HashEye

// SMART CONTRACT SECURITY AUDIT

Scroll Euclid Phase 2

Security assessment by HashEye · prepared for Scroll

HASHEYE AUDITED

PROJECT	Scroll Euclid Phase 2
PERFORMED BY	HashEye Security Review Team - Sergey, Brian
RESEARCH	Sergey, Brian
CLIENT	Scroll
CATEGORY	Scroll
PUBLISHED	April 1, 2025
REPORT ID	research-scroll-euclid-phase-2-2025-04-01-kkayp7

This report was produced by HashEye's security review team under the HashEye review process, combining source review, deployment-state checks, and senior manual verification. Full report detail is available at hasheye.io/audits/research-scroll-euclid-phase-2-2025-04-01-kkayp7.

Scroll Euclid Upgrade Phase 1 Security Assessment April 4, 2025

Prepared for: Roy Lou Scroll

Prepared by: Filipe Casal and Marc Ilunga

Table of Contents

Project Summary Contact Information The following project manager was associated with this project: Jeff Braswell, Project Manager jeff.braswell@hasheye.io The following engineering director was associated with this project: Jim Miller, Engineering Director, Cryptography james.miller@hasheye.io The following consultants were associated with this project: Filipe Casal, Consultant Marc Ilunga, Consultant filipe.casal@hasheye.io marc.ilunga@hasheye.io Project Timeline The significant events and milestones of the project are listed below. Date Event February 21, 2025 Pre-project kickoff call February 28, 2025 Status update meeting #1 March 6, 2025 Delivery of report draft and report readout meeting April 4, 2025 Delivery of final comprehensive report

Executive Summary Engagement Overview Scroll engaged HashEye to review the security of changes related to the Euclid Upgrade. These changes include OpenVM guest programs for the chunk, batch, and bundle verification circuits, changes to the Scroll smart contracts related to supporting a special state transition due to the change in the computation of the state root, and a Go command-line tool to validate states during the Euclid transition. A team of two consultants conducted the review from February 24 to March 5, 2025, for a total of three engineer-weeks of effort. Our testing efforts focused on assessing the soundness and correctness of the zero-knowledge circuits, whether the Euclid finalization function works as specified, and whether the Go command-line tool can validate that the state of two nodes is identical at a certain block height. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes. Observations and Impact We identified two high-severity soundness issues (TOB-SCREUC-1 and TOB-SCREUC-2) affecting the batch and bundle circuits that would allow obtaining a bundle proof for which a malicious prover could bypass the verification of the inner proofs for the chunk and batch circuits. We identified two impactful but difficult-to-exploit issues (TOB-SCREUC-6 and TOB-SCREUC-7) related to the GitHub actions used in the codebases that could undermine the security of the released images to DockerHub. Note that GitHub actions security was not in the audit's scope; we identified these issues using a static analysis tool for GitHub actions. We did not identify security issues in the smart contract related to the one-off Euclid finalization functionality or in the Go command-line utility. However, we found that the migration checker was tested only with test data and does not have associated unit tests for its subfunctions (see TOB-SCREUC-3). Recommendations Based on the findings identified during the security review, HashEye recommends that Scroll take the following steps: • Remediate the findings disclosed in this report. These findings should be addressed as part of a direct remediation or any refactor that may occur when addressing other recommendations.

• Add tests for the migration checker. The migration checker consists of several functions, each performing a specific verification test. These components are relatively self-contained and, therefore, can be tested against correct and incorrect inputs. Sufficient testing of the migration checker will help increase confidence in its correctness. Finding Severities and Categories

The following tables provide the number of findings by severity and category. EXPOSURE ANALYSIS Severity Count High 2 Medium 0 Low 2 Informational 4 Undetermined 0

CATEGORY BREAKDOWN Category Count Auditing and Logging 1 Configuration 2 Cryptography 2 Data Validation 2 Testing 1

Project Goals The engagement was scoped to provide a security assessment of the Scroll Euclid phase 1 code changes. Specifically, we sought to answer the following non-exhaustive list of questions: • Are there any soundness or correctness issues in the chunk, batch, and bundle verification circuits? • Is the integration with the OpenVM verifier correctly implemented on the smart contract? • Does the Go command-line tool correctly validate whether a given MPT and a ZKTrie correspond to the same inherent state? • Can the Security Council correctly finalize the Euclid transition batch? • Can the Euclid finalize function be maliciously used to finalize batches other than the transition batch? • Does batch reverting correctly reset the Euclid batch index? • Does the finalizeBundleWithProof function correctly prevent v4, v5, and v6 batches in the same bundle?

HashEye 5 Scroll Euclid Upgrade Phase 1

PUBLIC Security Assessment

Project Targets The engagement involved a review and testing of the targets listed below. zkvm-prover Repository https://github.com/scroll-tech/zkvm-prover/ Version da9c824e9a0751bd5482d00910a187e86350beee Type Rust, OpenVM scroll-contracts Repository https://github.com/scroll-tech/scroll-contracts/ Version c382953479f9a919e850e56115924e5711810dfd Type Solidity go-ethereum Repository https://github.com/scroll-tech/go-ethereum Version 642c35f9906cab9001ac77190e198d18e2a044b5 Type Go

HashEye 6 Scroll Euclid Upgrade Phase 1

PUBLIC Security Assessment

Project Coverage This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following: • Zkvm-prover. We manually reviewed the chunk, batch, and bundle circuits, focusing on identifying potential soundness and correctness issues. • Smart contracts. We manually reviewed the smart contracts, focusing on determining if the OpenVM verifier is correctly integrated to verify the zero-knowledge proof; if the Security Council can perform the Euclid state transition using the implemented functionality; if this functionality can be maliciously used after the Euclid transition occurs; and if batch reversion correctly updates the initialEuclidBatchIndex state variable. We additionally assessed whether the bundle proof and public input are stored in memory according to the memory layout in the documentation. • Migration checker. We manually reviewed the migration checker tool, focusing on assessing whether the code faithfully checks that a zkTrie and a Merkle Patricia Trie both encode the same chain state.

HashEye 7 Scroll Euclid Upgrade Phase 1

PUBLIC Security Assessment

Automated Testing HashEye uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in-house, to perform automated testing of source code and compiled software. Test Harness Configuration We used the following tools in the automated testing phase of this project: Tool Description Policy Semgrep An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time Appendix C Clippy An open-source Rust linter used to catch common mistakes and unidiomatic Rust code Appendix C Dylint A tool for running Rust lints from dynamic libraries Appendix C cargo-audit An open-source tool for checking dependencies against the RustSec advisory database Appendix C cargo-edit A tool for quickly identifying outdated crates Appendix C zizmor

A static analysis tool for GitHub Actions Appendix C CodeQL A code analysis engine developed by GitHub to automate security checks Appendix C

Areas of Focus Our automated testing and verification work focused on the following: • General code quality issues and unidiomatic code patterns • Security of CI and GitHub Actions • Identifying dependencies that are outdated or vulnerable to known attacks

HashEye 8 Scroll Euclid Upgrade Phase 1

PUBLIC Security Assessment

Test Results

The results of this focused testing are detailed below. Clippy Clippy reports warnings for three idiomatic and performance rules: manual assert, or_fun_call and explicit_iter_loop which we detailed in the Code Quality appendix. We recommend adding these rules to your regular Clippy runs. zizmor Running zizmor on the codebase identified findings TOB-SCREUC-6, TOB-SCREUC-7, and TOB-SCREUC-8. We recommend adding zizmor to your CI/CD pipeline across the organization’s repositories.

cargo-edit Cargo-edit identified several outdated dependencies, including some that are, in a semantic versioning sense, compatible with the version currently in use, and others that are not. Consider integrating Dependabot to automatically update dependencies as soon as these are available. $ cargo upgrade --incompatible --dry-run Checking virtual workspace's dependencies name old req compatible latest new req ==== ======= ========== ====== ======= bitcode 0.6.3 0.6.5 0.6.5 0.6.5 halo2curves-axiom 0.5.3 0.5.3 0.7.0 0.7.0 metrics 0.23.0 0.23.0 0.24.1 0.24.1 metrics-util 0.17 0.17.0 0.19.0 0.19 alloy-serde 0.8 0.8.3 0.11.1 0.11 serde_with 3.11.0 3.12.0 3.12.0 3.12.0 toml 0.8.14 0.8.20 0.8.20 0.8.20 Checking scroll-zkvm-batch-circuit's dependencies Checking scroll-zkvm-build-guest's dependencies name old req compatible latest new req ==== ======= ========== ====== ======= cargo_metadata 0.19.1 0.19.2 0.19.2 0.19.2 Checking scroll-zkvm-bundle-circuit's dependencies Checking scroll-zkvm-chunk-circuit's dependencies Checking scroll-zkvm-circuit-input-types's dependencies Checking scroll-zkvm-integration's dependencies name old req compatible latest new req ==== ======= ========== ====== ======= c-kzg 1.0 1.0.3 2.0.0 2.0 Checking scroll-zkvm-prover's dependencies name old req compatible latest new req ==== ======= ========== ====== =======

git-version 0.3.5 0.3.9 0.3.9 0.3.9 revm 19.0 19.5.0 19.5.0 19.5 Checking scroll-zkvm-verifier's dependencies name old req compatible latest new req ==== ======= ========== ====== ======= revm 19.0 19.5.0 19.5.0 19.5 Figure T.1: Result of running cargo-edit on the zkvm-prover codebase

cargo-audit cargo-audit did not identify any dependency vulnerable to known security issues.

CodeQL We ran CodeQL on the migration checker. CodeQL did not identify any vulnerabilities.

Summary of Findings The table below summarizes the findings of the review, including details on type and severity. ID Title Type Severity 1 The aggregation circuit does not validate the size of inner proofs Cryptography High 2 Insufficient validation of chunk information and blob bytes in the batch circuit Cryptography High 3 MPT migration code lacks unit tests Testing Informational 4 Incorrect data present on BundleInfo Data Validation Informational 5 Lax parsing of chunk data payload Data Validation Informational 6 Docker release action is vulnerable to cache poisoning Configuration Low 7 Unpinned external GitHub CI/CD action versions Auditing and Logging Low 8 Potential credential persistence in artifacts and stale GitHub action Configuration Informational

Detailed Findings

1. The aggregation circuit does not validate the size of inner proofs Severity: High Difficulty: Low Type: Cryptography Finding ID: TOB-SCREUC-1 Target: circuits/types/src/lib.rs, circuits/batch-circuit/src/main.rs, circuits/bundle-circuit/src/main.rs

Description The batch and bundle circuits are aggregation circuits. They aggregate proofs generated by inner circuits. They validate the aggregated public inputs and verify the proofs embedded in the witness. However, due to missing enforcement of the lengths of aggregated proofs, an aggregation circuit would verify an aggregated public input without the associated inner proof. Figure 1.1 shows the main function of the batch circuit. The function calls C::verify_proofs(witness), which verifies the inner proof embedded in the witness. Subsequently, the aggregated public inputs are validated against the aggregated public input hashes. fn main() { // Setup openvm extensions for the circuit. C::setup(); [...] // Verify the root proofs being aggregated in the circuit. let agg_proofs = C::verify_proofs(witness);

// Get the public-input values of the proofs being aggregated from witness. let agg_pis = C::aggregated_public_inputs(witness);

// Derive the digests of the public-input values of proofs being aggregated. let agg_pi_hashes = C::aggregated_pi_hashes(&agg_proofs);

// Validate that the pi hashes derived from the root proofs are in fact the digests of the // public-input values of the previous circuit layer. C::validate_aggregated_pi(&agg_pis, &agg_pi_hashes);

[...] // Reveal the public-input values of the current circuit layer. C::reveal_pi(&public_inputs);

HashEye 12 Scroll Euclid Upgrade Phase 1

PUBLIC Security Assessment

} Figure 1.1: The main process of the batch circuit (circuits/batch-circuit/src/main.rs#L18-L47) Figure 1.2 shows that the proof verification function does not check the length of the received vector of proofs and can accept an empty vector. Therefore, the returned aggregated proofs, agg_proofs, are also empty. /// Verify the proofs being aggregated. /// /// Also returns the root proofs being aggregated. fn verify_proofs(witness: &Self::Witness) -> Vec<proof::AggregationInput> { let proofs = witness.get_proofs();

for proof in proofs.iter() { Self::verify_commitments(&proof.commitment); proof::verify_proof(&proof.commitment, proof.public_values.as_slice()); }

proofs } Figure 1.2: Verification of aggregated proofs (circuits/types/src/lib.rs#L69-L81) As a consequence, the validation of the aggregated public inputs against the public input hashes—the call to C::validate_aggregated_pi(&agg_pis, &agg_pi_hashes) shown in figure 1.1—effectively verifies the aggregated public input against an empty list of hashes. Due to the use of zip, the lengths are checked for equality. With an empty vector of hash values, an assertion error would occur, and the circuit would terminate successfully. /// Validate that the public-input values of the aggregated proofs are well-formed. /// /// - That the public-inputs of contiguous chunks/batches are valid /// - That the public-input values in fact hash to the pi_hash values from the root proofs. fn validate_aggregated_pi(agg_pis: &[Self::AggregatedPublicInputs], agg_pi_hashes: &[B256]) { // Validation for the contiguous public-input values. for w in agg_pis.windows(2) { w[1].validate(&w[0]); }

// Validation for public-input values hash being the pi_hash from root proof. for (agg_pi, &agg_pi_hash) in agg_pis.iter().zip(agg_pi_hashes.iter()) { assert_eq!(agg_pi.pi_hash(), agg_pi_hash, "pi hash mismatch between proofs and witness computed"

HashEye 13 Scroll Euclid Upgrade Phase 1

PUBLIC Security Assessment

); } } Figure 1.3: Validation of aggregated public input against expected public input hashes (circuits/types/src/lib.rs#L89-L107) Exploit Scenario An attacker wishes to produce a wrong aggregated proof for the bundle input. They generate an empty witness and fabricate aggregated inputs, which the circuit verifies. Recommendations Short term, ensure that the aggregated proofs are not empty. Use zip_eq when comparing elements expected to have the same size. Long term, document invariants and assumptions that must hold in every part of the codebase.

HashEye 14 Scroll Euclid Upgrade Phase 1

PUBLIC Security Assessment

2. Insufficient validation of chunk information and blob bytes in the batch circuit Severity: High Difficulty: Medium Type: Cryptography Finding ID: TOB-SCREUC-2 Target: crates/circuits/batch-circuit/src/builder/v3.rs

Description The witness validation in the batch circuit does not properly validate that the chunk information matches the provided blob bytes. This leads to the circuit returning a batch header hash that uses blob data proof inconsistent with the chunks in the batch. Specifically, the batch circuit does not sufficiently validate that the chunk information transaction data digest matches the chunk data digest present in the EIP-4844 blob bytes, as it does not compare the length of the chunks_info slice with the length of the payload.chunk_data_digests slice. // Validate the tx data is match with fields in chunk info for (chunk_info, &tx_data_digest) in chunks_info.iter().zip(payload.chunk_data_digests.iter()) { assert_eq!(chunk_info.tx_data_digest, tx_data_digest); } Figure 2.1: crates/circuits/batch-circuit/src/builder/v3.rs#L32-L37

For example, if chunks_info contains information about two chunks but the blob_bytes has encoded only the first chunk, the zip iteration will validate only the first chunk data digest, and the blob consistency check will generate a blob data proof that concerns only the first chunk.

Exploit Scenario A malicious prover provides a witness value with a different number of chunks for the chunk_info and the blob_bytes, resulting in a batch information PI that is inconsistent: it contains a batch hash with the only the chunks in blob_bytes, but its state root and withdraw root concern the last chunk in chunk_info. Recommendations Short term, ensure that the lengths of chunks_info and payload.chunk_data_digests match, either with a separate assertion or by using zip_eq.

HashEye 15 Scroll Euclid Upgrade Phase 1

PUBLIC Security Assessment

Long term, review all uses of zip in the codebase for implicit assumptions on the length of the iterators and enforce those length checks where necessary.

HashEye 16 Scroll Euclid Upgrade Phase 1

PUBLIC Security Assessment

3. MPT migration code lacks unit tests Severity: Informational Difficulty: High Type: Testing Finding ID: TOB-SCREUC-3 Target: go-ethereum/cmd/migration-checker/main.go

Description As part of the Euclid upgrade, the Scroll team will migrate the chain state from the current zktrie to a Merkle Patricia Tree (MPT). The team has developed a tool in Go that allows the security council to check for a faithful migration of the chain’s state. However, there are no unit tests to ensure that the migration checker is implemented correctly. The migration checker code contains several subfunctions that perform specific verification tasks. Figure 3.1 shows a function used to check that storage nodes are equal across the zktrie and MPT data structures. The function scope is small enough and self-contained to enable unit tests exercising its behavior in both expected and adversarial scenarios. func checkStorageEquality(label string, _ *dbs, zkStorageBytes, mptStorageBytes []byte, _ bool) { zkValue := common.BytesToHash(zkStorageBytes) _, content, _, err := rlp.Split(mptStorageBytes) panicOnError(err, label, "failed to decode mpt storage") mptValue := common.BytesToHash(content) if !bytes.Equal(zkValue[:], mptValue[:]) { panic(fmt.Sprintf("%s storage mismatch: zk: %s, mpt: %s", label, zkValue.Hex(), mptValue.Hex())) } } Figure 3.1: (go-ethereum/cmd/migration-checker/main.go#158-166) However, the Scroll team has tested the migration checker on some of its test nodes. Though this test was successful, it may not necessarily guarantee the robustness of the code against incorrect inputs. Recommendations Short term, write unit tests for the migration checker before deployment. Ensure that both correctness and robustness are tested appropriately. Long term, ensure all critical software used for irreversible tasks is appropriately tested.

HashEye 17 Scroll Euclid Upgrade Phase 1

PUBLIC Security Assessment

4. Incorrect data present on BundleInfo Severity: Informational Difficulty: N/A Type: Data Validation Finding ID: TOB-SCREUC-4 Target: crates/circuits/bundle-circuit/src/circuit.rs

Description The BundleCircuit::validate function returns an incorrect prev_state_root value in the BundleInfo data. Instead of first_batch.parent_batch_hash, the correct value is first_batch.parent_state_root. let prev_state_root = first_batch.parent_batch_hash.into(); let prev_batch_hash = first_batch.parent_batch_hash.into(); Figure 4.1: crates/circuits/bundle-circuit/src/circuit.rs#L57-L58 Given that the smart contract rederives the public inputs from independent data, incorrectly setting this value causes a correctness issue when verifying the proof on the smart contract side. The Scroll team has also independently identified this issue. Recommendations Short term, modify the code to use the correct field for prev_state_root. Long term, document how each layer’s public inputs are used. Add tests verifying the correctness of the returned public inputs. Enforce end-to-end testing from proving to smart contract verification to ensure end-to-end correctness.

HashEye 18 Scroll Euclid Upgrade Phase 1

PUBLIC Security Assessment

5. Lax parsing of chunk data payload Severity: Informational Difficulty: N/A Type: Data Validation Finding ID: TOB-SCREUC-5 Target: crates/circuits/batch-circuit/src/payload.rs

Description The Payload::from_payload function does not fully validate the metadata chunk sizes when less than N_MAX_CHUNKS are present. The function parses the payload by 1) obtaining the number of chunks present valid_chunks, 2) obtaining the chunk size of each chunk up to N_MAX_CHUNKS and considering only the first valid_chunks lengths, and finally, obtaining the chunk bytes based on the metadata lengths while validating that no extra bytes remain at the end of the payload. However, this procedure does not validate that if, for example, only two chunks are present, all the remaining N_MAX_CHUNKS-2 chunk lengths equal zero. This means that different metadata bytes can lead to the same chunk data digests, as the payload provider could state that the chunk lengths after chunk number two are arbitrary values, not changing the behavior of the rest of the parsing. // Decoded batch bytes require segmentation based on chunk length let valid_chunks = metadata_bytes[..N_BYTES_NUM_CHUNKS] .iter() .fold(0usize, |acc, &d| acc * 256usize + d as usize);

let chunk_size_bytes = metadata_bytes[N_BYTES_NUM_CHUNKS..] .iter() .chunks(N_BYTES_CHUNK_SIZE); let chunk_lens = chunk_size_bytes .into_iter() .map(|chunk_bytes| chunk_bytes.fold(0usize, |acc, &d| acc * 256usize + d as usize)) .take(valid_chunks);

// reconstruct segments let (segmented_batch_data, final_bytes) = chunk_lens.fold((Vec::new(), batch_bytes), |(mut datas, rest_bytes), size| { datas.push(Vec::from(&rest_bytes[..size])); (datas, &rest_bytes[size..]) },

);

assert!(final_bytes.is_empty(), "chunk segmentation len must add up to the correct value"); Figure 5.1: crates/circuits/batch-circuit/src/payload.rs#L37-L62

Given that both the metadata digest and chunk digests are used in the computation of the challenge digest, we could not identify a way of further exploiting this issue.

Recommendations Short term, validate that the size of the chunks past the number specified in valid_chunks equals zero, ensuring that each unique payload corresponds to a unique pair of metadata digests and batch data digests.

6. Docker release action is vulnerable to cache poisoning Severity: Low Difficulty: High Type: Configuration Finding ID: TOB-SCREUC-6 Target: go-ethereum/.github/workflows/{docker-arm64.yaml, docker.yaml}, scroll-contracts/.github/workflows/docker-release.yml

Description The Docker release GitHub workflows use caching to store the buildx binary. An attacker who gains write access to the GitHub actions cache (e.g., through a valid GitHub token) can poison the cache and compromise the build process to maliciously alter build artifacts. - name: Set up Docker Buildx id: buildx uses: docker/setup-buildx-action@v2 - name: Login to Docker Hub uses: docker/login-action@v2 with: username: ${{ secrets.DOCKERHUB_USERNAME }} password: ${{ secrets.DOCKERHUB_TOKEN }} - name: Build docker image uses: docker/build-push-action@v2 with: platforms: linux/arm64 context: . file: Dockerfile push: true tags: scrolltech/l2geth:${{inputs.tag}}-arm64 Figure 6.1: .github/workflows/docker-arm64.yaml#26-41

jobs: build-and-push: runs-on: ubuntu-latest steps: - name: Checkout code uses: actions/checkout@v2 - name: Set up QEMU uses: docker/setup-qemu-action@v2 - name: Set up Docker Buildx uses: docker/setup-buildx-action@v2 Figure 6.2: .github/workflows/docker.yaml#10-19

- name: Set up Docker Buildx uses: docker/setup-buildx-action@v2

- name: Login to Dockerhub uses: docker/login-action@v2 with: username: ${{ secrets.DOCKERHUB_USERNAME }} password: ${{ secrets.DOCKERHUB_TOKEN }} Figure 6.3:

.github/workflows/docker-release.yml#31-39

Exploit Scenario An attacker gains unauthorized access to a GitHub token with write permissions to the GitHub actions cache. She uses this to poison the cache to inject a malicious buildx binary. When the Docker container is built, the final image is modified to download and run attacker-controlled software. Since the codebase is not tampered with directly, the attack goes unnoticed by the Scroll team. Recommendations Short term, in general, avoid using previously cached CI state within workflows intended to publish build artifacts such as Docker containers. Caching can be disabled by setting the cache-binary option to false. Long term, add zizmor to the CI/CD pipeline to detect GitHub action-related issues.

7. Unpinned external GitHub CI/CD action versions Severity: Low Difficulty: High Type: Auditing and Logging Finding ID: TOB-SCREUC-7 Target: .github/workflows/{docker-arm64.yaml, docker.yaml}, .github/workflows/l2geth_ci.yml

Description Several GitHub actions workflows in the go-ethereum and scroll-contracts repositories use third-party actions pinned to a tag or branch name instead of a full commit SHA as recommended by GitHub. This configuration enables repository owners to silently modify the actions. A malicious actor could use this ability to tamper with an application release or leak secrets. The following actions are owned by GitHub organizations or individuals that are not affiliated directly with Scroll:

• foundry-rs/foundry-toolchain@v1 • hrishikesh-kadam/setup-lcov@v1 • codecov/codecov-action@v3 • docker/setup-qemu-action@v2 • docker/setup-buildx-action@v2 • docker/login-action@v2 • docker/build-push-action@v3 • docker/metadata-action@v3 • actions-rs/toolchain@v1 jobs: foundry: if: github.event.pull_request.draft == false runs-on: ubuntu-latest

steps: - name: Checkout sources

uses: actions/checkout@v4 with: submodules: recursive

- name: Install Foundry uses: foundry-rs/foundry-toolchain@v1 with: version: nightly

- name: Setup LCOV uses: hrishikesh-kadam/setup-lcov@v1 Figure 7.1: .github/workflows/contracts.yml#24-41

Exploit Scenario An attacker gains unauthorized access to the account of a GitHub action owner. The attacker manipulates the action’s code to steal GitHub credentials and compromise the repository. Recommendations Short term, pin each third-party action to a specific full-length commit SHA, as recommended by GitHub. Additionally, configure Dependabot to update the actions’ commit SHAs after reviewing their available updates. Long term, add zizmor to the CI/CD pipeline.

8. Potential credential persistence in artifacts and stale GitHub action Severity: Informational Difficulty: N/A Type: Configuration Finding ID: TOB-SCREUC-8 Target: scroll-contracts/.github/workflows/{contracts.yml, docker-release.yml}, go-ethereum/.github/workflows/*

Description The default behavior of the actions/checkout GitHub action is to persist credentials, which means that the GitHub token is written to the local repository directory. This allows the action to execute Git commands that require authentication. The credentials are stored in .git/config files in the local repository directory, where they could be inadvertently included in workflow artifacts or accessed by subsequent workflow steps. name: Docker

on: push: tags: - '*' release: types: [published]

jobs: build-and-push: runs-on: ubuntu-latest steps: - name: Checkout code uses: actions/checkout@v2 Figure 8.1: .github/workflows/docker.yaml#1-15

The current Dockerfile does not appear to copy the .git/ folder to the image, meaning that this is not currently a security risk. However, if the Dockerfile is altered, credentials could be exposed in the Docker image. We also observed that scroll-contracts/.github/workflows/docker-release.yml file references a Dockerfile that no longer exists in the repository, so it can be removed from the codebase. Recommendations Short term, add persist-credentials: false to checkout actions that do not require privileged Git operations.

HashEye 25 Scroll Euclid Upgrade Phase 1

PUBLIC Security Assessment

Long term, add zizmor to the CI/CD pipeline. References • ArtiPACKED: Hacking Giants Through a Race Condition in GitHub Actions Artifacts • zizmor ArtiPACKED documentation

HashEye 26 Scroll Euclid Upgrade Phase 1

PUBLIC Security Assessment

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

HashEye 27 Scroll Euclid Upgrade Phase 1

PUBLIC Security Assessment

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

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

HashEye 28 Scroll Euclid Upgrade Phase 1

PUBLIC Security Assessment

B. Code Quality Findings We identified the following code quality issues through manual and automatic code review. These issues are not associated with any specific vulnerabilities. However, they will enhance code readability and may prevent the introduction of vulnerabilities in the future.

• Manual assert construction. The following if statement plus panic statement is equivalent to an assert statement. Use Clippy’s manual assert rule to find and fix all instances of this pattern in the codebase. fn verify_commitments(commitment: &ProgramCommitment) { if commitment.exe ≠ CHUNK_EXE_COMMIT { panic!("mismatch chunk-proof exe commitment: expected={:?}, got={:?}", CHUNK_EXE_COMMIT, commitment.exe,); } if commitment.leaf ≠ CHUNK_LEAF_COMMIT { panic!("mismatch chunk-proof leaf commitment: expected={:?}, got={:?}", CHUNK_EXE_COMMIT, commitment.leaf,); } } Figure B.1: crates/circuits/batch-circuit/src/circuit.rs#46-59

• Use of ok_or with function calls. Arguments passed to ok_or are eagerly evaluated and may lead to unnecessary computation. Since, in this case, a function is evaluated, we recommend using ok_or_else instead, as it will evaluate this function only if needed, preventing unnecessary computation. The codebase contains 16 instances of this pattern. Use Clippy's or_fun_call rule to find and fix all instances of this pattern. let sbv_chunk_info = SbvChunkInfo::from_blocks(chain_id, first_block.pre_state_root(), &blocks);

let chain_spec = get_chain_spec(Chain::from_id(sbv_chunk_info.chain_id())).ok_or(Error::GenProof(format!("{err_prefix}: failed to get chain spec")),)?; Figure B.2: crates/prover/src/prover/chunk.rs#83-88

• Loop code could be simplified. Some loops explicitly call .iter() when this is not needed. Figure B.3 shows a case where the .iter() call can be removed. Use Clippy's explicit_iter_loop rule to find and fix all instances of this pattern.

for block in blocks.iter() { let output = ManuallyDrop::new(EvmExecutor::new(chain_spec.clone(), &db, block) .execute() .expect("failed to execute block"),); db.update(&nodes_provider, output.state.state.iter()) .expect("failed to update db"); } Figure B.3: crates/circuits/chunk-circuit/src/execute.rs#56-64 • Incorrect documentation on the BatchHeaderV3Codec contract. The BatchHeaderV3Codec contract has two instances of incorrect documentation. The lastBlockTimestamp documentation mentions the L1 skipped messages, and the storeBlobDataProof documentation refers the timestamp: /// * lastBlockTimestamp 8 uint64 121 A bitmap to indicate which L1 messages are skipped in the batch /// * blobDataProof 64 bytes64 129 The blob data proof: z (32), y (32) Figure B.4: src/libraries/codec/BatchHeaderV3Codec.sol#17-18

/// @notice Store the last block timestamp of batch header. /// @param batchPtr The start memory offset of the batch header in memory. /// @param blobDataProof The blob data proof: z (32), y (32) function storeBlobDataProof(uint256 batchPtr, bytes calldata blobDataProof) internal pure { Figure B.5: src/libraries/codec/BatchHeaderV3Codec.sol#66-69

• Imprecise documentation on the ScrollChain contract. The error is raised when the parent batch hash of the genesis block is non-zero, instead of the state zero: /// @dev Thrown when the parent batch hash in genesis batch is zero. error ErrorGenesisParentBatchHashIsNonZero(); Figure B.6: src/L1/rollup/ScrollChain.sol#60-61

• Imprecise documentation of _loadBatchHeader. The documentation states that _totalL1MessagesPoppedOverall is a parameter, but it is a return value. /// @return _batchIndex The index of this batch. /// @param _totalL1MessagesPoppedOverall The number of L1 messages popped after this batch. function _loadBatchHeader(bytes calldata _batchHeader) internal view virtual

returns (uint256 batchPtr, bytes32 _batchHash, uint256 _batchIndex, uint256 _totalL1MessagesPoppedOverall) Figure B.7: src/L1/rollup/ScrollChain.sol#775-786 • Panic messages use incorrect data. The panic messages incorrectly reference BATCH_EXE_COMMIT and CHUNK_EXE_COMMIT when compared against BATCH_LEAF_COMMIT and CHUNK_LEAF_COMMIT. if commitment.leaf ≠ BATCH_LEAF_COMMIT { panic!("mismatch batch-proof leaf commitment: expected={:?}, got={:?}", BATCH_EXE_COMMIT, commitment.leaf,); } Figure B.8: crates/circuits/bundle-circuit/src/circuit.rs#L85-L90

if commitment.leaf ≠ CHUNK_LEAF_COMMIT { panic!("mismatch chunk-proof leaf commitment: expected={:?}, got={:?}", CHUNK_EXE_COMMIT, commitment.leaf,); } Figure B.9: crates/circuits/batch-circuit/src/circuit.rs#L53-L58 • Redundant batch index validation on the commitBatchWithBlobProof function. The commitBatchWithBlobProof function has a redundant check that batchIndex > euclidForkBatchIndex. The check appears to exist to allow committing v4 batches at an earlier index than the euclidForkBatchIndex (after this index has been set). However, given that batches need to be committed sequentially, it is not possible to commit a v4 batch before the already-committed v5 batch.

// Don't allow to commit version 4 after Euclid upgrade. // This check is to avoid sequencer committing wrong batch due to human error. // And This check won't introduce much gas overhead (likely less than 100). if (_version == 4) { uint256 euclidForkBatchIndex = initialEuclidBatchIndex; if (euclidForkBatchIndex > 0 && batchIndex > euclidForkBatchIndex) revert ErrorEuclidForkEnabled(); Figure B.10: src/L1/rollup/ScrollChain.sol#319-324

- The parallel branch loadMPT could be simplified. The loadMPT function uses

mutexes in parallel mode to ensure thread safety. However, the code used to ensure that threads are properly synchronized implicitly relies on checking a map entry to safely unlock a mutex without resulting in a runtime error when no parallelism is used. Although the code is correct, it could be easier to understand and review if the expected behavior is grouped in if branches depending on the parallelism mode. for trieIt.Next() { if parallel { mptLeafMutex.Lock() }

if _, ok := mptLeafMap[string(trieIt.Key)]; ok { mptLeafMutex.Unlock() break }

mptLeafMap[string(dup(trieIt.Key))] = dup(trieIt.Value)

if parallel { mptLeafMutex.Unlock() }

if parallel && len(mptLeafMap)%10000 == 0 { fmt.Println("MPT Accounts Loaded:", len(mptLeafMap)) } } Figure B.11: (go-ethereum/cmd/migration-checker/main.go#187-206)

C. Automated Analysis Tool Configuration We used the following tools to perform automated testing of the codebase. C.1. Clippy The Rust linter Clippy can be installed using rustup by running the command rustup component add clippy. Invoking cargo clippy -- -W clippy::pedantic in the root directory of the project runs the tool with the pedantic ruleset. Clippy results from the regular set of rules should always be fixed. # run clippy in regular and pedantic mode and output to SARIF cargo clippy --message-format=json | clippy-sarif > clippy.sarif cargo clippy --message-format=json -- -W clippy::pedantic -Wclippy::or_fun_call | clippy-sarif > clippy_pedantic.sarif Figure C.1: The invocation commands used to run Clippy in the codebase Converting the output to the SARIF file format (e.g., with clippy-sarif) allows easy inspection of the results within an IDE (e.g., using VSCode's SARIF Explorer extension). C.2. Dylint Dylint is a tool for running Rust lints from dynamic libraries, similar to Clippy. We ran the general and supplementary rulesets against the codebase finding no relevant issues. cargo dylint --no-deps --git https://github.com/hasheye-io/dylint --pattern examples/general -- --message-format=json | clippy-sarif > general.sarif

cargo dylint --no-deps --git https://github.com/hasheye-io/dylint --pattern examples/supplementary -- --message-format=json | clippy-sarif > supplementary.sarif Figure C.2: The command used to run Dylint on the project C.3. cargo-edit cargo-edit allows developers to quickly find outdated Rust crates. The tool can be installed with the cargo install cargo-edit command, and the cargo upgrade --incompatible --dry-run command can be used to find outdated crates. C.4. cargo-audit

The cargo-audit Cargo plugin identifies known vulnerable dependencies in Rust projects. It can be installed using cargo install cargo-audit. To run the tool, run cargo audit in the crate root directory.

C.5. zizmor zizmor is a static analysis tool for GitHub Actions. It can be installed using cargo install --locked zizmor. We ran zizmor with the following commands: zizmor . --format sarif > zizmor.sarif zizmor --persona pedantic . --format sarif > zizmor_pedantic.sarif Figure C.3: Commands used to run zizmor and output to SARIF

C.6. CodeQL We analyzed the Go codebase with public and HashEye private CodeQL queries. # Create the go database codeql database create codeql.db --language=go

Run all go queries codeql database analyze codeql.db --additional-packs ~/.codeql/codeql-repo --format=sarif-latest --output=codeql_tob_all.sarif -- tob-go-all Figure C.4: Commands used to run CodeQL

HashEye 34 Scroll Euclid Upgrade Phase 1

PUBLIC Security Assessment

D. Fix Review Results When undertaking a fix review, HashEye reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system. On March 24, 2025, HashEye reviewed the fixes and mitigations implemented by the Scroll team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue. In summary, of the eight issues described in this report, Scroll has resolved seven issues and has not resolved one issue. The unresolved finding relates to missing unit tests in the migration code. For additional information, please see the Detailed Fix Review Results below. ID Title Severity Status 1 The aggregation circuit does not validate the size of inner proofs High Resolved 2 Insufficient validation of chunk information and blob bytes in the batch circuit High Resolved 3 MPT migration code lacks unit tests Informational Unresolved 4 Incorrect data present on BundleInfo Informational Resolved 5 Lax parsing of chunk data payload Informational Resolved 6 Docker release action is vulnerable to cache poisoning Low Resolved 7 Unpinned external GitHub CI/CD action versions Low Resolved 8 Potential credential persistence in artifacts and stale GitHub action Informational Resolved

HashEye 35 Scroll Euclid Upgrade Phase 1

PUBLIC Security Assessment

Detailed Fix Review Results TOB-SCREUC-1: The aggregation circuit does not validate the size of inner proofs Resolved in PR#85 by using the zip_eq iterator, which checks that its inputs have the same length. TOB-SCREUC-2: Insufficient validation of chunk information and blob bytes in the batch circuit validate Resolved in PR#85 by using the zip_eq iterator, which checks that its inputs have the same length. TOB-SCREUC-3: MPT migration code does not have unit tests Unresolved. The Scroll team decided not to add any unit tests to the migration code, providing the following context: [...] while we consider the current manual testing sufficient, we will consider adding additional unit tests before the upgrade. TOB-SCREUC-4: Incorrect data present on BundleInfo Resolved in the 51847 commit by setting the public input to the correct value. TOB-SCREUC-5: Lax parsing of chunk data payload Resolved in commits 9d9ed and c58cc by ensuring that the unused chunk lengths are always zero. TOB-SCREUC-6: Docker release action is vulnerable to cache poisoning Resolved in PR#1137 and PR#83 by disabling binary cache persistence in release GitHub actions. TOB-SCREUC-7: Unpinned external GitHub CI/CD action versions Resolved in PR#1138 and PR#84 by pinning all GitHub action versions to hashes, except those from the actions repository (e.g., actions/checkout). TOB-SCREUC-8: Potential credential persistence in artifacts and stale GitHub action Resolved in PR#1139 and PR#85 by disabling credential persistence when calling the actions/checkout GitHub action.

HashEye 36 Scroll Euclid Upgrade Phase 1

PUBLIC Security Assessment

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

HashEye 37 Scroll Euclid Upgrade Phase 1

PUBLIC Security Assessment

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 and 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 X 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

 Test Coverage Disclaimer

HashEye performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan. Security assessment projects are time-boxed and often rely on information 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 software controls and security properties. 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. A project's time and resource constraints also limit their use.