

Managed Pools

Security assessment by HashEye · prepared for Balancer

HASHEYE AUDITED

PROJECT	Managed Pools
CLIENT	Balancer
CATEGORY	Blockchain
PUBLISHED	October 1, 2022
REPORT ID	research-managed-pools-2022-10-01-1vl0s6

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-managed-pools-2022-10-01-1vl0s6.

Balancer Managed Pool Smart Contracts Security Assessment December 5, 2022 Prepared for: Nicolás Venturo Balancer Labs Prepared by: Nat Chin, Michael Colburn, 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/hashey-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 [hashey](#) on Twitter and explore our public repositories at <https://github.com/hashey-io>. To engage us directly, visit our "Contact" page at <https://www.hashey.io/contact>, or email us at info@hashey.io. HashEye, Inc. 228 Park Ave S #80688 New York, NY 10003 <https://www.hashey.io> info@hashey.io HashEye 1 Balancer Labs Security Assessment PUBLIC

Notices and Remarks Copyright and Distribution © 2022 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 Balancer Labs under the terms of the project statement of work and has been made public at Balancer 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. 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 Balancer Labs Security Assessment PUBLIC

Table of Contents About HashEye 1 Notices and Remarks 2 Table of Contents 3 Executive Summary 4 Project Summary 5 Project Goals 6 Project Targets 7 Project Coverage 8 Codebase Maturity Evaluation 10 Summary of Recommendations 12 Summary of Findings 13 Detailed Findings 14 1. Pool owners may not receive BPT payouts 14 2. Test suite inconsistencies / failures 16 3. Use of outdated libraries 18 A. Vulnerability Categories 19 B. Code Maturity Categories 21 C. Design Recommendations 23 D. Code Quality 25 E. Slither Workarounds 27 F. Incident Response Plan Recommendations 28 G. Token Integration Checklist 30 HashEye 3 Balancer Labs Security Assessment PUBLIC

Executive Summary Engagement Overview Balancer Labs engaged HashEye to review the security of its Managed Pool smart contracts. From October 11 to October 25, 2022, a team of three consultants conducted a security review of the client-provided source code, with six person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report. Project Scope Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the system, including access to the source code and code comments. We performed static and dynamic automated and manual testing of the target system and its codebase. Summary of Findings The audit did not uncover any significant flaws or defects that could impact system confidentiality, integrity, or availability. A summary of the findings is provided below. EXPOSURE ANALYSIS Severity Count High 0 Medium 0 Low 0 Informational 3 Undetermined 0 CATEGORY BREAKDOWN Category Count Patching 1 Testing 1 Undefined Behavior 1 HashEye 4 Balancer Labs Security Assessment PUBLIC

Project Summary Contact Information The following managers were associated with this project: Dan Guido, Account Manager Mary O'Brien, Project Manager dan@hasheye.io mary.obrien@hasheye.io The following engineers were associated with this project: Nat Chin, Consultant Michael Colburn, Consultant natalie.chin@hasheye.io michael.colburn@hasheye.io Guillermo Larregay, Consultant guillermo.larregay@hasheye.io **Project Timeline** The significant events and milestones of the project are listed below.

Date	Event
October 4, 2022	Pre-project kickoff call
October 17, 2022	Status update meeting #1
October 26, 2022	Delivery of report draft
October 26, 2022	Report readout meeting
December 5, 2022	Delivery of final report

 HashEye 5 Balancer Labs Security Assessment PUBLIC

Project Goals The engagement was scoped to provide a security assessment of the Balancer Managed Pool smart contracts. Specifically, we sought to answer the following non-exhaustive list of questions: • Are join, exit, and swap operations properly handled? • Could a BPT swap result in unexpected behavior? • Could the virtual supply be manipulated? • Do the fees match their expected values? • Is it possible to steal funds or lose tokens? HashEye 6 Balancer Labs Security Assessment PUBLIC

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

balancer-v2-monorepo/pkg/pool-weighted/contracts/managed/	Repository	https://github.com/balancer-labs/balancer-v2-monorepo/
Version 20045fc39d83a60ea9910b5bbe58f0251c99b842	Type	Solidity PlatformEVM

 HashEye 7 Balancer Labs 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 and their results include the following: • **ManagedPool** .This contract implements a weighted token pool with preminted BPT. The contract enables pool managers to add / remove tokens to / from a pool and to redefine their weights, which can also fluctuate over time. We manually reviewed the processes of initializing and updating the pool's bookkeeping and defining and executing each type of swap. We also checked the parts of the contract that override the features of the base contracts (e.g., BasePool and BaseGeneralPool), looking for incomplete, unexpected, or incoherent behavior. • **ManagedPoolSettings** .This contract, inherited by the ManagedPool , stores the constants used for data validation throughout the contracts (minimum and maximum token amounts and swap fees). It also stores the addresses of the contracts involved in circuit breaker - based logic(ManagedPoolStorageLib , ManagedPoolAumStorageLib , and ManagedPoolTokenStorageLib)and manages protocol fees. • **Managed Pool wrapper contracts.**The following contracts are wrapper contracts that use the WordCodec library to encode and decode data: ◦ **ManagedPoolAumStorageLib** .This contract saves the assets under management (AUM) fee percentage and the timestamp of the last fee collection. ◦ **ManagedPoolStorageLib** .This contract controls variables related to weight change parameters, swap fees, recovery mode, and allowlists. ◦ **ManagedPoolTokenStorageLib** .This contract implements getters and setters for variables related to token weights and their respective scaling factors. We reviewed the contracts' encoding of values and decoding of parameters. • **CircuitBreakerLib** and **CircuitBreakerStorageLib** .These contracts implement minimum and maximum bounds on BPT prices, causing transactions to revert if a BPT price does not fall within those bounds. We checked that the bounds can be set, retrieved, and adjusted on a per-token basis. We also checked that the circuit breaker trips as expected if a price deviates from the acceptable range. HashEye 8 Balancer Labs Security Assessment PUBLIC

• **WeightedJoinsLib** and **WeightedExitsLib** .These two contracts wrap WeightedMath helper functions with a series of checks. We reviewed the contracts' use of the Math* libraries. **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: • We did not look for opportunities to conduct high-level economic attacks to subvert expected pool invariants. • We did not cover the economics of the fees (i.e., we assumed that the formulas described in the documentation are correct). • We did not look for issues that could arise from the use of other components (e.g., the Vault) or third-party integrations (e.g., the rate provider). • We did not look for complex arbitrage opportunities, particularly those requiring front-running of swap operations or drastic parameter changes. • We did not cover the Math* and WordCodec libraries, which were explicitly designated as out of scope. HashEye 9 Balancer Labs 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 SafeMath is used throughout the codebase where required. The arithmetic operations are analyzed through stateless fuzzing, and there are clear minimum and maximum bounds on those operations and related parameters. **Satisfactory Auditing** The contracts emit enough events to enable off-chain

monitoring of their behavior. However, the Balancer Labs team did not provide an off-chain monitoring plan or an incident response plan. Moderate Authentication / Access Controls The system's access controls are dispersed across multiple contracts, which makes the execution of chained functions fairly difficult to follow. Moderate Complexity Management The Balancer codebase contains many small interconnected functions; however, the flow of the function call stacks is complicated and difficult to follow. Moreover, although the functions are well documented, it is often difficult to determine the expected values of their input and output, as data validation occurs throughout the calls in a chain. Weak Cryptography and Key Management Cryptography and key management were not in scope. Not Applicable HashEye 10 Balancer Labs Security Assessment PUBLIC

Decentralization All of the managed pools are controlled by controllers, which are controlled by a multisignature wallet. Pool managers can change the tokens in a pool, the token weights, and fees; set circuit breakers and allowlists; and enable / disable swaps. Moderate Documentation The codebase contains very detailed inline documentation and per-function invariants. However, given the interconnected nature of the functions, identifying the system-level invariants is somewhat difficult. Moderate Front-Running Resistance We did not check the system for front-running opportunities, as front-running is unlikely to affect the in-scope components. Assessing the system's front-running resistance will require further analysis of the entire system. Further Investigation Required Low-Level Manipulation The low-level manipulation in the codebase is limited to the bit manipulation performed by WordCodec, which was explicitly omitted from the audit's scope. Satisfactory Testing and Verification While Balancer Labs has invested a significant amount of effort in creating numerous tests, the fact that those tests cannot run consistently out of the box indicates that they require more work. Additionally, the team should perform an in-depth review of any test cases that intermittently fail to ensure the failures are not caused by a system bug. Weak HashEye 11 Balancer Labs Security Assessment PUBLIC

Summary of Recommendations The Balancer Managed Pool smart contracts are a work in progress with multiple planned iterations. HashEye recommends that Balancer Labs address the findings detailed in this report and take the following additional steps prior to deployment: • Implement the design recommendations specified in appendix C. • Integrate dependency checks into the CI pipeline to ensure that outdated dependencies are not used in the codebase. • Make the test suite more robust by implementing end-to-end fuzz testing with Echidna and static analysis with Slither. Additionally, always investigate failing unit tests. HashEye 12 Balancer Labs 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 Pool owners may not receive BPT payouts Undefined Behavior Informational 2 Test suite inconsistencies / failures Testing Informational 3 Use of outdated libraries Patching Informational HashEye 13 Balancer Labs Security Assessment PUBLIC

Detailed Findings 1. Pool owners may not receive BPT payouts Severity: Informational Difficulty: High Type: Undefined Behavior Finding ID: TOB-BAL-MNG-1 Target: pkg/pool-weighted/contracts/managed/ManagedPoolSettings.sol Description The _collectAumManagementFees function collects assets under management (AUM) fees and divides them between the protocol and the pool owner. These fees are paid first to the protocol and then to the pool owner; however, after the protocol is paid, there may not be any fees left for the pool owner. As shown in figure 1.1, the _collectAumManagementFees function will immediately return 0 if the BPT amount is too low. If the calculated protocol fee is exactly equivalent to the BPT amount, the managerBPTAmount will be 0. function collectAumManagementFees(uint256 virtualSupply) internal returns (uint256) { [...] // Early return if either: // - AUM fee is disabled. // - no time has passed since the last collection. if (bptAmount == 0) { return 0; } // Split AUM fees between protocol and Pool manager. uint256 protocolBptAmount = bptAmount.mulUp(getProtocolFeePercentageCache(ProtocolFeeType.AUM)); uint256 managerBptAmount = bptAmount.sub(protocolBptAmount); _payProtocolFees(protocolBptAmount); emit ManagementAumFeeCollected(managerBptAmount); _mintPoolTokens(getOwner(), managerBptAmount); return bptAmount; } Figure 1.1: The _collectAumManagementFees function in ManagedPoolSettings.sol HashEye 14 Balancer Labs Security Assessment PUBLIC

Exploit Scenario The Balancer system is initialized with a cached protocol fee percentage of 10. Alice, a privileged user, adds a new token with an initial total supply of 1 to the pool. The management fee calculation determines that the Balancer protocol will receive 10 BPT tokens and that the pool owner will receive 0. Recommendations Short term, if the _collectAumManagementFees function is behaving as expected, clearly document the expectations surrounding the function; if it is not, bind the minimum BPT amount to at least the cached protocol fee percentage. Long term, analyze all incoming arguments and evaluate whether their bounds are safe. HashEye 15 Balancer Labs Security Assessment PUBLIC

2. Test suite inconsistencies / failures Severity:InformationalDifficulty:Medium Type: TestingFinding ID: TOB-BAL-MNG-2 Target: Test suite Description The Balancer protocol tests experience inconsistent behavior when they are built and run. The Balancer tests fail to run on certain machines with a fresh clone of the repository because they are unable to find certain artifacts (figure 2.1). An inability to consistently run tests across machines hinders the development and review processes. BaseManagedPoolFactory constructor arguments 1) "before each" hook: deploy factory & tokens at step "Running shared before each or reverting" for "sets the vault" 0 passing (988ms) 1 failing 1) BaseManagedPoolFactory "before each" hook: deploy factory & tokens at step "Running shared before each or reverting" for "sets the vault": HardhatError: HH700: Artifact for contract "TimelockAuthorizer" not found. at Artifacts._handleWrongArtifactForContractName (~/balancer-v2-monorepo/node_modules/hardhat/src/internal/artifacts.ts:678:11) at Artifacts._getArtifactPathFromFiles (~/balancer-v2-monorepo/node_modules/hardhat/src/internal/artifacts.ts:803:19) at Artifacts._getArtifactPathSync (~/balancer-v2-monorepo/node_modules/hardhat/src/internal/artifacts.ts:561:21) at Artifacts.readArtifactSync (~/balancer-v2-monorepo/node_modules/hardhat/src/internal/artifacts.ts:76:31) at getArtifact (~/balancer-v2-monorepo/pvt/helpers/src/contract.ts:57:20) at Object.deploy (~/balancer-v2-monorepo/pvt/helpers/src/contract.ts:31:20) at processTicksAndRejections (node:internal/process/task_queues:96:5) at Object.deploy (~/balancer-v2-monorepo/pvt/helpers/src/models/vault/VaultDeployer.ts:22:24) at Context.<anonymous> (~/balancer-v2-monorepo/pkg/pool-weighted/test/BaseManagedPoolFactory.test.ts:42:13) at Context.<anonymous> (~/balancer-v2-monorepo/pvt/common/sharedBeforeEach.ts:32:7) Figure 2.1: A terminal output error during the execution of yarntest HashEye 16Balancer Labs Security Assessment PUBLIC

Additionally, when it can be run, one of the Balancer tests intermittently fails (figure 2.2). Test failures such as these should always be investigated to ensure that they are not a sign of a bug in a codebase. 1354 passing (5m) 1 failing 1) CircuitBreakerLib when both bounds are set checks tripped status: AssertionError: expected true to be false + expected - actual -true +false at Context.<anonymous> (test/CircuitBreakerLib.test.ts:127:40) at runMicrotasks (<anonymous>) at processTicksAndRejections (internal/process/task_queues.js:95:5) at runNextTicks (internal/process/task_queues.js:64:3) at listOnTimeout (internal/timers.js:526:9) at processTimers (internal/timers.js:500:7) Figure 2.2: Terminal output indicating a failing test Exploit Scenario Alice, a Balancer developer, refactors a portion of the code as part of an effort to simplify it. In doing so, she accidentally introduces unexpected system behavior that could cause a pool to enter an invalid state. Because the refactored code cannot be tested, the behavior is not detected before the system is deployed. Recommendations Short term, investigate all failing test cases to ensure that the code is behaving as expected. Long term, implement comprehensive testing across the codebase to ensure that the code behaves as expected under all conditions. HashEye 17Balancer Labs Security Assessment PUBLIC

3. Use of outdated libraries Severity:InformationalDifficulty:Low Type: PatchingFinding ID: TOB-BAL-MNG-3 Target: pkg/pool-weighted/package.json Description HashEye used npm-check-updates to detect outdateddependencies in the codebase. This check found that 25 of the packages referenced by the package.json file are outdated. The following table lists only the dependencies that have not been updated to the latest major version. DependencyVersion currently in useLatest version available @types/mocha 8.0.310.0.0 @types/node 14.6.018.11.4 @typescript-eslint/eslint-plugin 4.1.15.41.0 @typescript-eslint/parser 4.1.15.41.0 eslint 7.9.08.26.0 eslint-plugin-prettier 3.1.44.2.1 mocha 8.2.110.1.0 ts-node 8.10.210.9.1 Recommendations Short term, update build process dependencies to the latest versions wherever possible. Use tools such as retire.js , nodeaudit , and yarnaudit to confirm that no vulnerable dependencies remain in the codebase. Long term, implement dependency checks as part of the CI / CD pipeline. Do not allow builds to continue with outdated dependencies. HashEye 18Balancer Labs Security Assessment PUBLIC

A. Vulnerability Categories The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document. Vulnerability Categories CategoryDescription Access ControlsInsufficient authorization or assessment of rights Auditing and LoggingInsufficient auditing of actions or logging of problems AuthenticationImproper identification of users ConfigurationMisconfigured servers, devices, or software components CryptographyA breach of system confidentiality or integrity Data ExposureExposure of sensitive information Data ValidationImproper reliance on the structure or values of data Denial of ServiceA system failure with an availability impact Error ReportingInsecure or insufficient reporting of error conditions PatchingUse of an outdated software package or library Session ManagementImproper identification of authenticated users TestingInsufficient test methodology or test coverage TimingRace conditions or other order-of-operations flaws Undefined BehaviorUndefined behavior triggered within the system HashEye 19Balancer Labs Security Assessment PUBLIC

Severity Levels SeverityDescription InformationalThe issue does not pose an immediate risk but is relevant to security best practices. UndeterminedThe extent of the risk was not determined during this engagement. LowThe risk is small or is not one the client has indicated is important. MediumUser information is at risk; exploitation could pose reputational, legal, or moderate financial risks. HighThe flaw could affect numerous users and have serious reputational, legal, or financial implications. Difficulty Levels DifficultyDescription UndeterminedThe difficulty of exploitation was not determined during this engagement. LowThe flaw is well known; public tools for its exploitation exist or can be scripted. MediumAn attacker must write an exploit or will need in-depth knowledge of the system. HighAn 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 20Balancer Labs 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 CategoryDescription ArithmeticThe proper use of mathematical operations and semantics AuditingThe 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 DecentralizationThe presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades DocumentationThe 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 21Balancer Labs Security Assessment PUBLIC

Rating Criteria RatingDescription StrongNo issues were found, and the system exceeds industry standards. SatisfactoryMinor issues were found, but the system is compliant with best practices. ModerateSome issues that may affect system safety were found. WeakMany issues that affect system safety were found. MissingA required component is missing, significantly affecting system safety. Not ApplicableThe category is not applicable to this review. Not ConsideredThe category was not considered in this review. Further Investigation Required Further investigation is required to reach a meaningful conclusion. HashEye 22Balancer Labs Security Assessment PUBLIC

C. Design Recommendations Because of the complexity of the Balancer codebase, it contains numerous in-code comments that describe function-specific expectations. This appendix provides recommendations for simplifying the system's design and creating high-level documentation. Data Manipulation • Standardize the code responsible for input validation and transformation by either moving it closer to the relevant functions or implementing it as a general pre-processing step. Many functions take inputs with several attributes that must be validated or transformed. Currently, these attributes are checked at different points of the input pipeline. For example, a value may be multiplied by a scaling factor and then undergo several checks (e.g., a check ensuring the value is a positive number) all before the relevant function is reached. As a result of this data manipulation pattern, it is difficult to discern the checks or transformations that an input has undergone before reaching a particular point in the codebase. High-Level Architecture Documentation • Create high-level documentation outlining the end-to-end architecture of the system. This documentation should include the following:

- Examples of walk-through scenarios for various user operations. These walk-through scenarios should map out all function calls involved in different pool operations, detail the underlying assumptions related to the various interactions, and provide a visual representation of the differences between operations (e.g., exit versus exitSwap operations).
- A flow diagram highlighting the points at which BPT token balances are considered. In most cases, the ManagedPool contract's `_getPoolTokens()` function automatically drops the BPT balance from the array of pool token balances it returns; however, certain corner cases require the code to explicitly drop the first element of the array. A diagram outlining the functions and flows in which that is required could help prevent the accidental inclusion of a BPT balance in the balance of a pool.
- Details on how the codebase has evolved over time and the design decisions that were made at each stage. This information would help contextualize the complexity of the codebase and could help highlight system-wide invariants.

HashEye 23Balancer Labs Security Assessment PUBLIC

Inheritance • Limit function flows to either a base contract or a derived contract. Balancer pool operations are defined in base contracts, and the derived contracts often override some, but not all, of those operations. This results in a pattern in which function calls bounce between the base and derived contract, making the code difficult to follow. These calls include those to the WeightedPoolProtocolFees contract's `inRecoveryMode()`, `_setRecoveryMode(bool)`, and `_getAuthorizer()`

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

- Reduce the amount of code duplication. This will increase the code's maintainability and prevent the introduction of errors if the code needs to be modified. ◦ Examples include the code duplicated in the `_onJoinSwap`, `_onExitSwap`, and `_onTokenSwap` functions, all of which retrieve token information and swap fee data.
- Avoid the use of ternary operators. This will enhance the code's clarity and readability.
- Flatten the contract hierarchy. Reducing the need to move between different functions in different files within a single operation will improve the code's readability. Moreover, complex inheritance trees can lead to unintended function shadowing and issues related to the diamond problem. This is a problem associated with multiple inheritance and results in ambiguity between the functions being executed. In some cases, libraries can be used instead of inheritance. ◦ For example, `ManagedPoolSettings._getAuthorizer()` collides in the inherited contracts `BasePoolAuthorization` and `BasePool`, and the `BasePool` implementation of the function is chosen.
- Pre-cache the bounds of array-traversing loops. If the length of an array traversed in a loop is not modified, the array length can be cached in memory to produce a minor gas optimization.
- Avoid using a function's side effects to perform actions unrelated to its name. A function's name should be indicative of the task it performs. Calling a function with counterintuitive arguments to trigger a side effect makes the code more difficult to understand. This occurs in the following code paths: ◦ `ManagedPoolSettings.sol#L159-L172` ◦ `ManagedPoolSettings.sol#L801` ◦ `ManagedPoolSettings.sol#L900`

HashEye 25Balancer Labs Security Assessment PUBLIC

- Check the code comments for errors and update the comments when the code changes. The code is thoroughly documented in comments; therefore, it is particularly important to keep those comments up to date and free of errors. ◦ An error appears in `ManagedPool.sol#L283-L284`, in which `GIVEN_IN` should be `GIVEN_OUT`.

HashEye 26Balancer Labs Security Assessment PUBLIC

E. Slither Workarounds Running Slither over the pool-weighted directory of the Balancer repository requires workarounds. Firstly, it is necessary to simplify the `hardhat.config.ts` file by specifying only Solidity version 0.7.6: % cat hardhat.config.ts module.exports = { solidity: { version: "0.7.6", }, } Figure E.1: The `hardhat.config.ts` file Then, it is necessary to use symlinks to relink the dependencies in the codebase, which enables Slither to find those dependencies: `cd node_modules mkdir @balancer-labs cd @balancer-labs ln -s ../../../../interfaces/ v2-interfaces ln -s ../../../../pool-utils v2-pool-utils ln -s ../../../../solidity-utils v2-solidity-utils` Figure E.2: Symlinks used to facilitate Slither testing

HashEye 27Balancer Labs Security Assessment PUBLIC

F. Incident Response Plan Recommendations This section provides recommendations on formulating an incident response plan.

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

HashEye 28Balancer Labs Security Assessment PUBLIC

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

HashEye 29Balancer Labs Security Assessment PUBLIC

G. Token Integration Checklist The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. For an up-to-date version of the checklist, see [crytic/building-secure-contracts](#) . For convenience, all Slither utilities can be run directly on a token address, such as the following: `slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken --erc erc20 slither-check-erc 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d KittyCore --erc erc721` To follow this checklist, use the below output from Slither for the token: `slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER] slither [target] --print human-summary slither [target] --print contract-summary slither-prop . --contract ContractName # requires configuration, and use of Echidna and Manticore`

General Considerations

- ☐ The contract has a security review. Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.
- ☐ You have contacted the developers. You may need to alert their team to an incident. Look for appropriate contacts on [blockchain-security-contacts](#) .
- ☐ They have a security mailing list for critical announcements. Their team should advise users (like you!) when critical issues are found or when upgrades occur.

Contract Composition

- ☐ The contract avoids unnecessary complexity. The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's human-summary printer to identify complex code.
- ☐ The contract uses SafeMath . Contracts that do not use SafeMath require a higher standard of review. Inspect the contract by hand for SafeMath usage.
- ☐ The contract has only a few non-token-related functions. Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's contract-summary printer to broadly review the code used in the contract.

HashEye 30 Balancer Labs Security Assessment PUBLIC

- ☐ The token has only one address. Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` may not reflect the actual balance).

Owner Privileges

- ☐ The token is not upgradeable. Upgradeable contracts may change their rules over time. Use Slither's human-summary printer to determine whether the contract is upgradeable.
- ☐ The owner has limited minting capabilities. Malicious or compromised owners can abuse minting capabilities. Use Slither's human-summary printer to review minting capabilities, and consider manually reviewing the code.
- ☐ The token is not pausable. Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.
- ☐ The owner cannot blacklist the contract. Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.
- ☐ The team behind the token is known and can be held responsible for abuse. Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

ERC20 Tokens

ERC20 Conformity Checks

Slither includes a utility, `slither-check-erc` , that reviews the conformance of a token to many related ERC standards. Use `slither-check-erc` to review the following:

- ☐ `Transfer` and `transferFrom` return a boolean. Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.
- ☐ The `name` , `decimals` , and `symbol` functions are present if used. These functions are optional in the ERC20 standard and may not be present.
- ☐ `Decimals` returns a `uint8` . Several tokens incorrectly return a `uint256` . In such cases, ensure that the value returned is below 255.
- ☐ The token mitigates the known ERC20 race condition. The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens. Slither includes a utility, `slither-prop` , that generates unit tests and security properties that can discover many common ERC flaws. Use `slither-prop` to review the following:

HashEye 31 Balancer Labs Security Assessment PUBLIC

- ☐ The contract passes all unit tests and security properties from `slither-prop` . Run the generated unit tests and then check the properties with Echidna and Manticore.

Risks of ERC20 Extensions

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ☐ The token is not an ERC777 token and has no external function call in `transfer` or `transferFrom` . External calls in the `transfer` functions can lead to reentrancies.
- ☐ `Transfer` and `transferFrom` should not take a fee. Deflationary tokens can lead to unexpected behavior.
- ☐ Potential interest earned from the token is taken into account. Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ☐ The supply is owned by more than a few users. If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ☐ The total supply is sufficient. Tokens with a low total supply can be easily manipulated.
- ☐ The tokens are located in more than a few exchanges. If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ☐ Users understand the risks associated with a large amount of funds or flash loans. Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ☐ The token does not allow flash minting. Flash minting can lead to substantial swings in the balance and the total supply,

ERC721 Tokens ERC721 Conformity Checks The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions: Transfers of tokens to the 0x0 address revert. Several tokens allow transfers to 0x0 and consider tokens transferred to that address to have been burned; however, the ERC721 standard requires that such transfers revert. safeTransferFrom functions are implemented with the correct signature. Several token contracts do not implement these functions. A transfer of NFTs to one of those contracts can result in a loss of assets. The name, decimals, and symbol functions are present if used. These functions are optional in the ERC721 standard and may not be present. If it is used, decimals returns a uint8(0). Other values are invalid. The name and symbol functions can return an empty string. This behavior is allowed by the standard. The ownerOf function reverts if the tokenId is invalid or is set to a token that has already been burned. The function cannot return 0x0. This behavior is required by the standard, but it is not always properly implemented. A transfer of an NFT clears its approvals. This is required by the standard. The token ID of an NFT cannot be changed during its lifetime. This is required by the standard. Common Risks of the ERC721 Standard To mitigate the risks associated with ERC721 contracts, conduct a manual review of the following conditions: The onERC721Received callback is taken into account. External calls in the transfer functions can lead to reentrancies, especially when the callback is not explicit (e.g., in safeMint calls). HashEye 33Balancer Labs Security Assessment PUBLIC

When an NFT is minted, it is safely transferred to a smart contract. If there is a minting function, it should behave similarly to safeTransferFrom and properly handle the minting of new tokens to a smart contract. This will prevent a loss of assets. The burning of a token clears its approvals. If there is a burning function, it should clear the token's previous approvals. HashEye 34Balancer Labs Security Assessment PUBLIC