existing NatSpec comments to ensure that they provide detailed specifications, including reasons the function might revert and a list of relevant side effects or known risks. • Resolve known share price inflation risks. At the beginning of our engagement, Lindy Labs disclosed that the ERC-4626 vaults under review are vulnerable to share price inflation. We recommend mitigating this risk prior to deployment by taking one of the following actions: ○ Require the deployer to make an initial, non-withdrawable deposit from the vault constructor. ○ Include virtual shares and assets in the exchange rate calculation. ○ Represent vault shares with additional precision, as described in this conversation on the Ethereum Magicians forum . • Expand the tests for the RewardTracker contract and its dependencies. In general, this project has thorough test coverage. Foundry reports 100% code coverage for most of the contracts under review, and mutation testing produces zero valid mutants for many of the contracts. However, mutation testing of the RewardTracker contract and its dependencies produces a relatively large number of valid mutants, indicating that they have many untested conditional statements. We recommend incorporating mutation testing strategies for the RewardTracker contract and its dependencies to provide a more dynamic view of the test coverage

and expanding the unit tests for the contract until all true-positive valid mutants are disqualified. • Maintain thorough off-chain monitoring. The extensive interactions with third-party financial services means that the security of Sandclock vaults is tied to the security of these other services. Sandclock’s public incident response plan is an important step in the right direction and shows that the Lindy Labs team is taking security seriously, but putting the plan into action in a timely manner requires ongoing monitoring. The internal due diligence section of Sandclock’s security documentation references the use of OpenZeppelin Defender, which is a valuable tool for administering and monitoring the health of the core Sandclock contracts; however, we recommend also monitoring the health of third-party services (e.g., using a subgraph or cryo), Twitter keywords, and the uptime of Chainlink oracles. The following tables provide the number of findings by severity and category. EXPOSURE ANALYSIS Severity Count High 1 Medium 0 Low 2 Informational 3 Undetermined 0 CATEGORY BREAKDOWN Category Count Data Validation 6

Project Summary Contact Information The following managers were associated with this project: Dan Guido , Account Manager Jeff Braswell , Project Manager dan@hasheye.io jeff.braswell@hasheye.io The following engineers were associated with this project: Bo Henderson , Consultant Guillermo Larregay , Consultant bo.henderson@hasheye.io guillermo.larregay@hasheye.io Project Timeline The significant events and milestones of the project are listed below. Date Event June 8, 2023 Pre-project kickoff

call June 16, 2023 Status update meeting #1 June 22, 2023 Status update meeting #2 June 29, 2023 Status update meeting #3 July 6, 2023 Status update meeting #4 July 18, 2023 Delivery of report draft July 19, 2023 Report readout meeting August 23, 2023 Delivery of final report with fix review HashEye 7 Lindy Labs Sandclock Security Assessment PUBLIC

Project Goals The engagement was scoped to provide a security assessment of the Lindy Labs Sandclock product. Specifically, we sought to answer the following non-exhaustive list of questions: • Are there any incorrect or error-prone steps in the deployment process? • Are appropriate access controls in place for administrative roles? • Is the system able to recover from insolvency or being liquidated on one of the lending platforms it interacts with? • Are asset swaps properly protected against extreme slippage and front-running? • Is there any way for internal accounting to become out of sync with the true asset balances of the system contracts? • Are there any edge cases or integration considerations that the system fails to take into account while interacting with third-party services? • Does the system's arithmetic match the expected business logic? Does it properly handle overflows, underflows, and invalid values provided by attackers? • Is there any way that a user could be prevented from withdrawing their funds? HashEye 8 Lindy Labs Sandclock Security Assessment PUBLIC

Project Targets The engagement involved a review and testing of the following target. sandclock-contracts Repository https://github.com/lindy-labs/sandclock-contracts Version a100f21a30dd332b69351d1e05d98dbc748c6ddc Type Solidity Platform Ethereum HashEye 9 Lindy Labs Sandclock 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: • scWETHv2 : This contract is an ERC-4626 vault that manages WETH deposits. A privileged keeper account uses Balancer flash loans to borrow WETH, deposit it into Lido in exchange for wstETH, and provide the wstETH as collateral to borrow WETH and to repay the flash loan. We reviewed this vault's interactions with Lido, Balancer, and other third-party contracts to ensure that it interacts with them safely; this review uncovered issues such as finding TOB-SANDCLOCK-1 . We traced the flow of funds and compared it to the internal accounting state variables, paying particular attention to the float and withdrawal process when not enough float is available. • scUSDCv2 : This contract is an ERC-4626 vault that manages USDC deposits. It invests USDC by supplying it as collateral, borrowing WETH, and supplying this WETH to the scWETHv2 contract. Although the scUSDCv2 vault offloads much of its investment logic to the scWETHv2 vault, it includes an emergency exit feature to recover from an underwater financial position. We reviewed the flow of funds through the emergency exit process, its use of flash loans, and the existing safeguards against slippage and front-running risks. • Adapters: The scWETHv2 and scUSDCv2 contracts use collateralized loans through a collection of third-party lending pools. An adapter is present for each lending pool, giving all of the lending pools a standard interface for the Sandclock vaults to interact with. We reviewed the external methods called by these adapters to ensure that third-party contracts are interacted with according to documented best practices. We also verified that the methods for adding, removing, and looping through these adapters are free of error. The adapters that we focused on during this review are enumerated below: ○ scUSDCv2 adapters: ■ AaveV2Adapter ■ AaveV3Adapter ○ scWETHv2 adapters: ■ AaveV3Adapter ■ CompoundAdapter ■ MorphoAaveV3Adapter HashEye 10 Lindy Labs Sandclock Security Assessment PUBLIC

• PriceConverter : This library contract exposes functions to interact with Chainlink and Lido oracles for the vaults to use while setting slippage limits, defining the share-to-asset exchange rate, and calculating required flash loan amounts. We reviewed these oracle interactions, and we identified that the protocol is at risk of using stale price data, as described in finding TOB-SANDCLOCK-6 . • Swapper : This contract implements functions to perform swaps between the different supported assets. Decentralized exchanges such as Uniswap, 0x, and Curve are used for most swaps; however, the conversion from WETH to wstETH is done via Lido and its respective wrapper. We reviewed these integrations to ensure that the exchanges are used with adequate protection against front-running and slippage. • scLiquity : This contract is an ERC-4626 vault that invests LUSD into the Liquity Stability Pool, compounding ETH received from liquidations into additional LUSD deposits. We reviewed the reward harvesting process and the disinvestment of funds during withdrawal. • RewardTracker : Sandclock's native token, QUARTZ, can be staked in the RewardTracker contract to earn a portion of the performance fees collected by the scWETHv2 vault. We reviewed the bonus and debt features that incentivize long-term staking and compared the flow of funds on-chain to the internal accounting mechanisms; this review identified a risk of misalignment between the reward distribution rate and the reward token balance, as described in finding TOB-SANDCLOCK-2 . 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: • Although we reviewed the Sandclock

system’s interactions with third-party lending pools, oracles, exchanges, and other financial services, we did not closely review the code of these third-party services. The security of the Sandclock platform depends on the security of these services, and any exploits that target them could severely impact Sandclock. • The scWETH and scUSDC contracts, the first versions of the vaults, were explicitly out of scope and, therefore, were not reviewed at all. We did briefly scan them to gain context for our review of the scWETHv2 and scUSDCv2 systems. • We received only a very general written overview of how the system should work. Apart from code comments, the system’s documentation is largely lacking. Lindy Labs was responsive during the engagement and answered our questions about HashEye 11 Lindy Labs Sandclock Security Assessment PUBLIC

undocumented aspects of the system thoroughly and in a timely manner. However, due to the lack of a formal specification, we may have some gaps in our understanding of the intended behavior of the system. • Due to a recent exploit that targeted Euler Finance, we deprioritized our analysis of the EulerAdapter contracts for both the scWETHv2 and scUSDCv2 vaults. Once Euler has recovered, relaunched, and stabilized, we recommend re-reviewing these adapters, taking into account any changes made to the Euler system. HashEye 12 Lindy Labs Sandclock Security Assessment PUBLIC

Automated Testing HashEye uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software. Test Harness Configuration We used the following tools in the automated testing phase of this project: Tool Description Policy slither A static analysis framework that can statically verify algebraic relationships between Solidity variables Used to detect common issues echidna A smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation crytic/properties necessist A mutation generator that flags unnecessary or unused lines in a test suite Used to assess test coverage universalmutator A source code mutation generator that helps detect untested logic Used to assess test coverage Testing of standard ERC-4626 invariants with Echidna: We ran Echidna against a library of premade invariants for contracts that implement an ERC-4626 interface. These invariants can be found in our open-source crytic/properties repository. Apart from an expected failure for the property that tests for vulnerabilities to share price inflation attacks (a known issue for all vaults), two other failed properties (those ensuring that msg.sender is not considered while exchange rates are being calculated) were identified as false positives. We received high-confidence results for the scLiquity vault by reusing the mocked services from unit tests, but we encountered frequent reverts while testing scWETHv2 and scUSDCv2 due to their extensive use of third-party financial services. Although no other properties explicitly failed, frequent transaction reversions and the resulting poor fuzzing coverage means that we have lower confidence in the results of these tests. universalmutator : The following table displays the proportion of mutants for which all unit tests passed. The presence of zero valid mutants indicates that coverage is thorough. A nonzero number of true-positive valid mutants indicates gaps in the test coverage where errors may go unnoticed. Zero true positives were identified for the sc4626 and HashEye 13 Lindy Labs Sandclock Security Assessment PUBLIC

Constants contracts; many true positives were identified for RewardTracker and its dependencies. Target Valid Mutants src/lib/Constants.sol 0.2% src/steth/PriceConverter.sol 0.0% src/steth/Swapper.sol 0.0% src/sc4626.sol 5.3% src/steth/BaseV2Vault.sol 0.0% src/steth/scWETHv2.sol 0.0% src/steth/scWethV2-adapters/CompoundV3Adapter.sol 0.0% src/steth/scWethV2-adapters/MorphoAaveV3Adapter.sol 0.0% src/steth/scWethV2-adapters/AaveV3Adapter.sol 0.0% src/steth/scUSDCv2.sol 0.0% src/steth/scUsdcV2-adapters/AaveV2Adapter.sol 0.0% src/steth/scUsdcV2-adapters/AaveV3Adapter.sol 0.0% src/liquity/scLiquity.sol 0.0% src/staking/BonusTracker.sol 6.1% src/staking/DebtTracker.sol 5.3% src/staking/RewardTracker.sol 9.9% HashEye 14 Lindy Labs Sandclock 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. 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 codebase uses a 0.8.x version of the Solidity compiler to protect against overflows and underflows, and no arithmetic is unchecked. However, no arithmetic specifications were provided, and some arithmetic lacks inline comments justifying expectations, such as those regarding rounding behavior. Minor arithmetic issues were identified (TOB-SANDCLOCK-3 , TOB-SANDCLOCK-4). Moderate Auditing All critical functions emit sufficient events for off-chain monitoring, although events are poorly documented. An off-chain monitoring strategy is mentioned in the documentation but lacks details regarding triggers/notifications. An incident response plan is publicly available. Satisfactory Authentication / Access Controls Privileged roles are split up where applicable according to the principle of least privilege, although the responsibilities of each role require

additional documentation. The use of the OpenZeppelin AccessControl library provides a well-vetted two-step transfer process. No missing access controls were identified. Satisfactory Complexity Management Libraries and library-like contracts have clear and narrow responsibilities. Inheritance is used only to the extent necessary to avoid code duplication. No individual function is overly complex, and each function has a tight responsibility. Satisfactory Decentralization The privileges of admin accounts are tightly controlled, preventing them from unilaterally seizing user funds; however, the system depends on centralized actors Weak HashEye 15 Lindy Labs Sandclock Security Assessment PUBLIC

(keepers) to manage investments, exposing a single point of failure for some functionality. The system's reliance on third-party contracts (such as Chainlink oracles or pools) and the associated risks are not documented. In edge cases (TOB-SANDCLOCK-4), privileged accounts are required for some users to withdraw their funds. The system is not upgradeable. No time delay or upper bound is present on fees taken from investment income, meaning users do not have a time window to opt out of changes. Documentation Very little documentation and no system specifications were provided by the team. However, in terms of code documentation, NatSpec comments are present for most public methods, and inline comments are included for limited parts of the protocol logic. In general, the unit tests are poorly documented with the exception of occasional brief inline comments. Weak Front-Running Resistance Price information from uncorrelated oracles and liquidity pools limit but do not eliminate the impact of front-running during swaps, as described in finding TOB-SANDCLOCK-6 . The team is aware of the possibility of a share inflation attack but has not implemented a solution yet. MEV attacks are not tested or well documented. Moderate Low-Level Manipulation No assembly is used in the codebase (except via imported libraries). Low-level interactions with the 0x router in the scLiquity vault have tightly controlled approvals, although they lack documentation and balance checks before and after, as is done by similar 0x router interactions in the Swapper library. Moderate Testing and Verification The test coverage for Sandclock vaults is thorough. Mutation testing revealed gaps only in the coverage for RewardTracker and its dependencies. Testing and static analysis is performed as part of an automated CI process. Satisfactory HashEye 16 Lindy Labs Sandclock 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 receiveFlashLoan does not account for fees Data Validation High 2 Reward token distribution rate can diverge from reward token balance Data Validation Low 3 Miscalculation in beforeWithdraw can leave the vault with less than minimum float Data Validation Informational 4 Last user in scWETHv2 vault will not be able to withdraw their funds Data Validation Low 5 Lido stake rate limit could lead to unexpected reverts Data Validation Informational 6 Chainlink oracles could return stale price data Data Validation Informational HashEye 17 Lindy Labs Sandclock Security Assessment PUBLIC

Detailed Findings 1. receiveFlashLoan does not account for fees Severity: High Difficulty: High Type: Data Validation Finding ID: TOB-SANDCLOCK-1 Target: src/steth/scWETHv2.sol , src/steth/scUSDCv2.sol Description The receiveFlashLoan functions of the scWETHv2 and scUSDCv2 vaults ignore the Balancer flash loan fees and repay exactly the amount that was loaned. This is not currently an issue because the Balancer vault does not charge any fees for flash loans. However, if Balancer implements fees for flash loans in the future, the Sandclock vaults would be prevented from withdrawing investments back into the vault. function flashLoan (IFlashLoanRecipient recipient, IERC20[] memory tokens, uint256 [] memory amounts, bytes memory userData) external override nonReentrant whenNotPaused { uint256 [] memory feeAmounts = new uint256 [](tokens.length); uint256 [] memory preLoanBalances = new uint256 [](tokens.length); for (uint256 i = 0 ; i < tokens.length; ++i) { IERC20 token = tokens[i]; uint256 amount = amounts[i]; preLoanBalances[i] = token.balanceOf(address (this)); feeAmounts[i] = _calculateFlashLoanFeeAmount(amount); token.safeTransfer(address (recipient), amount); } recipient.receiveFlashLoan(tokens, amounts, feeAmounts , userData); for (uint256 i = 0 ; i < tokens.length; ++i) { IERC20 token = tokens[i]; uint256 preLoanBalance = preLoanBalances[i]; uint256 postLoanBalance = token.balanceOf(address (this)); uint256 receivedFeeAmount = postLoanBalance - preLoanBalance; _require(receivedFeeAmount ⩾ feeAmounts[i]); _payFeeAmount(token, receivedFeeAmount); } } Figure 1.1: Abbreviated code showing the receivedFeeAmount check in the Balancer flashLoan method in 0xBA12222222228d8Ba445958a75a0704d566BF2C8#code#F5#L78 HashEye 18 Lindy Labs Sandclock Security Assessment PUBLIC

In the Balancer flashLoan function , shown in figure 1.1, the contract calls the recipient's receiveFlashLoan function with four arguments: the addresses of the tokens loaned, the amounts for each token, the fees to be paid for the loan for each token, and the calldata provided by the caller. The Sandclock vaults ignore the fee amount and repay only the principal, which would lead to reverts if the fees are ever changed to nonzero values. Although this problem is present in

multiple vaults, the receiveFlashLoan implementation of the scWETHv2 contract is shown in figure 1.2 as an illustrative example: function receiveFlashLoan (address [] memory , uint256 [] memory amounts, uint256 [] memory , bytes memory userData) external { _isFlashLoanInitiated(); // the amount flashloaned uint256 flashLoanAmount = amounts[0]; // decode user data bytes [] memory callData = abi.decode(userData, (bytes [])); _multiCall(callData); // payback flashloan asset.safeTransfer(address (balancerVault), flashLoanAmount); _enforceFloat(); } Figure 1.2: The feeAmounts parameter is ignored by the receiveFlashLoan method. (sandclock-contracts/src/steth/scWETHv2.sol#L232-L249) Exploit Scenario After Sandclock's scUSDv2 and scWETHv2 vaults are deployed and users start depositing assets, the Balancer governance system decides to start charging fees for flash loans. Users of the Sandclock protocol now discover that, apart from the float margin, most of their funds are locked because it is impossible to use the flash loan functions to withdraw vault assets from the underlying investment pools. Recommendations Short term, use the feeAmounts parameter in the calculation for repayment to account for future Balancer flash loan fees. This will prevent unexpected reverts in the flash loan handler function. Long term, document and justify all ignored arguments provided by external callers. This will facilitate a review of the system's third-party interactions and help prevent similar issues from being introduced in the future. HashEye 19 Lindy Labs Sandclock Security Assessment PUBLIC

2. Reward token distribution rate can diverge from reward token balance Severity: Low Difficulty: High Type: Data Validation Finding ID: TOB-SANDCLOCK-2 Target: src/staking/RewardTracker.sol Description The privileged distributor role is responsible for transferring reward tokens to the RewardTracker contract and then passing the number of tokens sent as the _reward parameter to the notifyRewardAmount method. However, the _reward parameter provided to this method can be larger than the number of reward tokens transferred. Given the accounting for leftover rewards, such a situation would be difficult to recover from. /// @notice Lets a reward distributor start a new reward period. The reward tokens must have already /// been transferred to this contract before calling this function. If it is called /// when a reward period is still active, a new reward period will begin from the time /// of calling this function, using the leftover rewards from the old reward period plus /// the newly sent rewards as the reward. /// @dev If the reward amount will cause an overflow when computing rewardPerToken, then /// this function will revert. /// @param _reward The amount of reward tokens to use in the new reward period. function notifyRewardAmount (uint256 _reward) external onlyDistributor { _notifyRewardAmount(_reward); } Figure 2.1: The comment on the notifyRewardAmount method hints at an unenforced assumption that the number of reward tokens transferred must be equal to the _reward parameter provided. (sandclock-contracts/src/staking/RewardTracker.sol#L185-L195) If a _reward value smaller than the actual number of transferred tokens is provided, the situation can be fixed by calling notifyRewardAmount again with a _reward parameter that accounts for the difference between the RewardTracker contract's actual token balance and the rewards already scheduled for distribution. This solution is possible because the _notifyRewardAmount helper function accounts for leftover rewards if it is called during an ongoing reward period. HashEye 20 Lindy Labs Sandclock Security Assessment PUBLIC

function _notifyRewardAmount (uint256 _reward) internal { ... uint64 rewardRate_ = rewardRate; uint64 periodFinish_ = periodFinish; uint64 duration_ = duration; ... if (block.timestamp ≥ periodFinish_) { newRewardRate = _reward / duration_; } else { uint256 remaining = periodFinish_ - block.timestamp ; uint256 leftover = remaining * rewardRate_; newRewardRate = (_reward + leftover) / duration_; } Figure 2.2: The accounting for leftover rewards in the _notifyRewardAmount helper method (sandclock-contracts/src/staking/RewardTracker.sol#L226-L262) This accounting for leftover rewards, however, makes the situation difficult to recover from if a _reward parameter that is too large is provided to the notifyRewardAmount method. As shown by the arithmetic in figure 2.2, if the reward period has not finished, the code for creating the newRewardRate value can only add to the reward distribution, not subtract from it. The only way to bring a too-large reward distribution back in line with the RewardTracker contract's reward token balance is to transfer additional reward tokens to the contract. Exploit Scenario The RewardTracker distributor transfers 10 reward tokens to the RewardTracker contract and then mistakenly calls the notifyRewardAmount method with a _reward parameter of 100. Some users call the claimRewards method early and receive inflated rewards until the contract's balance is depleted, leaving later users unable to claim any rewards. To recover, the distributor either needs to provide another 90 reward tokens to the RewardTracker contract or accept the reputational loss of allowing this misconfigured reward period to finish before resetting the reward payouts correctly during the next period. Recommendations Short term, modify the _notifyRewardAmount helper function to reset the rewardRate so that it is in line with the current rewardToken balance and the time remaining in the reward period. This change could also allow the fetchRewards method to maintain its current behavior but with only a single rewardToken.balanceOf external call. Long term, review the internal accounting state variables and document the ways in which they are influenced by the actual flow of funds. Pay attention to any internal accounting values that can be influenced by external sources, including privileged

accounts, and reexamine the system’s assumptions surrounding the flow of funds. HashEye 21 Lindy Labs Sandclock Security Assessment PUBLIC

3. Miscalculation in beforeWithdraw can leave the vault with less than minimum float Severity: Informational Difficulty: High Type: Data Validation Finding ID: TOB-SANDCLOCK-3 Target: src/steth/scWETHv2.sol Description When a user wants to redeem or withdraw, the beforeWithdraw function is called with the number of assets to be withdrawn as the assets parameter. This function makes sure that if the value of the float parameter (that is, the available assets in the vault) is not enough to pay for the withdrawal, the strategy gets some assets back from the pools to be able to pay. function beforeWithdraw (uint256 assets , uint256) internal override { uint256 float = asset.balanceOf(address (this)); if (assets ≤ float) return ; uint256 minimumFloat = minimumFloatAmount; uint256 floatRequired = float < minimumFloat ? minimumFloat - float : 0 ; uint256 missing = assets + floatRequired - float; _withdrawToVault(missing); } Figure 3.1: The affected code in sandclock-contracts/src/steth/scWETHv2.sol#L386-L396 When the float value is enough, the function returns and the withdrawal is paid with the existing float. If the float value is not enough, the missing amount is recovered from the pools via the adapters. The issue lies in the calculation of the missing parameter: it does not guarantee that the float value remaining after the withdrawal is at least the value of the minimumFloatAmount parameter. The consequence is that the calculation always leaves a float equal to floatRequired in the vault. If this value is small enough, it can cause users to waste gas when withdrawing small amounts because they will need to pay for the gas-intensive _withdrawToVault action. This eclipses the usefulness of having the float in the vault. HashEye 22 Lindy Labs Sandclock Security Assessment PUBLIC

The correct calculation should be uint256 missing = assets + minimumFloat - float; . Using this correct calculation would make the calculation of the floatRequired parameter unnecessary as it would no longer be required or used in the rest of the code. Exploit Scenario The value for minimumFloatAmount is set to 1 ether in the scWETHv2 contract. For this scenario, suppose that the current float is exactly equal to minimumFloatAmount . Alice wants to withdraw 0.15 WETH from her invested amount. Because this amount is less than the current float, her withdrawal is paid from the vault assets, leaving the float equal to 0.85 WETH after the operation. Then, Bill wants to withdraw 0.9 WETH, but the vault has no available assets to pay for it. In this case, when beforeWithdraw is called, Bill has to pay gas for the call to _withdrawToVault , which is an expensive action because it includes gas-intensive operations such as loops and a flash loan. After Bill’s withdrawal, the float in the vault is 0.15 WETH. This is a relatively small amount compared with minimumFloatValue , and it will likely make the next withdrawing/redeeming user also have to pay for the call to _withdrawToVault . Recommendations Short term, replace the calculation of the missing amount to be withdrawn on line 393 of the scWETHv2 contract with assets + minimumFloat - float . This calculation will ensure that the minimum float restriction is enforced after withdrawals. It will take the required float into consideration, so the separate calculation of floatRequired on line 392 of scWETHv2 would no longer be required. Long term, add unit or fuzz tests to make sure that the vault has an amount of assets equal to or greater than the minimum expected amount at all times. HashEye 23 Lindy Labs Sandclock Security Assessment PUBLIC

4. Last user in scWETHv2 vault will not be able to withdraw their funds Severity: Low Difficulty: Medium Type: Data Validation Finding ID: TOB-SANDCLOCK-4 Target: src/steth/scWETHv2.sol Description When a user wants to withdraw, the withdrawal amount is checked against the current vault float (the uninvested assets readily available in the vault). If the withdrawal amount is less than the float, the amount is paid from the available balance; otherwise, the protocol has to disinvest from the strategies to get the required assets to pay for the withdrawal. The issue with this approach is that in order to maintain a float equal to the minimumFloatValue parameter in the vault, the value to be disinvested from the strategies is calculated in the beforeWithdraw function, and its correct value is equal to the sum of the amount to be withdrawn and the minimum float minus the current float. If there is only one user remaining in the vault and they want to withdraw, this enforcement will not allow them to do so, because there will not be enough invested in the strategies to leave a minimum float in the vault after the withdrawal. They would only be able to withdraw their assets minus the minimum float at most. The code for the _withdrawToVault function is shown in figure 4.1. The line highlighted in the figure would cause the revert in this situation, as there would not be enough invested to supply the requested amount. function _withdrawToVault (uint256 _amount) internal { uint256 n = protocolAdapters.length(); uint256 flashLoanAmount ; uint256 totalInvested_ = _totalCollateralInWeth() - totalDebt(); bytes [] memory callData = new bytes [](n + 1); uint256 flashLoanAmount_ ; uint256 amount_ ; uint256 adapterId ; address adapter ; for (uint256 i ; i < n; i++) { (adapterId, adapter) = protocolAdapters.at(i); (flashLoanAmount_, amount_) = _calcFlashLoanAmountWithdrawing(adapter, _amount, totalInvested_); flashLoanAmount += flashLoanAmount_; HashEye 24 Lindy Labs Sandclock Security Assessment PUBLIC

callData[i] = abi.encodeWithSelector(this .repayAndWithdraw.selector, adapterId, flashLoanAmount_, priceConverter.ethToWstEth(flashLoanAmount_ + amount_)); } // needed otherwise counted as loss during harvest totalInvested -= _amount; callData[n] = abi.encodeWithSelector(scWETHv2.swapWstEthToWeth.selector, type(uint256).max, slippageTolerance); uint256 float = asset.balanceOf(address (this)); _flashLoan(flashLoanAmount, callData); emit WithdrawnToVault(asset.balanceOf(address (this)) - float); } Figure 4.1: The affected code in sandclock-contracts/src/steth/scWETHv2.sol#L342-L376 Additionally, when this revert occurs, an integer overflow is given as the reason, which obscures the real reason and can make the user's experience more confusing. Exploit Scenario Bob is the only remaining user in a scWETHv2 vault, and he has 2 ether invested. He wants to withdraw his assets, but all of his calls to the withdrawal function keep reverting due to an integer overflow. He keeps trying, wasting gas in the process, until he discovers that the maximum amount he is allowed to withdraw is around 1 ether. The rest of his funds are locked in the vault until the keeper makes a manual call to withdrawToVault or until the admin lowers the minimum float value. Recommendations Short term, fix the calculation of the amount to be withdrawn and make sure that it never exceeds the total invested amount. Long term, add end-to-end unit or fuzz tests that are representative of the way multiple users can interact with the protocol. Test for edge cases involving various numbers of users, investment amounts, and critical interactions, and make sure that the protocol's invariants hold and that users do not lose access to funds in the event of such edge cases.

5. Lido stake rate limit could lead to unexpected reverts Severity: Informational Difficulty: High Type: Data Validation Finding ID: TOB-SANDCLOCK-5 Target: src/steth/Swapper.sol Description To mitigate the effects of a surge in demand for stETH on the deposit queue, Lido has implemented a rate limit for stake submissions. This rate limit is ignored by the lidoSwapWethToWstEth method of the Swapper library, potentially leading to unexpected reversions. The Lido stETH integration guide states the following: To avoid [reverts due to the rate limit being hit], you should check if getCurrentStakeLimit() ⩾ amountToStake , and if it's not you can go with an alternative route. function lidoSwapWethToWstEth (uint256 _wethAmount) external { // weth to eth weth.withdraw(_wethAmount); // stake to lido / eth ⇒ stETH stEth.submit{value: _wethAmount}(address (0x00)); // stETH to wstEth uint256 stEthBalance = stEth.balanceOf(address (this)); ERC20(address (stEth)).safeApprove(address (wstETH), stEthBalance); wstETH.wrap(stEthBalance); } Figure 5.1: The submit method is subject to a rate limit that is not taken into account. (sandclock-contracts/src/steth/Swapper.sol#L130-L142) Exploit Scenario A surge in demand for Ethereum validators leads many people using Lido to stake ETH, causing the Lido rate limit to be hit, and the submit method of the stEth contract begins to revert. As a result, the Sandclock keeper is unable to deposit despite the presence of alternate routes to obtain stETH, such as through Curve or Balancer.

Recommendations Short term, have the lidoSwapWethToWstEth method of the Swapper library check whether the amount being deposited is less than the value returned by the getCurrentStakeLimit method of the stEth contract. If it is not, have the code use ZeroEx to swap or revert with a message that clearly communicates the reason for the failure. Long term, review the documentation for all third-party interactions and note any situations in which the integration could revert unexpectedly. If such reversions are acceptable, clearly document how they could occur and include a justification for this acceptance in the inline comments.

6. Chainlink oracles could return stale price data Severity: Informational Difficulty: High Type: Data Validation Finding ID: TOB-SANDCLOCK-6 Target: src/steth/PriceConverter.sol , src/liquity/scLiquity.sol Description The latestRoundData() function from Chainlink oracles returns five values: roundId , answer , startedAt , updatedAt , and answeredInRound . The PriceConverter contract reads only the answer value and discards the rest. This can cause outdated prices to be used for token conversions, such as the ETH-to-USDC conversion shown in figure 6.1. function ethToUsdc (uint256 _ethAmount) public view returns (uint256) { (, int256 usdcPriceInEth ,,,) = usdcToEthPriceFeed.latestRoundData(); return _ethAmount.divWadDown(uint256 (usdcPriceInEth) * C.WETH_USDC_DECIMALS_DIFF); } Figure 6.1: All returned data other than the answer value is ignored during the call to a Chainlink feed's latestRoundData method. (sandclock-contracts/src/steth/PriceConverter.sol#L67-L71) According to the Chainlink documentation , if the latestRoundData() function is used, the updatedAt value should be checked to ensure that the returned value is recent enough for the application. Similarly, the LUSD/ETH price feed used by the scLiquity vault is an intermediate contract that calls the deprecated latestAnswer method on upstream Chainlink oracles. contract LSUDUsdToLUSDEth is IPriceFeed { IPriceFeed public constant LUSD_USD = IPriceFeed(0x3D7aE7E594f2f2091Ad8798313450130d0Aba3a0); IPriceFeed public constant ETH_USD = IPriceFeed(0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419); function latestAnswer () external view override returns (int256) { return (LUSD_USD.latestAnswer() * 1 ether) /

ETH_USD.latestAnswer(); } } Figure 6.2: The custom latestAnswer method in 0x60c0b047133f696334a2b7f68af0b49d2F3D4F72#code#L19 HashEye 28 Lindy Labs Sandclock Security Assessment PUBLIC

The Chainlink API reference flags the latestAnswer method as “(Deprecated - Do not use this function.).” Note that the upstream IPriceFeed contracts called by the intermediate LSUDUsdToLUSDEth contract are upgradeable proxies. It is possible that the implementations will be updated to remove support for the deprecated latestAnswer method, breaking the scLiquity vault’s lusd2eth price feed. Because the oracle price feeds are used for calculating the slippage tolerance, a difference may exist between the oracle price and the DEX pool spot price, either due to price update delays or normal price fluctuations or because the feed has become stale. This could lead to two possible adverse scenarios: • If the oracle price is significantly higher than the pool price, the slippage tolerance could be too loose, introducing the possibility of an MEV sandwich attack that can profit on the excess. • If the oracle price is significantly lower than the pool price, the slippage tolerance could be too tight, and the transaction will always revert. Users will perceive this as a denial of service because they would not be able to interact with the protocol until the price difference is settled. Exploit Scenario Bob has assets invested in a scWETHv2 vault and wants to withdraw part of his assets. He interacts with the contracts, and every withdrawal transaction he submits reverts due to a large difference between the oracle and pool prices, leading to failed slippage checks. This results in a waste of gas and leaves Bob confused, as there is no clear indication of where the problem lies. Recommendations Short term, make sure that the oracles report up-to-date data, and replace the external LUSD/ETH oracle with one that supports verification of the latest update timestamp. In the case of stale oracle data, pause price-dependent Sandclock functionality until the oracle comes back online or the admin replaces it with a live oracle. Long term, review the documentation for Chainlink and other oracle integrations to ensure that all of the security requirements are met to avoid potential issues, and add tests that take these possible situations into account. HashEye 29 Lindy Labs Sandclock 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 Undefined behavior triggered within the system HashEye 30 Lindy Labs Sandclock 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. HashEye 31 Lindy Labs Sandclock Security Assessment PUBLIC

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 HashEye 32 Lindy Labs Sandclock Security Assessment PUBLIC

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. HashEye 33 Lindy Labs Sandclock Security Assessment PUBLIC

C. Code Quality Recommendations The following recommendations are not associated with specific vulnerabilities. However, implementing them may enhance code readability and may prevent the introduction of vulnerabilities in the future. • Disambiguate the system's contract names. The EulerAdapter and AaveV3Adapter contract names refer to more than one distinct contract. Some frameworks and analysis tools will crash when a project contains two contracts with the same name. Additionally, this pattern increases the risk of miscommunication between developers, auditors, and/or users. Consider adding a prefix or suffix to such contract names or, if applicable, consolidating them into a single contract. • Consider having helper functions load values directly from storage. Many methods load values from storage and then pass them to internal helper functions. Some of these methods, enumerated below, do not use the loaded values except to pass them to the helper functions. Consider replacing the following arguments with direct storage loads. ○ BonusTracker._earnedBonus ■ accountBalance ⇒ balanceOf[account] ■ accountBonus ⇒ bonus[account] ○ RewardTracker._earned ■ _accountBalance ⇒ balanceOf[account] + multiplierPointsOf[_account] ■ _accountRewards ⇒ rewards[account] ○ RewardTracker._calcRewardPerToken ■ _totalSupply ⇒ totalSupply + totalBonus • Use helper functions consistently. An assignment to the lastTimeRewardApplicable_ variable on line 241 of the RewardTracker contract features a ternary expression identical to that of the lastTimeRewardApplicable method in the same contract. To reduce code duplication and improve readability, replace this lastTimeRewardApplicable_ assignment with a call to this helper method instead. • Remove unnecessary variable assignments. The lastUpdateTime variable is updated on line 251 of the RewardTracker contract, but this new value is not used before the same variable is updated again on line 270. Removing the first HashEye 34 Lindy Labs Sandclock Security Assessment PUBLIC

assignment on line 251 will yield identical behavior and improve the code's readability. • Use constants consistently. The addresses defined on lines 23–28 of the scLiquity contract could be specified as constant to save gas during deployment. However, to improve the maintainability of the code, consider defining these variables in the src/lib/Constants.sol file to keep all third-party contract addresses in one place. The following are the Sandclock contracts that define third-party addresses in place; consider importing them from Constants instead: ○ scLiquity ○ AaveV2Adapter (both) ○ EulerAdapter (only in scUSDCv2-adapters) ○ MorphoAaveV3Adapter ○ Swapper Note that some of these, such as the xrouter address defined by the scLiquity contract, are already present in the Constants library. Also note that some addresses in the Constants library are defined twice, such as the ZEROX_ROUTER and ZERO_EX_ROUTER addresses. • Document the omission of LQTY in the scLiquity contract's totalAssets method. The LQTY rewards earned by depositing into the Liquity Stability Pool are expected to be small, but the gas costs for factoring LQTY rewards into the scLiquity vault are significant. As a result, the Sandclock team has made the decision to omit LQTY rewards from the totalAssets value. This design decision should be clearly specified and justified in inline comments. • Document the requirements for using the Swapper contract. The Swapper contract handles ether but does not implement a receive method because it is intended to be used via delegatecall from a contract that does implement the receive method. Add comments to the Swapper contract documenting these requirements so that future developers and auditors are aware of how it should be safely used. • Use libraries in the scLiquity vault. The scLiquity vault uses error-prone low-level calls to the 0x router while harvesting proceeds for LUSD. The Swapper library implements similar swap logic with additional safety measures. Consider using the Swapper library to exchange assets instead of re-implementing this logic in place. For consistency, also consider moving the LUSD-to-ETH price-fetching logic into the PriceConverter library. HashEye 35 Lindy Labs Sandclock Security Assessment PUBLIC

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 July 27, 2023 , HashEye 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 summary, of the six issues described in this report, Lindy Labs has resolved

four, has partially resolved one, and has not resolved the remaining issue. For additional information, refer to the Detailed Fix Review Results below. ID Title Severity Status 1 receiveFlashLoan does not account for fees High Resolved 2 Reward token distribution rate can diverge from reward token balance Low Resolved 3 Miscalculation in beforeWithdraw can leave the vault with less than minimum float Informational Resolved 4 Last user in scWETHv2 vault will not be able to withdraw their funds Low Resolved 5 Lido stake rate limit could lead to unexpected reverts Informational Unresolved 6 Chainlink oracles could return stale price data Informational Partially Resolved Detailed Fix Review Results TOB-SANDCLOCK-1: receiveFlashLoan does not account for fees Resolved. The feeAmounts parameter provided by Balancer flash loans is no longer ignored by the receiveFlashLoan method. The fees are now incorporated into the asset management strategy; other updates include a modified borrow amount from Aave and a modified swap amount from Uniswap. New unit tests have been added to ensure that the

newly added logic properly accounts for nonzero fees that might be introduced by Balancer in the future. TOB-SANDCLOCK-2: Reward token distribution rate can diverge from reward token balance Resolved. The current token balance is now used by the _startRewardsDistribution method (renamed from _notifyRewardAmount) of the RewardTracker contract. An additional state variable tracks the previously measured token balance to properly account for any balance increases. The distributor role no longer provides a parameter specifying how many reward tokens to allocate, eliminating any possibility of divergence. TOB-SANDCLOCK-3: Miscalculation in beforeWithdraw can leave the vault with less than minimum float Resolved. The arithmetic calculating how much float is missing has been fixed. A new unit test has been added that reproduces the provided exploit scenario to ensure that this change fixes the underlying issue and to prevent regression. TOB-SANDCLOCK-4: Last user in scWETHv2 vault will not be able to withdraw their funds Resolved. The _withdrawToVault method was refactored to limit the amount that the vault withdraws from investment strategies. This allows the last user to withdraw their remaining balance from the vault despite normal float requirements. A new assertion was added to the unit tests to ensure that the last user is able to withdraw their funds. TOB-SANDCLOCK-5: Lido stake rate limit could lead to unexpected reverts Unresolved. The client provided the following context for this finding’s fix status: We acknowledge the issue with a potential lido stake rate limit but we feel it will have a limited effect and not lead to reverts on user deposits since user funds first end up in the vault. Additional comments have been added specifying that the ZeroEx exchange will be the default method for swapping WETH for wstETH and that Lido will be used as a fallback, decreasing the likelihood of hitting the rate limit. For the benefit of future auditors, we recommend adding an inline comment to the lidoSwapWethToWstEth method in the Swapper library to explain the acceptance of the risk and the use of Lido only as a fallback. TOB-SANDCLOCK-6: Chainlink oracles could return stale price data Partially resolved. The third-party LSUDUsdToLUSDEth oracle has been deprecated in favor of the more reliable USD-to-ETH oracle maintained by Liquity. This solution has the side effect of assuming that the value of LUSD is exactly equal to that of USD. However, the

amount of ETH held by the scLiquity vault at any given time is small relative to its exposure to the LUSD price, so the Lindy Labs team is willing to accept this trade-off. We recommend correcting an inline code comment that erroneously refers to the reported price as being in terms of LUSD rather than USD. For the benefit of future auditors and developers, we also recommend adding inline comments explaining the implications of and justifications for this trade-off. The usdcToEthPriceFeed and stEThToEthPriceFeed oracles have not been updated and are still at risk of using stale data without raising any warning.