HashEye

// SMART CONTRACT SECURITY AUDIT

Immutable

Security assessment by HashEye · prepared for Immutable

HASHEYE AUDITED

PROJECT	Immutable
CLIENT	Immutable
CATEGORY	Blockchain
PUBLISHED	August 1, 2023
REPORT ID	research-immutable-2023-08-01-13rg68

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-immutable-2023-08-01-13rg68.

Immutable Smart Contracts Security Assessment October 4, 2023 Prepared for: Ryan Teoh Immutable Prepared by: Michael Colburn, Elvis Skoždopolj, and Priyanka Bose

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 Immutable Smart Contracts 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 Immutable Smart Contracts Security Assessment PUBLIC

Table of Contents About HashEye 1 Notices and Remarks 2 Table of Contents 3 Project Summary 4 Executive Summary 5 Project Goals 7 Project Targets 8 Project Coverage 9 Codebase Maturity Evaluation 12 Summary of Findings 14 Detailed Findings 15 1. Initialization functions vulnerable to front-running 15 2. Lack of lower and upper bounds for system parameters 17 3. RootERC20Predicate is incompatible with nonstandard ERC-20 tokens 20 4. Lack of event generation 22 5. Withdrawal queue can be forcibly activated to hinder bridge operation 24 A. Vulnerability Categories 26 B. Code Maturity Categories 28 C. Code Quality Recommendations 30 D. Testing Improvement Recommendations 31 E. Fix Review Results 38 Detailed Fix Review Results 39 F. Fix Review Status Categories 40 HashEye 3 Immutable Smart Contracts Security Assessment PUBLIC

Project Summary Contact Information The following managers were associated with this project: Dan Guido , Account Manager Anne Marie Barry , Project Manager dan@hasheye.io annemarie.barry@hasheye.io The following engineers were associated with this project: Michael Colburn , Consultant Elvis Skoždopolj , Consultant michael.colburn@hasheye.io elvis.skozdopolj@hasheye.io Priyanka Bose , Consultant priyanka.bose@hasheye.io Project Timeline The significant events and milestones of the project are listed below. Date Event August 17, 2023 Pre-project kickoff call August 24, 2023 Status update meeting #1 August 30, 2023 Delivery of report draft August 31, 2023 Report readout meeting September 26, 2023 Delivery of comprehensive report October 4, 2023 Delivery of comprehensive report with fix review HashEye 4 Immutable Smart Contracts Security Assessment PUBLIC

Executive Summary Engagement Overview Immutable engaged HashEye to review the security of its extensions of the Polygon ERC-20 bridge contracts, extensions to the OpenSea Seaport contracts that implement SIP-7, guarded multicaller contract, and EIP-1559 base fee contract. A team of three consultants conducted the review from August 21 to August 30, 2023, for a total of three engineer-weeks of eff ort. Our testing eff orts focused primarily on identifying any instances that could result in lost or stuck funds or the ability to bypass the bridge's withdrawal queue or Immutable Seaport's SIP-7 mechanism. With full access to source code and documentation, we performed static and dynamic testing of the codebases, using automated and manual processes. Observations and Impact The codebases are built on well-known protocols and the modifications and extensions are logically separated into their own contracts, so changes to contracts in the underlying codebases are minimal. Each of the codebases was accompanied by a test plan and a thorough threat model identifying the security considerations and mitigations, which facilitated understanding of the codebases and identification of potential areas of concern. The two primary patterns underlying the issues in this report pertain to potential front-running risks in the bridge and edge cases that may have been known but not documented. Recommendations Based on the codebase maturity evaluation and findings identified during the security review, HashEye recommends that Immutable 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. • Develop deployment scripts for the bridge contracts. Include scenarios both for deployment and upgrades, taking care to account for TOB-IMM-1 . One motivating factor for the withdrawal queue is to provide time to respond in the event of an incident, with bridge upgrades being the most likely avenue for mitigation. However, the absence of deployment scripts or tests of the upgrade process will make responding more difficult and error-prone. • Continue to develop the system documentation. The repositories contain fairly comprehensive documentation of the systems and security considerations. Ensure this documentation carries over to longer-term user-facing documentation as well. HashEye 5 Immutable Smart Contracts Security Assessment PUBLIC

• Continue to improve the testing suites. Using more advanced testing techniques like integration, fork, fuzz, and mutation testing can provide much more thorough coverage of the codebases and prevent regressions. The following tables provide the number of findings by severity and category. EXPOSURE ANALYSIS Severity Count High 0 Medium 1 Low 1 Informational 3 Undetermined 0 CATEGORY BREAKDOWN Category Count Auditing and Logging 1 Data Validation 2 Denial of Service 1 Timing 1 HashEye 6 Immutable Smart Contracts Security Assessment PUBLIC

Project Goals The engagement was scoped to provide a security assessment of the in-scope Immutable smart contracts. Specifically, we sought to answer the following non-exhaustive list of questions: • Does the bridge's flow metering mechanism contain arithmetic errors? • Can users bypass the bridge's withdrawal queue? • Can items in the withdrawal queue be duplicated or finalized multiple times? • Are any functions vulnerable to front-running attacks? • Are functions vulnerable to reentrancy? • Can calling syncState directly result in bypassing the intended withdrawal or deposit mechanisms? • Do the function selectors of the functions called via cross-chain messages collide with any other function selector in the system? • Can user assets be frozen or stolen by other users? • Does the ImmutableSignedZone contract correctly implement SIP-7? • Can a user bypass the signature requirement? HashEye 7 Immutable Smart Contracts Security Assessment PUBLIC

Project Targets The engagement involved a review and testing of the targets listed below. Bridge flow rate contracts Repository https://github.com/immutable/poly-core-contracts Version b5065149b7ff87d5123616d9e4df7312ef0eb316 Type Solidity Platform EVM Immutable Signed Zone (SIP-7) contracts Repository https://github.com/immutable/immutable-seaport Version 6cb6f2352d1e091d08552d862a53acfcf83ed49b Type Solidity Platform EVM Guarded multicaller contract Repository https://github.com/immutable/multicaller-contracts Version 605c3e2956c914a77181baba85dfca9bde8c3829 Type Solidity Platform EVM Base fee contract Repository https://github.com/immutable/poly-core-contracts Version be1599dda478bcd6cf4e7472e3111189cf47a6f6 Type Solidity Platform EVM HashEye 8 Immutable Smart Contracts Security Assessment PUBLIC

Project Coverage This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. The review was scoped to include only the extensions and changes made to each codebase and their interactions with that codebase, but not the entire codebase itself. • ERC-20 bridge: poly-core-contracts/contracts/root/flowrate/RootERC20PredicateFlowRate.sol and the contracts it inherits • Immutable Seaport: the contracts in the immutable-seaport repository • Guarded multicaller: multicaller-contracts/contracts/GuardedMulticaller.sol , which is a standalone contract • Base Fee: poly-core-contracts/contracts/child/EIP1559BurnLogic.sol and poly-core-contracts/contracts/child/ProxyAdminInherited.sol Our approaches and results included the following: • Bridge flow rate contracts. This codebase is a fork of Polygon's core-contracts repository. The ERC-20 bridge contracts have been extended to implement support for transferring native ether, pausing deposits and withdrawals, and adding a withdrawal queue that can act as a

timelock to delay withdrawals in certain scenarios, such as individual large value transfers or sufficient aggregate transfers in a particular window of time. ○ We reviewed the flow metering mechanism for arithmetic errors and race conditions. We explored how transaction ordering or front-running could impact the bridge, which resulted in the discovery of TOB-IMM-5 . ○ We reviewed the way withdrawals are queued and finalized to determine whether the queue can be bypassed, whether queue items can be duplicated, or whether queue items can be finalized multiple times. ○ We reviewed the deposit and withdrawal flow of the contracts for reentrancy concerns, specifically related to the use of nonstandard or malicious ERC-20 tokens and the use of low-level calls for value transfers. ○ We reviewed common concerns related to upgradeability that could lead to issues as part of an upgrade. This included checking the storage layout and HashEye 9 Immutable Smart Contracts Security Assessment PUBLIC

function selectors for collisions and checking the inheritance structure of the contracts. ○ Due to the bridge contracts allowing arbitrary tokens to be bridged, we checked for unusual token interactions (e.g., fee on transfer tokens, rebasing tokens, ERC-777 tokens, etc.), and whether these interactions can allow users to reenter functions or otherwise cause funds to be lost or trapped. ○ We reviewed the codebase against recent bridge exploits to determine if these exploits are applicable to the codebase. ○ We reviewed the codebase for ways that funds can become lost or trapped. ○ We reviewed the access controls placed on the various functions to ensure that there are no avenues for access control bypass or privilege escalation. ● Immutable Signed Zone (SIP-7) contracts. These contracts are built on top of OpenSea's Seaport 1.5 NFT marketplace. They extend the codebase by implementing SIP-7 , which specifies a common method of signing orders. These signatures will be issued by Immutable and validated on-chain to ensure that any orders that are fulfilled meet certain requirements. ○ We reviewed the way the system manages signers and processes signatures to determine whether the signatures are properly validated and whether the signature schema is designed to prevent common issues such as signature malleability or replay attacks. ○ We reviewed the ImmutableSignedZone contract to determine whether it properly conforms to the SIP-7 standard and whether any of the (sub)standards can lead to undefi ned behavior. ■ We reviewed the differences between the Seaport zone reference implementations and the ImmutableSignedZone implementation. ● Guarded multicaller contract. This contract allows authenticated users to batch multiple calls to allowlisted functions in a single transaction. ○ We reviewed the access controls placed on the various functions to ensure that there are no avenues for access control bypass or privilege escalation. ○ We reviewed the signature validation for common issues such as signature malleability or replay attacks. HashEye 10 Immutable Smart Contracts Security Assessment PUBLIC

○ We reviewed the execute function to determine whether it is vulnerable to reentrancy attacks or similar issues that could result from bundling multiple calls in one transaction. ● Base fee contract. The EIP1559BurnLogic contract is the initial logic contract for managing fees burned as part of the EIP-1559 gas fee market. This contract will be deployed behind a TransparentUpgradeableProxy that is administered by a slightly modified ProxyAdmin contract to allow it to be initialized properly during the genesis of a Polygon Edge-based chain. ○ We reviewed this contract and the proposed configuration to identify any concerns with this setup or the minor modifications to the original contracts required to achieve it. 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 explore advanced testing such as fuzzing due to the complex setup needed to achieve meaningful results in each codebase in such a short time. ● Any contracts outside those mentioned in the above section were out of scope. They may have been referenced to aid in understanding the system but were not a priority for this review. HashEye 11 Immutable Smart Contracts Security Assessment PUBLIC

Codebase Maturity Evaluation HashEye uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Defi ciencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs. Category Summary Result Arithmetic The contracts in scope use compiler versions that include arithmetic checks by default ($\geq$ 0.8.0) and do not use unchecked blocks. Mathematical expressions are simple and used sparingly, primarily for the flow rate calculation logic. Satisfactory Auditing Though we did identify two functions that would benefi t from additional event emissions (TOB-IMM-4), most other contracts have adequate event coverage for state- modifying functions. Satisfactory Authentication / Access Controls Most of the in-scope contracts contain role-based access controls. The roles are clearly documented in the provided documentation and threat models, and the division of responsibilities between the roles is sufficiently robust. It would be benefi cial to incorporate the role defi nitions into the user-facing documentation once such documentation is created. Satisfactory Complexity Management The contracts' compositions are simple and can be easily analyzed. Changes

made to the underlying codebases are minimal and clearly documented. The complexity of individual contracts is managed very well by separating them into coherent and smaller individual components. Satisfactory Decentralization This category was not applicable for this review. Not Applicable Documentation The codebases contain detailed audit documentation and threat models, including written descriptions and specifications; flow chart diagrams with multiple examples; a list of functions that includes their selectors, Satisfactory HashEye 12 Immutable Smart Contracts Security Assessment PUBLIC

visibility, and access controls; and various details on threat actors, capabilities, and mitigations. The contracts contain thorough NatSpec and inline comments. Front-Running Resistance The codebases do not contain any mitigations related to front-running or transaction reordering. The initialization functions of the ERC-20 bridge contracts are vulnerable to front-running attacks (TOB-IMM-1), and user withdrawals could be front-run to force them into the withdrawal queue, delaying their withdrawal by 24 hours. However, the transaction reordering attacks cannot directly extract value or permanently lock out a user from withdrawing. Moderate Low-Level Manipulation Both the bridge and multicaller codebases use low-level calls to forward arbitrary function calls from their users and perform adequate checks on the return values of those calls. In the Immutable Seaport codebase, the contracts were written explicitly to avoid assembly use (in contrast to Seaport itself, which heavily uses assembly), with a minimal increase in gas cost. Satisfactory Testing and Verification The codebases contain testing strategies that use the branching tree technique to defi ne a test specification. The threat models clearly defi ne various exploit scenarios and threat actors. However, the testing coverage could be improved, especially through other testing approaches like fuzzing, fork testing, and integration testing, which would improve the robustness of the testing suite. Additionally, although the contracts are meant to be upgradeable, the test setups do not use proxies but instead test the logic contracts directly. Moderate HashEye 13 Immutable Smart Contracts Security Assessment PUBLIC

Summary of Findings The table below summarizes the findings of the review, including type and severity details. ID Title Type Severity 1 Initialization functions vulnerable to front-running Timing Informational 2 Lack of lower and upper bounds for system parameters Data Validation Informational 3 RootERC20Predicate is incompatible with nonstandard ERC-20 tokens Data Validation Low 4 Lack of event generation Auditing and Logging Informational 5 Withdrawal queue can be forcibly activated to hinder bridge operation Denial of Service Medium HashEye 14 Immutable Smart Contracts Security Assessment PUBLIC

Detailed Findings 1. Initialization functions vulnerable to front-running Severity: Informational Difficulty: High Type: Timing Finding ID: TOB-IMM-1 Target: Throughout Description Several implementation contracts have initialization functions that can be front-run, which would allow an attacker to incorrectly initialize the contracts. Due to the use of the delegatecall proxy pattern, the RootERC20Predicate and RootERC20PredicateFlowRate contracts (as well as other upgradeable contracts that are not in scope) cannot be initialized with a constructor, so they have initialize functions: function initialize (address newStateSender , address newExitHelper , address newChildERC20Predicate , address newChildTokenTemplate , address nativeTokenRootAddress) external virtual initializer { __RootERC20Predicate_init(newStateSender, newExitHelper, newChildERC20Predicate, newChildTokenTemplate, nativeTokenRootAddress); } Figure 1.1: Front-runnable initialize function (RootERC20Predicate.sol) function initialize (address superAdmin , address pauseAdmin , address unpauseAdmin , address rateAdmin , address newStateSender , address newExitHelper , address newChildERC20Predicate , HashEye 15 Immutable Smart Contracts Security Assessment PUBLIC

address newChildTokenTemplate , address nativeTokenRootAddress) external initializer { __RootERC20Predicate_init(newStateSender, newExitHelper, newChildERC20Predicate, newChildTokenTemplate, nativeTokenRootAddress); __Pausable_init(); __FlowRateWithdrawalQueue_init(); _setupRole(DEFAULT_ADMIN_ROLE, superAdmin); _setupRole(PAUSER_ADMIN_ROLE, pauseAdmin); _setupRole(UNPAUSER_ADMIN_ROLE, unpauseAdmin); _setupRole(RATE_CONTROL_ROLE, rateAdmin); } Figure 1.2: Front-runnable initialize function (RootERC20PredicateFlowRate.sol) An attacker could front-run these functions and initialize the contracts with malicious values. The documentation provided by the Immutable team indicates that they are aware of this issue and how to mitigate it upon deployment of the proxy or when upgrading the implementation. However, there do not appear to be any deployment scripts to demonstrate that this will be correctly done in practice, and the codebase’s tests do not cover upgradeability. Exploit Scenario Bob deploys the RootERC20Predicate contract. Eve deploys an upgradeable version of the ExitHelper contract and front-runs the RootERC20Predicate initialization, passing her contract’s address as the exitHelper argument. Due to a lack of post-deployment checks, this issue goes unnoticed and the protocol functions as intended for some time, drawing in a large amount of deposits. Eve then upgrades the ExitHelper contract to allow her to arbitrarily call the

onL2StateReceive function of the RootERC20Predicate contract, draining all assets from the bridge. Recommendations Short term, either use a factory pattern that will prevent front-running the initialization, or ensure that the deployment scripts have robust protections against front-running attacks. Long term, carefully review the Solidity documentation , especially the “Warnings” section, and the pitfalls of using the delegatecall proxy pattern. HashEye 16 Immutable Smart Contracts Security Assessment PUBLIC

2. Lack of lower and upper bounds for system parameters Severity: Informational Difficulty: High Type: Data Validation Finding ID: TOB-IMM-2 Target: contracts/root/flowrate/RootERC20PredicateFlowRate.sol Description The lack of lower and upper bound checks when setting important system parameters could lead to a temporary denial of service, allow users to complete their withdrawals prematurely, or otherwise hinder the expected performance of the system. The setWithdrawalDelay function of the RootERC20PredicateFlowRate contract can be used by the rate control role to set the amount of time that a user needs to wait before they can withdraw their assets from the root chain of the bridge. // RootERC20PredicateFlowRate.sol function setWithdrawalDelay (uint256 delay) external onlyRole(RATE_CONTROL_ROLE) { _setWithdrawalDelay(delay); } // FlowRateWithdrawalQueue.sol function _setWithdrawalDelay (uint256 delay) internal { withdrawalDelay = delay; emit WithdrawalDelayUpdated(delay); } Figure 2.1: The setter functions for the withdrawalDelay state variable (RootERC20PredicateFlowRate.sol and FlowRateWithdrawalQueue.sol) The withdrawalDelay variable value is applied to all currently pending withdrawals in the system, as shown in the highlighted lines of figure 2.2. function _processWithdrawal (address receiver , uint256 index) internal returns (address withdrawer , address token , uint256 amount) { // ... // Note: Add the withdrawal delay here, and not when enqueuing to allow changes // to withdrawal delay to have effect on in progress withdrawals. uint256 withdrawalTime = withdrawal.timestamp + withdrawalDelay; // slither-disable-next-line timestamp if (block.timestamp < withdrawalTime) { HashEye 17 Immutable Smart Contracts Security Assessment PUBLIC

// solhint-disable-next-line not-rely-on-time revert WithdrawalRequestTooEarly(block.timestamp , withdrawalTime); } // ... } Figure 2.2: The function completes a withdrawal from the withdrawal queue if the withdrawalTime has passed. (FlowRateWithdrawalQueue.sol) However, the setWithdrawalDelay function does not contain any validation on the delay input parameter. If the input parameter is set to zero, users can skip the withdrawal queue and immediately withdraw their assets. Conversely, if this variable is set to a very high value, it could prevent users from withdrawing their assets for as long as this variable is not updated. The setRateControlThreshold allows the rate control role to set important token parameters that are used to limit the amount of tokens that can be withdrawn at once, or in a certain time period, in order to mitigate the risk of a large amount of tokens being bridged after an exploit. // RootERC20PredicateFlowRate.sol function setRateControlThreshold (address token , uint256 capacity , uint256 refillRate , uint256 largeTransferThreshold) external onlyRole(RATE_CONTROL_ROLE) { _setFlowRateThreshold(token, capacity, refillRate); largeTransferThresholds[token] = largeTransferThreshold; } // FlowRateDetection.sol function _setFlowRateThreshold (address token , uint256 capacity , uint256 refillRate) internal { if (token == address (0)) { revert InvalidToken(); } if (capacity == 0) { revert InvalidCapacity(); } if (refillRate == 0) { revert InvalidRefillRate(); } Bucket storage bucket = flowRateBuckets[token]; if (bucket.capacity == 0) { bucket.depth = capacity; } HashEye 18 Immutable Smart Contracts Security Assessment PUBLIC

bucket.capacity = capacity; bucket.refillRate = refillRate; } Figure 2.3: The function sets the system parameters to limit withdrawals of a specific token. (RootERC20PredicateFlowRate.sol and FlowRateDetection.sol) However, because the _setFlowRateThreshold function of the FlowRateDetection contract is missing upper bounds on the input parameters, these values could be set to an incorrect or very high value. This could potentially allow users to withdraw large amounts of tokens at once, without triggering the withdrawal queue. Exploit Scenario Alice attempts to update the withdrawalDelay state variable from 24 to 48 hours. However, she mistakenly sets the variable to 0 . Eve uses this setting to skip the withdrawal queue and immediately withdraws her assets. Recommendations Short term, determine reasonable lower and upper bounds for the setWithdrawalDelay and setRateControlThreshold functions, and add the necessary validation to those functions. Long term, carefully document which system parameters are configurable and ensure they have adequate upper and lower bound checks. HashEye 19 Immutable Smart Contracts Security Assessment PUBLIC

3. RootERC20Predicate is incompatible with nonstandard ERC-20 tokens Severity: Low Difficulty: Low Type: Data Validation Finding ID: TOB-IMM-3 Target: contracts/root/RootERC20Predicate.sol Description The deposit and depositTo functions of the RootERC20Predicate contract are incompatible with nonstandard ERC-20 tokens, such as tokens that take a fee on transfer. The RootERC20Predicate contract allows users to deposit arbitrary tokens into the root chain of the bridge and mint the

corresponding token on the child chain of the bridge. Users can deposit their tokens by approving the bridge for the required amount and then calling the deposit or depositTo function of the contract. These functions will call the internal _depositERC20 function, which will perform a check to ensure the token balance of the contract is exactly equal to the balance of the contract before the deposit, plus the amount of tokens that are being deposited. function _depositERC20 (IERC20Metadata rootToken, address receiver , uint256 amount) private { uint256 expectedBalance = rootToken.balanceOf(address (this)) + amount; _deposit(rootToken, receiver, amount); // invariant check to ensure that the root token balance has increased by the amount deposited // slither-disable-next-line incorrect-equality require ((rootToken.balanceOf(address (this)) == expectedBalance), "RootERC20Predicate: UNEXPECTED_BALANCE"); } Figure 3.1: Internal function used to deposit ERC-20 tokens to the bridge (RootERC20Predicate.sol) However, some nonstandard ERC-20 tokens will take a percentage of the transferred amount as a fee. Due to this, the require statement highlighted in figure 3.1 will always fail, preventing users from depositing such tokens. Recommendations Short term, clearly document that nonstandard ERC-20 tokens are not supported by the protocol. If the team determines that they want to support nonstandard ERC-20 implementations, additional logic should be added into the _deposit function to determine the actual token amount received by the contract. In this case, reentrancy HashEye 20 Immutable Smart Contracts Security Assessment PUBLIC

protection may be needed to mitigate the risks of ERC-777 and similar tokens that implement callbacks whenever tokens are sent or received. Long term, be aware of the idiosyncrasies of ERC-20 implementations. This standard has a history of misuses and issues. References • Incident with non-standard ERC20 defl ationary tokens • d-xo/weird-erc20 HashEye 21 Immutable Smart Contracts Security Assessment PUBLIC

4. Lack of event generation Severity: Informational Difficulty: Low Type: Auditing and Logging Finding ID: TOB-IMM-4 Target: RootERC20PredicateFlowRate.sol , ImmutableSeaport.sol Description Multiple critical operations do not emit events. This creates uncertainty among users interacting with the system. The setRateControlThresholds function in the RootERC20PredicateFlowRate contract does not emit an event when it updates the largeTransferThresholds critical storage variable for a token (figure 4.1). However, having an event emitted to refl ect such a change in the critical storage variable may allow other system and off-chain components to detect suspicious behavior in the system. Events generated during contract execution aid in monitoring, baselining behavior, and detecting suspicious activity. Without events, users and blockchain-monitoring systems cannot easily detect behavior that falls outside the baseline conditions, and malfunctioning contracts and attacks could go undetected. 1 function setRateControlThreshold (2 address token , 3 uint256 capacity , 4 uint256 refillRate , 5 uint256 largeTransferThreshold 6) external onlyRole(RATE_CONTROL_ROLE) { 7 _setFlowRateThreshold(token, capacity, refillRate); 8 largeTransferThresholds[token] = largeTransferThreshold; 9 } Figure 4.1: The setRateControlThreshold function (RootERC20PredicateFlowRate.sol #L214-L222) In addition to the above function, the following function should also emit events: • The setAllowedZone function in seaport/contracts/ImmutableSeaport.sol Recommendations Short term, add events for all functions that change state to aid in better monitoring and alerting. HashEye 22 Immutable Smart Contracts Security Assessment PUBLIC

Long term, ensure that all state-changing operations are always accompanied by events. In addition, use static analysis tools such as Slither to help prevent such issues in the future. HashEye 23 Immutable Smart Contracts Security Assessment PUBLIC

5. Withdrawal queue can be forcibly activated to hinder bridge operation Severity: Medium Difficulty: Low Type: Denial of Service Finding ID: TOB-IMM-5 Target: RootERC20PredicateFlowRate.sol Description The withdrawal queue can be forcibly activated to impede the proper operation of the bridge. The RootERC20PredicateFlowRate contract implements a withdrawal queue to more easily detect and stop large withdrawals from passing through the bridge (e.g., bridging illegitimate funds from an exploit). A transaction can enter the withdrawal queue in four ways: 1. If a token’s flow rate has not been configured by the rate control admin 2. If the withdrawal amount is larger than or equal to the large transfer threshold for that token 3. If, during a predefi ned period, the total withdrawals of that token are larger than the defi ned token capacity 4. If the rate controller manually activates the withdrawal queue by using the activateWithdrawalQueue function In cases 3 and 4 above, the withdrawal queue becomes active for all tokens, not just the individual transfers. Once the withdrawal queue is active, all withdrawals from the bridge must wait a specified time before the withdrawal can be finalized. As a result, a malicious actor could withdraw a large amount of tokens to forcibly activate the withdrawal queue and hinder the expected operation of the bridge. Exploit Scenario 1 Eve observes Alice initiating a transfer to bridge her tokens back to the mainnet. Eve also initiates a transfer, or a series of transfers to avoid exceeding the per-transaction limit, of sufficient tokens to exceed the expected

flow rate. With Alice unaware she is being targeted for griefi ng, Eve can execute her withdrawal on the root chain first, cause Alice's withdrawal to be pushed into the withdrawal queue, and activate the queue for every other bridge user. HashEye 24 Immutable Smart Contracts Security Assessment PUBLIC

Exploit Scenario 2 Mallory has identified an exploit on the child chain or in the bridge itself, but because of the withdrawal queue, it is not feasible to exfiltrate the funds quickly enough without risking getting caught. Mallory identifies tokens with small flow rate limits relative to their price and repeatedly triggers the withdrawal queue for the bridge, degrading the user experience until Immutable disables the withdrawal queue. Mallory takes advantage of this window of time to carry out her exploit, bridge the funds, and move them into a mixer. Recommendations Short term, explore the feasibility of withdrawal queues on a per-token basis instead of having only a global queue. Be aware that if the flow rates are set low enough, an attacker could feasibly use them to grief all bridge users. Long term, develop processes for regularly reviewing the configuration of the various token buckets. Fluctuating token values may unexpectedly make this type of griefi ng more feasible. HashEye 25 Immutable Smart Contracts Security Assessment PUBLIC

A. Vulnerability Categories The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document. Vulnerability Categories Category Description Access Controls Insufficient authorization or assessment of rights Auditing and Logging Insufficient auditing of actions or logging of problems Authentication Improper identification of users Configuration Misconfigured servers, devices, or software components Cryptography A breach of system confidentiality or integrity Data Exposure Exposure of sensitive information Data Validation Improper reliance on the structure or values of data Denial of Service A system failure with an availability impact Error Reporting Insecure or insufficient reporting of error conditions Patching Use of an outdated software package or library Session Management Improper identification of authenticated users Testing Insufficient test methodology or test coverage Timing Race conditions or other order-of-operations flaws Undefined Behavior Undefi ned behavior triggered within the system HashEye 26 Immutable Smart Contracts Security Assessment PUBLIC

Severity Levels Severity Description Informational The issue does not pose an immediate risk but is relevant to security best practices. Undetermined The extent of the risk was not determined during this engagement. Low The risk is small or is not one the client has indicated is important. Medium User information is at risk; exploitation could pose reputational, legal, or moderate financial risks. High The flaw could affect numerous users and have serious reputational, legal, or financial implications. Diffi culty 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 27 Immutable Smart Contracts 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 defi ned functions Cryptography and Key Management The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution Decentralization The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades Documentation The presence of comprehensive and readable codebase documentation 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 28 Immutable Smart Contracts 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 29 Immutable Smart Contracts Security Assessment PUBLIC

C. Code Quality Recommendations The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future. Bridge Contracts • The _setupRole function was deprecated in OpenZeppelin Contracts v4.4.0 in favor of the _grantRole function. Replace the deprecated calls with similar calls to _grantRole . • Using 0x1 as the root chain token address for native ether could have subtle unexpected side eff ects because this address corresponds to the ecrecover precompile. Consider using a different address outside of the existing precompile space (e.g., some protocols use an address consisting of all 0xe s). • Prefer using larger time units when possible to improve readability. For example, the DEFAULT_WITHDRAWAL_DELAY constant in the FlowRateWithdrawalQueue contract could be defi ned as 1 days instead of 60 * 60 * 24 . • Assigning constant state variables to be the result of an expression that uses other constant variables can lead to an increase in gas costs, bytecode size, and unexpected results. This is because the compiler will replace all mentions of the constant with the expression itself, as opposed to replacing it with the result of the expression in case of literals. This can cause subtle issues if the expression results in a revert or panic (e.g., arithmetic underflow). HashEye 30 Immutable Smart Contracts Security Assessment PUBLIC

D. Testing Improvement Recommendations This appendix aims to provide general recommendations on improving processes and creating a robust testing suite. General Recommendations Creating a robust testing suite is an iterative process that can involve defi ning or adjusting internal development processes and using multiple testing strategies and approaches. We compiled a list of general guidelines for improving testing suite quality below: 1. Define a clear test directory structure. A clear directory structure can be benefi cial for structuring the work of multiple developers, making it easier to identify which components and behaviors are being tested and giving insight into the overall test coverage. 2. Write a design specifi cation of the system, its components, and its functions in plain language. Defi ning a specification can allow the team to more easily detect bugs and inconsistencies in the system, reduce the likelihood that future code changes will introduce bugs, improve the maintainability of the system, and create a robust and holistic testing strategy. 3. Use the function specifi cations to guide the creation of unit tests. Creating a specification of all preconditions, postconditions, failure cases, entry points, and execution paths for a function will make it easier to maintain high test coverage and identify edge cases. 4. Use the interaction specifi cations to guide the creation of integration tests. An interaction specification will make it easier to identify the interactions that need to be tested and the external failure cases that need to be validated or guarded against. It will also help identify issues related to access controls and external calls. 5. Use fork testing for integration testing with third-party smart contracts and to ensure the deployed system works as expected. Fork testing can be used to test interactions between the protocol contracts and third-party smart contracts by providing an environment that is as close to production as possible. Additionally, fork testing can be used to identify whether the deployed system is behaving as expected. 6. Implement fuzz testing by first defining a set of system- and function-level invariants and then testing them with Echidna and/or Foundry. Fuzz testing is a powerful technique for exposing security vulnerabilities and finding edge cases that are unlikely to be found through unit testing or manual review. Fuzz testing can be done on a single function by passing in randomized arguments and on an entire HashEye 31 Immutable Smart Contracts Security Assessment PUBLIC

system by generating a sequence of random calls to various functions inside the system. Both testing approaches should be applied. 7. Use mutation testing to identify gaps in test coverage and more easily identify bugs in the code. Mutation testing can help identify coverage gaps in unit tests and help discover security vulnerabilities. Taking a two-pronged approach using Necessist to mutate tests and universalmutator to mutate source code can prove valuable in creating a robust testing suite. Directory Structure Creating a specific directory structure for the system’s tests will make it easier to develop and maintain the testing suite and to find coverage gaps. This section contains brief guidelines on defi ning a directory structure: • Create individual directories for each test type (e.g., unit/ , integration/ , fork/ , fuzz/) and for the utility contracts. The individual directories can be further divided into directories based on components or behaviors being tested. • Create a single base contract that inherits from the shared utility contracts and is inherited by individual test contracts. This will help reduce code duplication across the testing suite. • Create a clear naming convention for test files and test functions to make it easier to filter tests and understand the properties or contracts that are being tested. Specification Guidelines This section provides generally accepted best practices and guidance for how to write design specifications. A good design specification serves three purposes: 1. It helps development teams detect bugs and inconsistencies in a proposed system architecture before any code is written. Codebases written without adequate specifications are often littered with snippets of code that have lost their relevance as the system’s design has evolved. Without a specification, it is exceedingly challenging to detect such code snippets and remove them without extensive

validation testing. 2. It reduces the likelihood that bugs will be introduced in implementations of the system . In systems without a specification, engineers must divide their attention between designing the system and implementing it in code. In projects requiring multiple engineers, engineers may make assumptions about how another engineer’s component works, creating an opportunity for bugs to be introduced. HashEye 32 Immutable Smart Contracts Security Assessment PUBLIC

3. It improves the maintainability of system implementations and reduces the likelihood that future code changes will introduce bugs. Without an adequate specification, new developers need to spend time “on-ramping,” where they explore the code and learn how it works. This process is highly error-prone and can lead to incorrect assumptions and the introduction of bugs. Low-level designs may also be used by test engineers to create property-based fuzz tests and by auditors to reduce the time needed to audit a specific protocol component. Specification Construction A good specification must describe system components in enough detail that an engineer unfamiliar with the project can use the specification to implement those components. The level of detail required to achieve this can vary from project to project, but generally, a low-level specification will include the following details, at a minimum: • How each system component (e.g., a contract or plugin) interacts with and relies on other components • The actors and agents that participate in the system, the way they interact with the system, and their permissions, roles, authorization mechanisms, and expected known-good flows • The expected failure conditions the system may encounter and the way those failures are mitigated, including failures that are automatically mitigated • Specification details for each function that the system will implement, which should include the following: ○ A description of the function’s purpose and intended use ○ A description of the function’s inputs and the various validations that are performed against each input ○ Any specific restrictions on the function’s inputs that are not validated or not obvious ○ Any interactions between the function and other system components ○ The function’s various failure modes, such as failure modes for queries to a Chainlink oracle for a price (e.g., stale price, oracle disabled) ○ Any authentication or authorization required by the function ○ Any function-level assumptions that depend on other components behaving in a specific way HashEye 33 Immutable Smart Contracts Security Assessment PUBLIC

In addition, specifications should use standardized RFC-2119 language as much as possible. This language pushes specification authors toward a writing style that is both detailed and easy to understand. One relevant example is the ERC-4626 specification , which uses RFC-2119 language and provides enough constraints on implementers so that a vault client for a single implementation may be used interchangeably with other implementations. Interaction Specification Example An interaction specification is used to describe how the components of the system depend on each other. It includes a description of the other components that the system interacts with, the nature of those interactions, expected behavior or dependencies, and access relationships. A diagram can often be a helpful aid for modeling component interactions, but it should not be used to substitute a textual description of the component’s interactions. Part of the goal of a specification is to help derive a list of properties that can be explicitly tested, and deriving properties from a diagram is much more challenging and error-prone than deriving properties from a textual specification. Here is an example of an interaction specification. RootERC20Predicate interacts with the following contracts: • StateSender : RootERC20Predicate calls the syncState function of this contract whenever: 1. a root token is mapped to a child token address, or 2. an ERC-20 or native asset is deposited into the contract. This function will emit an event that will be picked up by the off-chain components and later included as part of a commit to the receiving chain. The following contracts interact with RootERC20Predicate : • [Example contract] The following actors interact with RootERC20Predicate : • [Example actor] Function Specification Example Here is an example of a specification for the mapToken(IERC20Metadata rootToken) function. The mapToken function can be called by any account to map the root token address to a child token on the other chain. If the rootToken input parameter is zero, mapToken () must revert. HashEye 34 Immutable Smart Contracts Security Assessment PUBLIC

If the rootToken address has already been mapped to a child token, mapToken () must revert. If the provided address is not the zero address and the address has not been mapped to a child token before, mapToken () must set the rootTokenToChildToken[rootToken] mapping to the address return value of the Clones.predictDeterministicAddress function, call the syncState function of the StateSender contract, emit the TokenMapped event, and return the child token address. Another complementary technique for defi ning a function specification, which can be especially useful for defi ning test cases, is the branching tree technique (proposed by Paul Berg), which is a tree-like structure based on all the execution paths, the contract state or function arguments that lead to each path, and the end result of each path. This type of specification can be useful when developing unit tests because it makes it easy to identify the execution paths, conditions, and edge cases that need to be tested. Constraints Specification Example Here is an example of a

constraints specification. A RootERC20Predicate contract must be initialized in the same transaction as the deployment by calling the initialize function. RootERC20Predicate has two constraints that limit the contract's operation: the stateSender address and the exitHelper address. The stateSender address must be defi ned during initialization and must implement the syncState function. If this address is not defi ned or is misconfigured, all calls to deposit or map tokens will fail, preventing the bridge from operating. The exitHelper address must be defi ned during initialization. If this address is not defi ned or is misconfigured, users will not be able to withdraw their bridged assets. Integration and Fork Testing Integration tests build on unit tests by testing how individual components integrate with each other or with third-party contracts. It can often be useful to run integration testing on a fork of the network to make the testing environment as close to production as possible and to minimize the use of mock contracts whose implementation can differ from the third-party contracts. We provide general recommendations on performing integration and fork testing below: • Use the interaction specifi cation to develop integration tests. Ensure that the integration tests aid in verifying the interaction's specification. HashEye 35 Immutable Smart Contracts Security Assessment PUBLIC

• Identify valuable input data for the integration tests that can maximize code coverage and test potential edge cases. • Use negative integration tests , similar to negative unit tests, to test common failure cases. • Use fork testing to build on top of the integration testing suite. Fork testing will aid in testing third-party contract integrations and the proper configuration of the system once it is deployed. • Enrich the forked integration testing suite with fuzzed values and call sequences (refer to the recommendations below). This will aid in increasing code coverage, validating system-level invariants, and identifying edge cases. Fuzz Testing • Define system- and function-level invariants. Invariants are properties that should always hold within a system, component, or function. Defi ning invariants is a prerequisite for developing eff ective fuzz tests that can detect unexpected behavior. Developing a robust system specification will directly aid in the identification of system- and function-level invariants. • Improving fuzz testing coverage. When using Echidna, regularly review the coverage files generated at the end of a run to determine whether the property tests' assertions are reached and what parts of the codebase are explored by the fuzzer. To improve the fuzzer's exploration and increase the chances that it finds an unexpected edge case, avoid overconstraining the function arguments. • Integrate fuzz testing into the CI/CD workflow. Continuous fuzz testing can help quickly identify any code changes that will result in a violation of a system property and can force developers to update the fuzz testing suite in parallel with the code. Running fuzz campaigns stochastically may cause a divergence between the operations in the code and the fuzz tests. • Add comprehensive logging mechanisms to all fuzz tests to aid in debugging. Logging during smart contract fuzzing is crucial for understanding the state of the system when a system property is broken. Without logging, it is difficult to identify the arithmetic or operation that caused the failure. • Enrich each fuzz test with comments explaining the preconditions and postconditions of the test. Strong fuzz testing requires well-defi ned preconditions (for guiding the fuzzer) and postconditions (for properly testing the invariants in question). Comments explaining the bounds on certain values and the importance of the system properties being tested will aid in testing suite maintenance and debugging eff orts. HashEye 36 Immutable Smart Contracts Security Assessment PUBLIC

Mutation Testing At a high level, mutation tests make several changes to each line of a target file and rerun the testing suite for each change. Changes that result in test failures indicate adequate test coverage, while changes that do not result in test failures indicate gaps in the test coverage. Although mutation testing is a slow process, it allows auditors to focus their review on areas of the codebase that are most likely to contain latent bugs, and it allows developers to identify and add missing tests. We recommend using two mutation tools, both of which can help detect redundant code, insufficient test coverage, incorrectly defi ned tests or conditions, and bugs in the underlying source code under test: • Necessist performs mutation of the testing suite by iteratively removing lines in the test cases. • universalmutator performs mutation of the underlying source code. HashEye 37 Immutable Smart Contracts Security Assessment PUBLIC

E. 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 October 2, 2023, HashEye reviewed the fixes and mitigations implemented by the Immutable team for the issues identified in this report. We reviewed each fix to determine its eff ectiveness in resolving the associated issue. In summary, the Immutable team has resolved all five issues described in this report. For additional information, please see the Detailed Fix Review Results below. ID Title Status 1 Initialization functions vulnerable to front-running Resolved 2 Lack of lower and upper bounds for system parameters Resolved 3 RootERC20Predicate is incompatible with nonstandard ERC-20 tokens Resolved 4 Lack of event generation Resolved 5 Withdrawal queue can be forcibly activated to hinder bridge operation Resolved HashEye 38 Immutable Smart Contracts Security Assessment PUBLIC

Detailed Fix Review Results TOB-IMM-1: Initialization functions vulnerable to front-running Resolved in commit 9156c04 through inline documentation. The Immutable team incorporated explanatory inline comments in both initialize functions. These comments highlight the potential for front-running and propose preventive steps, such as calling initialize when TransparentUpgradeableProxy is deployed through its constructor or calling initialize post-deployment of TransparentUpgradeableProxy to check for possible front-running. TOB-IMM-2: Lack of lower and upper bounds for system parameters Resolved in commit 29251f9 through inline documentation. The Immutable team added descriptive inline comments for both the setWithdrawalDelay and setRateControlThreshold functions, outlining the potential risks associated with setting excessively high and low threshold values and outlining the strategies to mitigate these risks, such as monitoring and resetting the threshold values. TOB-IMM-3: RootERC20Predicate is incompatible with nonstandard ERC-20 tokens Resolved in commit 9b5a008 through inline documentation. The Immutable team added inline comments indicating that the deposit function is not compatible with nonstandard ERC-20 tokens that implement fees during transfers. TOB-IMM-4: Lack of event generation Resolved in commit 64f1a11 and commit b46f2cc . Both the setRateControlThreshold and setAllowedZone functions now incorporate the events emitted after the execution of the corresponding action. TOB-IMM-5: Withdrawal queue can be forcibly activated to hinder bridge operation Resolved in commit 7ff87f2 through inline documentation. The Immutable team incorporated inline comments in the RootERC20PredicateFlowRate contract to highlight the dangers of misconfigured flow rates, which could potentially be exploited by an attacker, and to outline the measures to avoid such configuration errors.

F. Fix Review Status Categories The following table describes the statuses used to indicate whether an issue has been sufficiently addressed. Fix Status Status Description Undetermined The status of the issue was not determined during this engagement. Unresolved The issue persists and has not been resolved. Partially Resolved The issue persists but has been partially resolved. Resolved The issue has been sufficiently resolved.