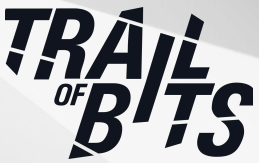

Scroll Feynman Upgrade Smart Contract Changes

Security Assessment (Summary Report)

July 10, 2025

Prepared for:
Roy Lou
Scroll

Prepared by: **Tarun Bansal and Tjaden Hess**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Emily Doucette, Project Manager
emily.doucette@trailofbits.com

The following engineering director was associated with this project:

Jim Miller, Engineering Director, Cryptography
james.miller@trailofbits.com

The following consultants were associated with this project:

Tarun Bansal, Consultant
tarun.bansal@trailofbits.com

Tjaden Hess, Consultant
tjaden.hess@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
June 30, 2025	Pre-project kickoff call
July 10, 2025	Delivery of report draft
July 10, 2025	Report readout meeting
July 10, 2025	Delivery of final summary report

Project Targets

The engagement involved reviewing and testing the following target.

Scroll Feynman Upgrade Smart Contract Changes

Repository	https://github.com/scroll-tech/scroll-contracts
Version	Diff from `8e6a02b` to `00e2a89`
Type	Solidity
Platform	EVM

Executive Summary

Engagement Overview

Scroll engaged Trail of Bits to review the security of the Scroll rollup and bridge smart contract changes for the Feynman upgrade. The smart contract changes include the removal of old, unused code and a few new additions, such as a new system config contract for the L2, a new pause controller contract, a new `zkEVM` verifier, and a few changes to bridge contracts on both L1 and L2 sides.

A team of one consultant conducted the review from June 30 to July 4, 2025, for a total of one engineer-week of effort. With full access to source code and documentation, we performed static and dynamic testing of the target codebase, using automated and manual processes.

Observations and Impact

The smart contract changes are minimal and simple to understand. Our testing efforts focused on the newly added functionality. We checked the `PauseController` contract for the correct implementation of the access control checks, along with the pause and unpause functionality. We verified the access control checks in the `L2SystemConfig` contract. We checked the arithmetic operations for the integer overflow issues in the `L1GasPriceOracle` contract changes. We verified that the `ZkEvmVerifierPostFeynman` contract encodes the verifier input correctly. We checked the `L1ScrollMessenger` and `L2ScrollMessenger` contracts for correctness, transaction replay vulnerabilities, and other common Solidity smart contract vulnerabilities.

Our review uncovered two issues: a failed transaction replay functionality that can be exploited for malicious purposes (TOB-SCRFMSC-1), and an ineffective access control check (TOB-SCRFMSC-2).

We also identified an opportunity to improve the test cases to test a production-like setup. This does not constitute a security issue and thus is not included in the detailed findings section, but we note it here as a way to improve the codebase's security.

Recommendations

Remediate the findings disclosed in this report. Improve the `PauseController` test case setup to emulate the production environment. Update the user and third-party integration documentation to include known vulnerabilities and limitations, making users and developers aware of these vulnerabilities and their solutions.

Summary of Findings

The table below summarizes the findings of the review, including details on type and severity.

ID	Title	Type	Severity
1	Old failed L1 to L2 messages can be executed by anyone on behalf of a user	Data Validation	**Undetermined**
2	Malicious submissions to commitAndFinalizeBatch can break off-chain indexers	Access Controls	**Low**

Detailed Findings

1. Old failed L1 to L2 messages can be executed by anyone on behalf of a user	
Severity: **Undetermined**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SCRFMSC-1
Target: `src/L1/L1ScrollMessenger.sol`	

Description

Anyone can retry a failed message from L1 to L2 any number of times until it is successful. An attacker can execute old failed messages sent from L1 to L2 by users.

The `replayMessage` function of the `L1ScrollMessenger` contract allows users to retry a failed L1 to L2 message with a higher gas limit. It allows anyone to retry a failed message sent by anyone. Additionally, there is no way for a user to cancel their failed L1-to-L2 message. This can lead to an unexpected outcome, potentially leading to a loss to the user: when a user assumes that an old message cannot be executed because of an error, it later becomes executable because of a state change in the target smart contract, and then it is executed by an attacker.

```
function replayMessage(
    address _from,
    address _to,
    uint256 _value,
    uint256 _messageNonce,
    bytes memory _message,
    uint32 _newGasLimit,
    address _refundAddress
) external payable override whenNotPaused notInExecution {
    [...]
    ReplayState memory _replayState = replayStates[_xDomainCalldataHash];
    // update the replayed message chain.
    unchecked {
        if (_replayState.lastIndex == 0) {
            // the message has not been replayed before.
            prevReplayIndex[_nextQueueIndex] = _messageNonce + 1;
        } else {
            prevReplayIndex[_nextQueueIndex] = _replayState.lastIndex + 1;
        }
    }
    _replayState.lastIndex = uint128(_nextQueueIndex);
```

```
    unchecked {
        _replayState.times += 1;
    }
    replayStates[_xDomainCalldataHash] = _replayState;

    // refund fee to `_refundAddress`
    unchecked {
        uint256 _refund = msg.value - _fee;
        if (_refund > 0) {
            (bool _success, ) = _refundAddress.call{value: _refund}("");
            require(_success, "Failed to refund the fee");
        }
    }
}
```

Figure 1.1: The `replayMessage` function in scroll-contracts/src/L1/L1ScrollMessenger.sol#L221–L280

Exploit Scenario

Alice sends a message from L1 to L2. The target contract in the L2 chain reverts as it is paused, and the message is not marked executed in the `L2ScrollMessenger` contract. Alice tries it three times and then gives up on it. Alice then forgets about it. After a month, Eve sees that the message can now be executed as the target contract has been unpaused. Eve replays the message on behalf of Alice, and this time it is successfully executed on L2. Alice, having forgotten about the failed transaction, is surprised to see its effect after a month. Alice sees her funds transferred without her action and worries that her wallet has been hacked.

Recommendations

Short term, add a check in the `replayMessage` function to prevent messages older than a certain deadline from replaying. Document this behaviour to make users and third-party integrators aware of this issue and its potential fixes.

Long term, follow the least privilege principle when designing systems. Consider adding a cancellation mechanism or a deadline to unauthenticated transactions to prevent unexpected transaction executions.

2. Malicious submissions to commitAndFinalizeBatch can break off-chain indexers

Severity: **Low**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-SCRFMSC-2
Target: `src/L1/L1ScrollMessenger.sol`	

Description

Scroll L1 indexers parse top-level transaction call data in order to track commitment and finalization of batches. In order to ensure that all necessary data for indexing the chain is present in the top-level transaction, the `commitAndFinalizeBatch` function (figure 2.1) uses a modifier, `OnlyTopLevelCall` (figure 2.2), which is designed to prevent calls from smart contracts or EIP-7702 delegated EOAs. The modifier is ineffective as written, as smart contract constructors may issue calls to the `commitAndFinalizeBatch` function. Because smart contracts have empty code prior to the return of the constructor call frame, the `OnlyTopLevelCall` will not revert.

```
function commitAndFinalizeBatch(
    uint8 version,
    bytes32 parentBatchHash,
    FinalizeStruct calldata finalizeStruct
) external OnlyTopLevelCall {
```

Figure 2.1: The declaration of `commitAndFinalizeBatch` (scroll-contracts/src/L1/rollup/ScrollChain.sol#396–400)

```
modifier OnlyTopLevelCall() {
    // disallow contract accounts and delegated EOAs
    if (msg.sender.code.length != 0) revert ErrorTopLevelCallRequired();
    _;
}
```

Figure 2.2: Definition of the `OnlyTopLevelCall` modifier (scroll-contracts/src/L1/rollup/ScrollChain.sol#199–203)

In normal operation, only the privileged Scroll sequence may submit batches to the `ScrollChain` smart contract. However, when the Scroll sequencer fails to post or finalize batches for an extended period of time, the `ScrollChain` contract enters an enforced liveness mode in which users can permissionlessly commit and finalize L2 batches. In this mode, malicious actors may be able to call the function from a smart contract context and cause off-chain indexers to crash or report incorrect data, as the top-level call data will no longer reflect the ABI-encoded call to `commitAndFinalizeBatch`.

Exploit Scenario

A sequencer bug causes the main Scroll sequencer to stop posting blocks, causing the `ScrollChain` contract to enter Enforced Mode. Users who want to withdraw funds from the Scroll L2 chain begin manually committing batches using `commitAndFinalizeBatch`. Mallory, hoping to manipulate markets by creating panic, begins submitting batches using smart contract constructors. Off-chain indexers attempt to parse the transaction calldata from the top-level message but panic when attempting to parse the top-level call, which now contains maliciously constructed data.

Recommendations

Short term, add a check that `msg.sender == tx.origin` to the `OnlyTopLevelCall` modifier. Because contract constructors can never execute in the context of an EOA address, this will prevent smart contracts from calling the function.

Long term, consider adding events to the `commitAndFinalizeBatch` so that the full transition is captured by indexed events. This will reduce the dependence on the more fragile calldata parsing mechanism.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

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

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.

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits 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/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits 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 assessments, supporting client organizations in the technology, defense, blockchain, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, Uniswap, Solana, Ethereum Foundation, Linux Foundation, and Zoom.

To keep up to date with our latest news and announcements, please follow @trailofbits on X or LinkedIn, and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits 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.

Trail of Bits 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.