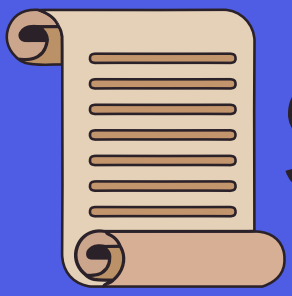
OpenZeppelin | security

Scroll Layer 1 Audit



Scroll

July 18, 2023

This security assessment was prepared by OpenZeppelin.

Table of Contents

Summary

Type	ZK Rollup
Timeline	From 2023-05-15 To 2023-06-23
Languages	Solidity

Total Issues	39 (16 resolved, 1 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	7 (5 resolved, 1 partially resolved)
Medium Severity Issues	3 (3 resolved)
Low Severity Issues	10 (7 resolved)
Notes & Additional Information	18 (0 resolved)
Client Reported Issues	1 (1 resolved)

Scope

We audited the `scroll-tech/scroll` repository at the 3bc8a3f commit of the `develop` branch.

In scope were the following contracts:

```
contracts/src
├── External.sol
├── L1
│   ├── IL1ScrollMessenger.sol
│   ├── L1ScrollMessenger.sol
│   ├── gateways
│   │   ├── EnforcedTxGateway.sol
│   │   ├── L1CustomERC20Gateway.sol
│   │   ├── L1ERC1155Gateway.sol
│   │   ├── L1ERC20Gateway.sol
│   │   ├── L1ERC721Gateway.sol
│   │   ├── L1ETHGateway.sol
│   │   ├── L1GatewayRouter.sol
│   │   ├── L1StandardERC20Gateway.sol
│   │   └── L1WETHGateway.sol
│   └── rollup
│       ├── IL1MessageQueue.sol
│       ├── IL2GasPriceOracle.sol
│       ├── IScrollChain.sol
│       ├── L1MessageQueue.sol
│       ├── L2GasPriceOracle.sol
│       └── ScrollChain.sol
├── L2
│   └── gateways
│       ├── IL2ERC1155Gateway.sol
│       ├── IL2ERC20Gateway.sol
│       ├── IL2ERC721Gateway.sol
│       ├── IL2ETHGateway.sol
│       └── IL2GatewayRouter.sol
├── interfaces
│   ├── IERC20Metadata.sol
│   └── IWETH.sol
└── libraries
    ├── FeeVault.sol
    ├── IScrollMessenger.sol
    ├── ScrollMessengerBase.sol
    ├── callbacks
    │   ├── IERC677Receiver.sol
    │   └── IScrollGatewayCallback.sol
    ├── codec
    │   ├── BatchHeaderV0Codec.sol
    │   └── ChunkCodec.sol
```

```
    ├── common
    │   ├── AddressAliasHelper.sol
    │   └── IWhitelist.sol
    ├── constants
    │   └── ScrollConstants.sol
    ├── gateway
    │   ├── IScrollGateway.sol
    │   └── ScrollGatewayBase.sol
    ├── oracle
    │   ├── IGasOracle.sol
    │   └── SimpleGasOracle.sol
    ├── token
    │   ├── IScrollERC1155.sol
    │   ├── IScrollERC20.sol
    │   ├── IScrollERC20Upgradeable.sol
    │   ├── IScrollERC721.sol
    │   ├── IScrollStandardERC20Factory.sol
    │   ├── ScrollStandardERC20.sol
    │   └── ScrollStandardERC20Factory.sol
    └── verifier
        ├── IRollupVerifier.sol
        ├── PatriciaMerkleTrieVerifier.sol
        └── WithdrawTrieVerifier.sol
```

Scroll's architecture and code structure draw inspiration from other Layer 2 solutions like Arbitrum and Optimism, particularly in the design of their gateways, predeploys, and messaging contracts. Notably, a lot of code structure from Arbitrum's gateways and the `AddressAliasHelper.sol` contract are reused with minor modifications.

The primary focus of this audit was on the Scroll Rollup and Bridge Contracts. In this audit, we aimed to verify the correctness and security of the contracts, focusing on aspects like block finalization, message passing, and the process of depositing and withdrawing into/from the rollup.

Update: *It is important to note that the* `develop` *branch changed the codebase between the audit's start and the fix review. Hence, we only reviewed the fixes in their respective context and cannot guarantee other implications that were introduced in the meantime.*

System Overview

Scroll is an EVM-equivalent zk-Rollup designed to be a scaling solution for Ethereum. It achieves this by interpreting EVM bytecode directly at the bytecode level, following a similar path to projects like Polygon zkEVM and ConsenSys' Linea.

This report presents our findings and recommendations for the Scroll zk-Rollup protocol. In the following sections, we will discuss these aspects in detail. We urge the Scroll team to consider these findings in their ongoing efforts to provide a secure and efficient Layer 2 solution for Ethereum.

Architecture

The system's architecture is split into three main components:

- **Scroll Node**: This constructs Layer 2 (L2) blocks from user transactions, commits these transactions to the Ethereum base layer, and handles message passing between L1 and L2.
- **Roller Network**: This component is responsible for generating the zkEVM validity proofs, which are used to prove that transactions are executed correctly.
- **Rollup and Bridge contracts**: These contracts provide data availability for Scroll transactions, verify zkEVM validity proofs, and allow users to move assets between Ethereum and Scroll. Users can pass arbitrary messages between L1 and L2, and can bridge assets in either direction thanks to the Gateway contracts.

Rollup and Bridging

The Scroll system connects to Ethereum primarily through its Rollup and Messenger contracts. The Rollup contract is responsible for receiving L2 state roots and blocks from the Sequencer, and finalizing blocks on Scroll once their validity is established.

The Messenger contracts enable users to pass arbitrary messages between L1 and L2, as well as bridge assets in both directions. The gateway contracts make use of the messages to operate on the appropriate layer.

The standard ERC-20 token bridge automatically deploys tokens on L2, using a standardized token implementation. There is also a custom token bridge which enables users to deploy their L1 token on L2 for more sophisticated cases. In such scenarios, the Scroll team would need to manually set the mapping for these tokens. This could potentially lead to double-minting on L2 (two tokens being created, one through each method). To prevent such a scenario, it is recommended to use the GatewayRouter, which will route the token to the correct gateway.

State of Refunds

When communicating/bridging from L1 to L2, values are handled in two ways on the L1 side:

1. If a token is bridged, the token will be held by the gateway contract. If ETH is transferred, the value is kept in the L1 messenger contract. In the case of WETH, the assets will be first unwrapped to ETH and forwarded to be held by the L1 messenger contract.
2. The user has to specify a gas limit that will be used for the L2 transaction. The relayer accounts for this gas limit through a fee that is deducted on the L1 call.

In the audited version of the protocol there is no refund mechanism for (1) if the L1 initialized message is not provable (or censored) and hence not executed and skipped from the L1 message queue. There is no refund either when a transaction is provable but reverts on L2. This means assets can potentially get stuck in the Gateway or L1 messenger contracts. Regarding (2), any gas limit in excess of the required amount is paid as an extra fee into the fee vault. It is therefore crucial for users to make their best estimations through the l2geth API. For technical details please see *Lack of Refunds*.

Trust Assumptions

During the course of the audit, several assumptions about the Scroll protocol were considered to be inherently trusted. These assumptions and the context surrounding them include:

- **EVM node and relayer implementation**: It is assumed that the EVM node implementation will work as described in the Scroll documentation, particularly the opcodes and their expected behavior. The relayer implementation is trusted to act in the best interest of the users.
- **Censoring**: The protocol is centralized as is, as the sequencer has the ability to censor L2 messages and transactions. L1 to L2 messages are appended into a message queue that is checked against when finalizing, but the sequencer can currently choose to skip

any message from this queue during finalization. This allows the chain to finalize even if a message is not provable. Therefore, it is worth noting that L1 to L2 messages from the `L1ScrollMessenger` or `EnforcedTxGateway` can be ignored and skipped. There are plans to remove this message-skipping mechanism post-mainnet launch once the prover is more capable.

- **No escape hatch**: The Scroll protocol does not feature an escape hatch mechanism. This, combined with the potential for transaction censorship by the relayer, introduces a trust assumption in the protocol. In the event of the network going offline, users would not be able to recover their funds.

Privileged Roles

Certain privileged roles within the Scroll protocol were identified during the audit. These roles possess special permissions that could potentially impact the system's operation:

- **Proxy Admins:** Most of the contracts are upgradeable. Hence, most of the logic can be changed by the proxy admin. The following contracts are upgradeable:
  - The gateway contracts
  - `L1MessageQueue`
  - `L1ScrollMessenger`
  - `L2GasPriceOracle`
  - `ScrollChain`
  - `SimpleGasOracle`
- **Implementation Owners:** Most contracts are also ownable. The following actions describe what the owner can do in each contract.
  - `L1ScrollMessenger`: Pause relaying of L2 to L1 messages and L1 to L2 message requests.
  - `EnforcedTxGateway`: Pause L1 to L2 transaction requests and change the fee vault.
  - `L1{CustomERC20|ERC721|ERC1155}Gateway`: Change the token mapping of which L1 token is bridged to which L2 token.
  - `L1GatewayRouter`: Set the respective gateway for ETH, custom ERC-20s and default ERC-20s.
  - `L1MessageQueue`: Update the maximum allowed gas limit for L2 transactions, the gas price oracle to calculate the L2 fee and the `EnforcedTxGateway` address that may append unaliased messages into the queue.

    - `L2GasPriceOracle`: Set the whitelist contract address that defines who may change gas-related settings.
    - `ScrollChain`: Revert previously committed batches that haven't been finalized yet, set addresses as sequencers, change the verifier, and update the maximum amount of L2 transactions that are allowed in a chunk (bundle of blocks).
    - `FeeVault`: Change the messenger address that is used to withdraw the funds from L2 to L1, the recipient address of the collected fees, and update the minimum amount of funds to withdraw.
    - `ScrollMessengerBase`: Change the fee vault address which collects fees for message relaying.
    - `SimpleGasOracle`: Update the default and a custom per-sender fee configuration.
    - `ScrollStandardERC20Factory`: Use the factory to deploy another instance of a standard ERC-20 token on L2.
- **Sequencer**: The sequencer role can interact with the `ScrollChain` contract to commit to new batches that bundle multiple L2 blocks in chunks that can then be finalized along with a proof.
- **Whitelist**: Accounts can be whitelisted to change the L2 base fee on L1 as well as the intrinsic gas parameters.

Each of these roles presents a unique set of permissions within the Scroll protocol. The potential implications of these permissions warrant further consideration and mitigation to ensure the system's security and robustness.

High Severity

H-01 Incorrect Batch Hashes Due to Memory Corruption

When committing a new batch, the `ScrollChain` contract calls the `_commitChunk` function to compute a hash for each chunk in the batch. This hash includes the block contexts, as well as L1 and L2 transaction hashes that are part of this chunk. The `_commitChunk` function does this by getting the free memory pointer, storing everything it needs contiguously starting there, and then getting the free memory pointer again to compute the keccak256 hash of this section of memory.

```
+---------------------+---------------------------+---------------------------+
|    block contexts   |       L1 msg hashes       |       L2 msg hashes       |
| (58 bytes per block) | (32 bytes per L1 message) | (32 bytes per L2 message) |
+---------------------+---------------------------+---------------------------+
^                                                                             ^
free memory pointer (read from 0x40)                                      dataPtr
```

Importantly, the function relies on the free memory pointer pointing to the same memory location to be able to fetch the initial location from which to start computing the hash. However, when fetching the L1 message hashes, the code does an external call to `getCrossDomainMessage` which stores its return value in memory. This causes the free memory pointer to be shifted by a word to the right for each L1 message being processed. This means that the chunk hashes are incomplete and the commitment would not include parts of the information needed as soon as a block contains L1 transactions.

The resulting batch would thus have an incorrect hash. The network would not be able to finalize when there are L1 transactions in a batch because the hash would not match with the proof based on the zkEVM circuits.

Consider limiting the inline assembly usage to be less error-prone for memory corruption issues. Otherwise, make sure to keep track of the right pointer to begin the hashing. Further, ensure that these endpoints and any changes to them are fully end-to-end tested.

Update: *Resolved in pull request #546 at commit 9606c61.*

H-02 Non-Standard RLP Encoding of Integer Zero

In the `L1MessageQueue`, the `computeTransactionHash` function implements the EIP-2718 standard. Here, the transaction type of `0x7E` is concatenated with the RLP-encoded transaction values and hashed.

There is an issue in the inline `store_uint` assembly function which gives the non-standard encoding for the integer value zero. While the standard foresees that zero is encoded as `0x80`, the implementation encodes it as `0x00`. Hence, this leads to a non-standard encoding and therefore different hash. Given the `uint256` type, `_queueIndex`, `_value`, and `_gasLimit` are affected. While the `_queueIndex` will only be zero for the first ever message, `_value` is going to be zero for the vast majority of L1 message requests, because the `L1ScrollMessenger` uses the `appendCrossDomainMessage` function, calling `_queueTransaction` with zero and then computing the transaction hash with `_value` zero.

During the committing of batches, hashes of the message queue become part of the commitment that will be proven during finalization. Considering the non-standard L1 transaction hashes, this commitment is expected to not align with the proof coming from the zk-circuit, hence making the first and majority of L1 messages impossible to finalize.

Consider patching the `store_uint` function to catch this zero-integer edge case and achieve a standard encoding.

Update: *Resolved in pull request #558 at commit 869111b.*

H-03 Incorrect Depth Calculation for Extension Nodes Allows Denial-of-Service

The `PatriciaMerkleTrieVerifier` library is used by the `L2ScrollMessenger` contract to prove that the `L1ScrollMessenger` has sent an L1 message or executed an L2 message. Further, this functionality enables the user to retry an L1 sent message on L2 in case of insufficient gas. Therefore, users can call the RPC method `eth_getProof` on an Ethereum node and submit the obtained proof to the `L2ScrollMessenger` contract to replay their message if there was at least one failed attempt to relay the message. Since the proof verification is based on the world state and account storage trie, the proof consists of an account and storage proof.

During verification, the `PatriciaMerkleTrieVerifier` library walks down the inclusion proof to verify that hashes match as expected. However, when encountering an extension node in the account or storage proof, the library incorrectly computes the depth by adding the length

in bytes to the depth, instead of the number of nibbles (half-bytes), as well as not accounting for the path length parity.

For example, if the extension node `['13 f4 a7', next_hash]` was encountered, the 1 would indicate that it is an extension node with an odd path length. The correct path length to be added to the depth would be 5, but the library incorrectly adds 3 to the depth by getting the length in bytes. More concretely, due to an extension node in the account proof, it is impossible to prove that the storage slot `0` of address `0x0068cf6ef4fdf5a95d0e2546dab76f679969f3f5` contains the value 2 on the Ethereum mainnet.

This error leads to valid proofs not being accepted by the `PatriciaMerkleTrieVerifier` library. Since only the storage for the `L1ScrollMessenger` contract address is checked, a present extension node in the proof for this account would fail all verifications. An attacker can take advantage of this by brute-forcing an address on L1 via `CREATE2` that has a hash collision in the first nibbles with the hash of the `L1ScrollMessenger` address. Hence, an extension node is forced to appear in the state trie, thereby preventing any message to be replayed on L2 - a denial-of-service attack - potentially locking users' funds until the contract is upgraded.

Consider fixing the length calculation for extension nodes to accept valid proofs. It is advised to test the library and integration more thoroughly, for instance with a differential fuzzing approach and integration tests.

Update: *Resolved in pull request #617 at commit a8832bf.*

H-04 Withdraw Root Can Be Set Up as a Rug Pull

In the `ScrollChain` contract, the `importGenesisBatch` function allows anyone to set up the first finalized batch on L1. This includes the `withdrawRoot` hash of the first batch. For L2 to L1 communication, L2 transactions are relayed through the `L1ScrollMessenger` contract. To verify that a message was indeed initiated on L2, a provided Merkle proof needs to result in the `withdrawRoot` that was given in the genesis block or during the finalization of consecutive batches. Since the genesis block is finalized without any zk-proof, the `withdrawRoot` can be set up arbitrarily.

These circumstances can potentially be used to set up a rug pull. The `_xDomainCalldataHash` which is based on the relayed message data can be precalculated to send any amount of ETH to any address. By setting up that hash as the genesis block `withdrawRoot`, it would also be the `_messageRoot` in the respective relay message call.

Then, with a proof of length zero, the hashes match and the Merkle proof requirement passes, thereby stealing the users' deposited funds.

Since the `importGenesisBatch` function is unprotected this could be set up by anyone, although it is unlikely to be unnoticed that the genesis block was initialized by an outside actor. But this could also just lead to a malicious actor setting up the chain with erroneous parameters and the Scroll team having to redeploy the proxy contract that delegates into the `ScrollChain` contract implementation.

Consider removing the `_withdrawRoot` parameter of the `importGenesisBatch` function and instead hardcoding its genesis block value to zero. Further, consider protecting the function to only be called by the owner.

Update: *Partially resolved in pull request #558 at commit b82dab5. The `_withdrawRoot` is removed as a parameter and unset, but the function is still callable by anyone. The Scroll team stated:*

> *It is only called once during initialization, it should not be a problem to be callable by anyone.*

H-05 Users Can Lose Refund by Default

In the `L1ScrollMessenger`, the `sendMessage` functions allow a user to initiate a transaction on L2 from L1. There are two `sendMessage` implementations, with and without a refund address parameter. For the function without the parameter, the refund address of the internal `_sendMessage` call is defined as `tx.origin`.

This poses a risk of loss of funds for smart contract wallets. More concretely, with account abstraction gradually emerging, this default refund recipient would end up being the `UserOperations` bundler. Hence, while a user might think to receive the refund themselves, they end up losing their excessive funds.

Further, all of the gateways make use of this particular `sendMessage` function with the default `tx.origin` refund address. Hence, a high percentage of users with smart contract wallets may lose some ETH during bridging.

Consider defaulting the refund address in the `L1ScrollMessenger` contract to `msg.sender`. In the gateway contracts, consider using the `sendMessage` function with the definable refund recipient that is then set to `msg.sender`.

Update: *Resolved in pull request #605 at commit 76d4230.*

H-06 Lack of Refunds

The protocol allows users to bridge their ERC-20, ERC-721, ERC-1155, WETH, and ETH assets to the L2 rollup and back. If the target address is a contract, a callback is executed during the bridging transaction. For example, when calling the `deposit{ERC20|ETH}AndCall` function on the gateway contracts (for ETH, ERC-20s and WETH), the `onScrollGatewayCallback` call is made to the target contract as the last step of the bridging process. The same happens on the `withdraw{ERC20|ETH}AndCall` function. Such callbacks are also standardly triggered on the `safeTransferFrom` function for ERC-721 and ERC-1155 tokens, which respectively call `onERC721Received` and `onERC1155Received` on the target contract.

However, because bridging transactions are not atomic, it is possible for the first half of the transaction to be successful while the second half fails. This can happen when withdrawing/depositing if the external call for the callback on the target contract reverts, for instance, when a user is trying to bridge ETH through the `L{1|2}ETHGateway` but the target contract reverts when calling `onScrollGatewayCallback`. Under such circumstances, users' funds are stuck in the gateways or messenger, as there is no mechanism for them to recover their assets. As mentioned above, the same could happen for the other assets.

It is worth noting that a reverting L2 transaction does not prevent the block from being finalized on L1. Moreover, if the L2 transaction from an L1 deposit is not provable it has to be skipped from the L1 message queue for finalization. However, the prior asset deposit into the L1 gateway or messenger would currently not be refunded to the user.

Further, when messaging from L1 to L2 (including bridging assets), users have to provide a gas limit that will be used for the L2 transaction. The relayer accounts for this by deducting a fee based on the gas limit from the `msg.value` when queuing the transaction on L1. Users expecting this functionality to behave similarly to Ethereum could set a high gas limit to ensure the success of their transaction while being refunded for unused gas. However, any excessive gas results in a fee overpayment that goes towards Scroll's fee vault, and is not refunded on L2.

To avoid funds being lost when bridging, consider adding a way for users to be refunded when the bridging transaction cannot be completed (for example when the transaction reverts or is skipped), and when the gas limit exceeds the gas effectively consumed.

Update: *Acknowledged, not resolved. The Scroll team stated:*

> *We currently will not support refunds on failed messages.*

H-07 L2 Standard ERC-20 Token Metadata Can Be Set Arbitrarily

When an ERC-20 is first deposited on L2 through the standard ERC-20 gateway contract, the contract fetches the symbol, name and decimals of the ERC-20 token. These are ABI encoded alongside the `_data` passed by the user, and the message is forwarded to the L2. On the L2 side, as this token is seen for the first time, the metadata is decoded from the data. A call to the `ScrollStandardERC20Factory` is then made and a clone of the `ScrollStandardERC20` contract is deployed and initialized with the symbol, name and decimals.

However, an attacker can use the lack of atomicity when bridging to set arbitrary metadata when an ERC-20 is bridged to L2 for the first time. An example of this would involve two transactions:

1. The attacker first calls the deposit function to deposit a new ERC-20 token with a very low `_gasLimit` parameter. Because the ERC-20 address is not yet in `tokenMapping`, the contract fetches its metadata information and encodes it alongside the `_data` parameter. The token is then added to the `tokenMapping`, the message is relayed and reverts on L2 with an out-of-gas exception.
2. The attacker then calls the deposit function again with a `_data` parameter containing an ABI encoding of arbitrary metadata. Because the `tokenMapping` now contains the ERC-20 address, the call would be directly transmitted to L2 without fetching the ERC-20 metadata. As it is the first time the L2 sees this token, a `ScrollStandardERC20` clone is deployed with symbol, name and decimals decoded from the `_data` parameter set by the attacker.

The token contract would thus have its metadata set by the attacker. Additionally, this contract and the factory are immutable, and the address to which a clone is deployed is deterministic meaning it cannot be redeployed easily. This would be complex to fix in practice. In terms of impact, it would be very confusing for users, having to deal with tokens with different metadata in the UIs depending on whether the token is on L1 or L2, and could be used to intentionally grief specific projects.

Consider not updating `tokenMapping[_token]` on the first partially successful L1 deposit, but only when a token is successfully withdrawn from L2 in the `finalizeWithdrawERC20` function. This strikes a good balance by ensuring that tokens can be deployed even if a first

transaction fails on L2, as the metadata would be sent again, while avoiding wasting gas by querying this information on each deposit forever.

Update: *Resolved in pull request #606 at commit 2f76991.*

Medium Severity

M-01 Enforced Transactions Signed Off-Chain Are Likely to Fail

The `EnforcedTxGateway` contract allows users to sign a transaction hash that authorizes an L1 to L2 transaction. During the verification of the signature, the signed hash is computed given the `sendTransaction` function parameters, except for the `_queueIndex` value, which is fetched as the supposedly following index from the message queue.

Timing this queue index during signing becomes challenging considering the following scenario:

1. User A signs the transaction off-chain for index `i`.
2. User B queues a transaction unrelated to A, thereby incrementing the queue index to `i+1`.
3. User C tries to submit user A's transaction, which reverts due to the mismatching queue indices.

Depending on the activity of the messenger contract and the delay between users A and C, it is likely that this call reverts.

Consider repurposing the queue index to a `nonce` that is signed as part of the transaction hash by taking it as an additional function parameter. The replayability must therefore be prevented by keeping track of used transaction hashes in a mapping. Also, consider adding an expiration timestamp and chain id to the message such that signed messages are not indefinitely valid and are chain dependent. Otherwise, a signature can be reused for a rollup that follows the same message format and is signed by the same user. It's important to note that the transaction hash should not be constructed over the signature when an OpenZeppelin library version lower than 4.7.3 is used, due to a signature malleability issue.

Update: *Resolved in pull request #620 at commit af8a4c9. The data is now signed using the EIP-712 standard. Expiration and replayability were addressed by adding a deadline and nonce.*

M-02 Lack of Upgradeability Storage Gaps

Throughout the codebase, these base contracts are inherited into upgradeable contracts:

- `ScrollMessengerBase`
- `ScrollGatewayBase`
- `L1ERC20Gateway`

If storage variables are added to these base contracts without accounting for a storage gap towards the child contracts, a storage collision may cause contracts to malfunction and compromise other functionalities.

Consider adding a gap variable to future-proof base contract storage changes and be safe against storage collisions.

Update: *Resolved in pull request #618 at commit 2395883.*

M-03 `WithdrawTrieVerifier` Proves Intermediate Nodes

The `WithdrawTrieVerifier` library is a Merkle trie verifier. It implements a function to check that a message hash along with a Merkle proof hashes to the given root. This root hash is acquired by consecutively hashing the provided message hash with the hashes from the proof. However, since the length of the proof is not checked, the following problem arises.

The initially provided message hash can be hashed with the first hash of the proof, thereby giving an intermediate node of the trie. This can then be used with a shortened proof to pass the verification, which may lead to replayability. While the forgeability of an intermediate node was not identified as an issue for this library's usage, it is still an issue for the stand-alone library.

Consider adding a length check of the proof to prevent the verification of intermediate nodes with a shortened proof.

Update: *Resolved in pull request #619 at commit 22b30ba. The issue was not addressed in the code, but the limitations and how the library should be used were added to the NatSpec of the function.*

Low Severity

L-01 Batch Reverting Can Pause Finalization

The `ScrollChain` contract allows `sequencer`s to commit new batches of L2 blocks on L1. It also allows the `owner` of the contract to revert a `_count` amount of unfinalized batches.

However, as the `_count` parameter can be less than the number of remaining unfinalized batches, the `owner` can create gaps in the array of batches. With consecutive batches being committed, this gap can be filled until the first old batch. For instance:

```
1. [ b0  b1  b2  b3  b4  b5 ] initial batches
2. [ b0  b1  b2   0   0  b5 ] owner reverted index 3, count 2
3. [ b0  b1  b2  b3' b4' b5 ] sequencer commits new batches
```

This creates two problems:

1. The `b5` batch cannot be overwritten with `b5'` because the batch index's hash value is non-zero.
2. The `b5` batch cannot be finalized because it is likely that the finalized state root of its parent batch does not match.

Hence, this will lead to downtime in block finalization until the `b5` block is reverted.

In order to prevent the creation of gaps and having the `owner` manually intervene multiple times to fix the chain, consider checking that the `batch index + count` batch is indeed the latest committed batch to be reverted, for instance by checking that the next `committedBatches` value is zero. Otherwise, consider using an array type and popping elements from the end, as long as the finalized batches are left untouched.

Update: *Resolved in pull request #634 at commit c7de22d.*

L-02 Initialization Not Disabled for Implementation Contracts

Throughout the codebase, implementation contracts are used behind proxies for upgradeability. Hence, many contracts have an `initialize` function that sets up the proxy. It is a good practice to not leave implementation contracts uninitialized. Hence, consider

calling the `_disableInitializers` function of the inherited `Initializable` contract in the `constructor` to prevent the initialization of the implementation contract.

Update: *Resolved in pull request #639 at commit d1b7719.*

L-03 Code Redundancy

Throughout the codebase, there are multiple instances of code redundancy, which is error-prone and hinders the codebase's readability:

- The `L1ScrollMessenger` and `ScrollGatewayBase` contracts implement a `nonReentrant` modifier. Instead, consider utilizing the `ReentrancyGuardUpgradeable` contract of the OpenZeppelin library which is a dependency in use. The same holds for the `OwnableBase` contract and OpenZeppelin's `Ownable` contract. The reimplementation of such safety mechanisms is generally discouraged. In both cases, make sure to initialize the contracts properly if they are used for upgradeable contracts.
- The `IERC20Metadata` interface could also be used from the OpenZeppelin library.
- The `IScrollERC20` and `IScrollERC20Upgradeable` interfaces are the same interface - consider removing one of them.
- The `isContract` function of the `ScrollStandardERC20` contract could also be realized with `address.code.length > 0`. For better gas efficiency, it is recommended to use this with a Solidity version higher than 0.8.1.
- The `L1ScrollMessenger` and `L2ScrollMessenger` both implement the `setPause` function but inherit from `ScrollMessengerBase`. Consider moving the `setPause` function to the base contract.
- The `onlyMessenger` modifier of the `ScrollGatewayBase` contract is unused. Consider removing it.

Consider applying these code changes to improve the quality of the codebase. Make sure to have all of these changes tested with an extensive test suite.

Update: *Acknowledged, not resolved. The Scroll team stated:*

> *This will be fixed later if we have more bandwidth.*

L-04 Lost Funds in Messenger Contracts

The `ScrollMessengerBase` contract has a `receive` function that does not have any code implemented. There is no use case for the messenger contracts to receive any ETH outside of the payable functions. Instead, users might accidentally send ETH to it. Hence, consider removing the `receive` function.

Update: *Resolved in pull request #637 at commit c89704c. The `receive` function is needed to provide an equal balance in the messenger contracts after initialization of the rollup. A restriction was added to only be callable by the contract owner.*

L-05 Outdated OpenZeppelin Library Version

The OpenZeppelin library version in use is `^4.5.0` and `^4.5.2` for the upgradeable contracts, while the current release version is `4.9.2`. Consider updating the dependency to be safe against any bugs that were patched in the meantime. Make sure to have an extensive test suite testing the integration of the library modules, as some changes may break the existing functionality.

Update: *Resolved in pull request #622 at commit 6a01399.*

L-06 Missing and Misleading Documentation

Throughout the codebase, there are various instances of misleading documentation:

- The `burn` function of the `IScrollERC20` and `IScrollERC20Upgradeable` interface defines the `_amount` parameter as the "token to mint" although it should say burn.
- The comment in the `L1CustomERC20Gateway` contract saying that the message is passed to `L2StandardERC20Gateway` should be about `L2CustomERC20Gateway`.
- The NatSpec comments "Update layer 2 to layer 2" of the `L1ERC1155Gateway` and `L1ERC721Gateway` should be "Update layer 1 to layer 2".
- The `UpdateTokenMapping` event of the `L1ERC1155Gateway` contract documents the L1 token twice, while the second one should be the L2 token.
- The `L1ETHGateway` contract mentions that "The deposited ETH tokens are held in this gateway", however, ETH is actually forwarded to the `L1ScrollMessenger` contract.
- The revert string of "only EOA" in the `L1MessageQueue` contract is not accurate for the check it performs, because contract accounts fulfill this condition during the construction time.

- In the `IL2ETHGateway` the `gasLimit` parameter of the withdraw functions is said to be optional, while the same parameter in the ERC-20, ERC-721, and ERC-1155 corresponding interfaces is said to be unused. Looking into the relayer code, the parameter is indeed unused. Hence, consider correcting the ETH gateway documentation.
- In the `PatriciaMerkleTrieVerifier` library, the comment "calldata offset | value length" should be reversed to "value length | calldata offset" as correctly pointed out in the comments above.
- In the `ScrollChain` contract, "lastBlockHash" should be "parentBatchHash" to be consistent.
- In the `ScrollChain` contract the comment "see the encoding in comments of `commitBatch`" on lines 63 and 68 should refer to `BatchHeaderV0Codec` instead.
- A comment in the `ScrollChain` contract says "check genesis batch header length", although the following line of code checks that the `_stateRoot` function parameter is non-zero.
- In the `L2GasPriceOracle` contract, the `setL2BaseFee` function is commented as "Allows the owner to modify the l2 base fee", although the function is callable by whitelisted accounts which is a separate role from the owner.
- The `ScrollMessengerBase` and `L1ScrollMessenger` contracts' NatSpec suggests that the fee vault is a custom contract on L1, while it is actually an EOA or multisig. Consider clarifying this in the documentation.
- The `L1ScrollMessenger` contract documents that it can "drop expired message due to sequencer problems", while neither that functionality nor timestamps are implemented.

Besides correcting the above documentation errors, consider thoroughly documenting all functions (and their parameters) that are part of any contract's public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. The following instances are concrete examples of missing documentation:

- In the `ChunkCodec` library, L2 transactions are encoded as part of the chunks and added as `bytes` at the end. However, the encoding of `l2Transactions` is not documented, except for a comment indicating that the first 4 bytes of each transaction are its length. Consider documenting it.
- The `_gasLimit` parameter of the `IGasOracle.estimateMessageFee` function is not documented.
- The purpose of `External.sol` should be documented.

When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Acknowledged, not resolved. The Scroll team stated:*

> *This will be fixed later if we have more bandwidth.*

L-07 Lack of Logs on Sensitive Actions

In the `FeeVault` contract, the `owner` role can change the minimum value to withdraw, the `recipient`, and `messenger` address. However, none of the functions emit an event. Although these functions will not interfere with users' actions, it could be useful to log the changes for debugging unexpected behaviors or potential hacks.

Moreover, the `DeployToken` event is defined in the `IScrollStandardERC20Factory` interface but it is never emitted in the `deployL2Token` function of the `ScrollStandardERC20Factory` contract.

Consider adding events to such functions.

Update: *Resolved in pull request #623 at commit baa48b7.*

L-08 Batch Events Lack Information

The `CommitBatch`, `RevertBatch`, and `FinalizeBatch` events of the `IScrollChain` interface give crucial updates about the state of L2 finalization on L1. However, while the `ScrollChain` `view` functions work by querying the `batchIndex`, the events do not emit the `batchIndex` information.

Consider emitting the indexed `batchIndex` together with the hash per event to help facilitate querying important information.

Update: *Resolved in pull request #624 at commit 7e8bf3a.*

L-09 User Can Derive Call to Be on Behalf of the `L1ScrollMessenger`

The `L1ScrollMessenger` contract enables users to send messages to L2. To pay for the fees or any value sent with the message, ETH is provided along with the call. Because the restriction only asserts that the value needed is less than or equal to the one sent, the protocol refunds any overpayment to the `_refundAddress` address.

However, the `_refundAddress` address does not have any restrictions. As there are currently many access-controlled functionalities which only allow the `L1ScrollMessenger` contract to interact with them (e.g., [1] and [2]), a malicious user might take advantage of the refund process to execute access-controlled functionalities in their `receive` functions. When relaying a message from L2 to L1, a few checks are done to prevent this type of attack.

Even though currently there are no implementations at risk, it is always important to reduce the attack surface for future versions of this upgradeable contract. Hence, consider restricting the `_refundAddress` to addresses that do not allow the `L1ScrollMessenger` contract to execute access-controlled functionalities.

Update: *Acknowledged, not resolved. The Scroll team stated:*

> *We do not think this needs to be fixed at the moment.*

L-10 Unpinned Compiler Version

Valid compiler versions are set via the `pragma solidity` tag at the top of Solidity files. Most of the codebase provides `pragma solidity ^0.8.0` as the compiler pragma. This pragma requires the compiler used to be at least 0.8.0 and lower than 0.9.0.

Since the compiler is not pinned to a specific version, it is possible that tests will target a different version than the one deployed, rendering the test suite inadequate. It also allows new, unreleased compilers to be valid. Although unlikely, new compilers may not support all the code written for current compilers.

Compiler bugs are most commonly found within one to two versions of their introduction. This means the safest, most up-to-date compiler version is a few versions behind the latest unless the code is affected by a bug that was recently fixed. For example, Solidity version 0.8.13 was found to suffer from a bug where under certain conditions some assembly instructions are ignored by the compiler. While the codebase in its current state does not seem to be affected by this specific bug, pinning the version reduces the odds of such vulnerabilities affecting it in the future.

Moreover, the `RollupVerifier` library makes use of a different pragma (`>=0.4.16 <0.9.0`), which is inconsistent with the one used in the rest of the codebase.

To ensure the released version matches the tested version, consider pining the Solidity compiler of the entire codebase to a specific version, preferably slightly behind the most up-to-date version (currently 0.8.20).

Update: *Resolved in pull request #636 at commit 6d88f92. The Scroll team stated:*

> *The version of all deployable contracts is pinned with `=0.8.16`. For interfaces, libraries, and abstract contracts, `^0.8.16` is used.*

Notes & Additional Information

N-01 Constant Not Using `UPPER_CASE` Format

In `AddressAliasHelper.sol`, the `offset` constant is not declared using `UPPER_CASE` format.

According to the Solidity Style Guide, constants should be named with all capital letters with underscores separating words. For better readability, consider following this convention.

Update*: Acknowledged, will resolve. The Scroll team stated:*

> *This is not a priority at the time. It will be addressed later on.*

N-02 Error-Prone Call Encoding

Throughout the codebase, calls are either encoded with `abi.encodeWithSignature` or `abi.encodeWithSelector`, both of which are prone to type or typo errors. Instead, consider using the `abi.encodeCall` function that protects against both mistakes. When making this change, use Solidity version 0.8.13 or higher, due to a bug encoding literals.

Update*: Acknowledged, will resolve. The Scroll team stated:*

> *This is not a priority at the time. It will be addressed later on.*

N-03 Events Should Emit Old and New Value

There are events in the codebase that would benefit from emitting old and new values for the sake of traceability:

- The events in the `IL1GatewayRouter` and `IL2GatewayRouter` interfaces.
- The `UpdateTokenMapping` events in the L1 and L2 gateway contracts for the address of the counterpart token.
- The `L2BaseFeeUpdated` event of the `L2GasPriceOracle` contract for the `l2BaseFee` value.

Consider emitting the respective old values to enable traceability for off-chain applications and facilitate monitoring rules for suspicious activity.

***Update**: Acknowledged, will resolve. The Scroll team stated:*

> *This is not a priority at the time. It will be addressed later on.*

N-04 Events Split Between Contracts and Interfaces

Throughout the codebase, events are placed in both contracts and interfaces. With the current pattern, events on authorized actions are placed in the contracts while user-relevant events are placed in the interfaces, negatively impacting the codebase's readability. Consider moving the events from the contracts to their respective interfaces. Furthermore, to facilitate monitoring capabilities (which rely on checking events) it is easier to compile the interfaces and obtain their ABI when they are all located in one place.

***Update**: Acknowledged, will resolve. The Scroll team stated:*

> *This is not a priority at the time. It will be addressed later on.*

N-05 Gas Optimizations

There are a few instances in the codebase where gas consumption can be reduced. For instance:

- In `ScrollChain.sol`, the `lastFinalizedBatchIndex` check can be moved up to fail early.

- Consider using the `delete` keyword instead of overwriting variables to their default.
  - In `ScrollChain.sol` `committedBatches` is overwritten.
  - In `L1MessageQueue.sol` `messageQueue` is overwritten.
- Consider using `++i` instead of `i++` for `for` loop increments.
- During the initialization of the `L1ScrollMessenger` contract, the `xDomainMessageSender` variable is set a second time after the `_initialize` function of the base contract
- In the `L1ScrollMessenger` the `relayMessageWithProof` function keeps track of all relay message calls by bundling the information into an id and setting it in a mapping. This seems to be obsolete code and an unnecessary storage write.
- Consider bumping the Solidity version to at least 0.8.1 where `address.code.length` is used, to not copy the code into memory. For more information see the release announcement.
- The double hashing of the `_l1Token` address in the `_getSalt` and `getL2ERC20Address` function is unnecessary since `_gateway` / `counterpart` and `_l1Token` are both fixed-size values that cannot result in a hash collision for different values.
- In `EnforcedTxGateway`, the `messageQueue` storage variable is read onto the stack to save gas, but then again read from storage a second time.

Consider making the above changes to reduce gas consumption.

***Update**: Acknowledged, will resolve. The Scroll team stated:*

> *This is not a priority at the time. It will be addressed later on.*

N-06 Inconsistent Integer Base in Inline Assembly

The codebase makes use of inline assembly for multiple features. When performing these calculations, a decimal and hexadecimal integer base is used interchangeably. Mostly as `32` and `0x20` to calculate word offsets in memory. Consider sticking to one integer base to ease readability and prevent calculation errors.

***Update**: Acknowledged, will resolve. The Scroll team stated:*

> *This is not a priority at the time. It will be addressed later on.*

N-07 Lack of Indexed Event Parameters

Throughout the codebase, several events do not have their parameters indexed. For instance:

- The `UpdateFeeVault` event
- The `UpdateTokenMapping` events [1] [2] [3]
- The events in `L1MessageQueue`
- The events in `L2GasPriceOracle`
- The `UpdateVerifier` event
- The `Withdrawal` event
- The `UpdateFeeVault` event

Consider indexing event parameters to improve the ability of off-chain services to search and filter for specific events.

***Update**: Acknowledged, will resolve. The Scroll team stated:*

> *This is not a priority at the time. It will be addressed later on.*

N-08 Multiple Event Emissions Can Confuse Off-Chain Clients

Throughout the codebase, there are multiple `onlyOwner`-protected functions that set sensitive addresses or values. When passing the same address or value as the one the variable currently has, the triggered event will suggest that the variable has changed its value, creating confusion for off-chain clients potentially reacting to it.

Consider validating the current setting in storage before setting the variable with the passed value and emitting the event.

***Update**: Acknowledged, will resolve. The Scroll team stated:*

> *This is not a priority at the time. It will be addressed later on.*

N-09 No Function to Remove a Custom Setting

In the `SimpleGasOracle` contract, message fees are estimated based on a fee configuration. This configuration can be set by the owner for default values but also per sender for a custom configuration. However, there is no functionality to return from a custom configuration to the default one. While the custom configuration could be changed to match

the default, it would not change along with it. Consider being more flexible and future-proof by adding a `removeCustomFeeConfig` function instead of having to implement this through a more gas-consuming contract upgrade.

Further, in the `L1{CustomERC20|ERC721|ERC1155}Gateway`, the owner can update the token mapping of L1 to L2 contracts. However, this mapping is not resettable back to address zero to prevent depositing or withdrawing such tokens in cases of deprecation or scams. As such, consider allowing the mapping to be set to zero.

***Update**: Acknowledged, will resolve. The Scroll team stated:*

> *This is not a priority at the time. It will be addressed later on.*

N-10 `SimpleGasOracle` Is Not Used

The `SimpleGasOracle` contract does not find any utility in the rest of the codebase and appears to be obsolete. Consider removing it.

***Update**: Acknowledged, will resolve. The Scroll team stated:*

> *This is not a priority at the time. It will be deleted later on.*

N-11 Token Counterpart Address in WETH Gateway Can Be Misleading

To align with the inherited L1 and L2 ERC-20 gateway interfaces, the WETH gateways ([1], [2]) implement functions to fetch the L1 and L2 token counterparts as `getL1ERC20Address` and `getL2ERC20Address`. But, since the WETH gateway only interacts with one token on each layer, these functions just return the hardcoded address regardless of what token is queried through the function parameter.

This could cause confusion since only one L1 WETH token maps the L2 token and vice versa. However, the function could suggest that other tokens also fulfill this mapping.

Even though the protocol expects users to interact with the gateways by calling the `L1GatewayRouter` contract, it is possible for users to bypass this contract and reduce the gas cost. For that reason, consider returning the zero address if the queried token does not match the WETH token of the respective layer.

***Update**: Acknowledged, will resolve. The Scroll team stated:*

> *This is not a priority at the time. It will be addressed later on.*

N-12 Typographical Errors

Throughout the codebase there are multiple instances of typographical errors:

- "called by a list of validator" should be "called by a list of validators".
- "choosen" [1] [2] should be "chosen".
- "return true is" should be "return true if"
- "The the" [1] [2] should be "The".
- "Return the message of in `queueIndex`" should be "Return the message at `queueIndex`.
- "All deposited Ether (including `WETH` deposited throng `L1WETHGateway`) will locked in this contract." should be "through" and "will be locked".
- "transfered" should be "transferred" (throughout the whole codebase).
- "recieve" should be "receive".
- "addess" should be "address".
- "malicous" should be "malicious".
- "continous" should be "continuous".

Consider fixing these and any other typographical errors to improve the readability of the codebase.

***Update**: Acknowledged, will resolve. The Scroll team stated:*

> *This is not a priority at the time. It will be addressed later on.*

N-13 Unintuitive Bitmap Ordering and Type for Skipped Messages

In the `ScrollChain` contract, as a new batch is committed, L1 messages can be skipped from being included in the batch. The indication of whether a message should be skipped or not is realized through a bitmap of type `bytes`. However, the dynamic bytes type is expected to have a length of `k` words. Thus, a `bytes32[]` array would be a more suitable type for this parameter.

Further, while the `bytes` value is processed from left to right as words, the bits are processed from right to left, giving an unintuitive ordering of skipped messages.

Consider sticking to a dynamic bytes value of arbitrary length that is processed from left to right, or changing the type to a `bytes32` array.

***Update**: Acknowledged, will resolve. The Scroll team stated:*

> *This is not a priority at the time. It will be addressed later on.*

N-14 WETH Is Passed as a Parameter

The WETH address is being passed as a parameter in the `constructor` of the `L1WETHGateway` contract. However, as the address could be considered a constant in the ecosystem, consider hard-coding the address with a constant variable in the implementation to decrease the likelihood of errors when deploying the contract.

***Update**: Acknowledged, will resolve. The Scroll team stated:*

> *This is not a priority at the time. It will be addressed later on.*

N-15 Unused Imports

Throughout the codebase, there are imports that are unused and could be removed. For instance:

- Import `ProxyAdmin` of `External.sol`
- Import `TransparentUpgradeableProxy` of `External.sol`
- Import `AddressAliasHelper` of `L1ScrollMessenger.sol`
- Import `IL2GatewayRouter` of `L1GatewayRouter.sol`
- Import `IScrollGateway` of `L1GatewayRouter.sol`
- Import `IL1ScrollMessenger` of `L1GatewayRouter.sol`

Consider removing unused imports to improve the overall clarity and readability of the codebase.

***Update**: Acknowledged, will resolve. The Scroll team stated:*

> *This is not a priority at the time. It will be addressed later on.*

N-16 Unused Named Return Variable

Named return variables are a way to declare variables that are meant to be used within a function's body for the purpose of being returned as the function's output. They are an alternative to explicit in-line `return` statements.

In `ScrollStandardERC20.sol`, there are two instances of unused named return variables:

- The `success` return variable in the `transferAndCall` function
- The `hasCode` return variable in the `isContract` function

Consider using or removing any unused named return variables. Moreover, consider keeping a consistent style throughout the codebase regarding the usage of named return variables.

***Update**: Acknowledged, will resolve. The Scroll team stated:*

> *This is not a priority at the time. It will be addressed later on.*

N-17 Use Custom Errors

Throughout the codebase, the code makes use of `require` statements with error strings to describe the reasons for reverting.

However, since Solidity version 0.8.4, custom errors provide a cleaner and more cost-efficient way to explain to users why an operation failed versus using `require` and `revert` statements with custom error strings.

For better conciseness, consistency, and gas savings, consider replacing hard-coded `require` and `revert` messages with custom errors.

***Update**: Acknowledged, will resolve. The Scroll team stated:*

> *This is not a priority at the time. It will be addressed later on.*

N-18 Variable Name Inconsistency

Throughout the codebase, some variables referring to the same matter have different names. For instance, the `ScrollChain` contract address in the `L1ScrollMessenger` contract is named `rollup`, while in the `L1MessageQueue` contract, it is `scrollChain`.

For better readability, consider giving variables the same name if they refer to the same contract.

***Update**: Acknowledged, will resolve. The Scroll team stated:*

> *This is not a priority at the time. It will be addressed later on.*

Client Reported

CR-01 Missing Chain ID Allows Reuse of Proofs

In the `ScrollChain` contract, during the finalization of a batch, the proof is checked against a hash over:

- The previous state root of the L2 chain
- The post-state root of the L2 chain
- The withdraw root for L2 to L1 messaging
- The data hash of the batch summarizing the batch's chunks of blocks and transactions

Similar conditions could be met in the case of the Scroll rollup being cloned or forked, although the proof was only meant for one particular chain. Hence, that proof can potentially be reused.

***Update**: Resolved. The Scroll team fixed this bug at commit 55f5857 of PR#517 by adding the L2 chain ID to the hashed parameters that the proof is checking against.*

Recommendations

ERC-20 Factory Design

Tokens can be bridged in a custom and standard way. For the latter, the `ScrollStandardERC20` is the default implementation that will represent the L1 token on L2. This is realized with the `Clones` library and the EIP-1167 standard. It works by deploying a minimal proxy that delegates its calls into the token implementation and that is initialized as the token instance.

These standard tokens are not upgradeable, which comes with a trade-off. On the one hand, it is more secure since the logic can not be changed. On the other hand, it is less future-proof meaning that standards like ERC-677 - which is not a finalized EIP - might at some point be overruled by a new standard that finds mass adoption.

An alternative future-proof factory design would be the Beacon proxy pattern. In a similar approach, the `BeaconProxy` will be the token instance, but then fetches the implementation contract to delegate into from a single `UpgradeableBeacon` contract. This enables upgrading all tokens in one transaction.

Regarding the security implications of upgradeable contracts, it is crucial to have the `UpgradeableBeacon` secured through a timelock, multisig, and cold wallets.

Update: *Acknowledged. The Scroll team stated:*

> *For safety concerns, we prefer the contract to be non-upgradeable.*

ERC-165 Support

While most of the codebase includes custom contracts which do not implement a specific standard, the `ScrollStandardERC20` contract is implementing the ERC-20 and ERC-677 standards. As such, it makes sense to also add ERC-165 support to it to enable other parties to identify its interface and the standard it implements.

Update: *Acknowledged. The Scroll team stated:*

> *Makes sense, we will support it if we have time.*

Testing Coverage

Due to the complex nature of the system, we believe this audit would have benefitted from more complete testing coverage.

While insufficient testing is not necessarily a vulnerability, it implies a high probability of additional hidden vulnerabilities and bugs. Given the complexity of this codebase and the numerous interrelated risk factors, this probability is further increased. Testing provides a full implicit specification along with the expected behaviors of the codebase, which is especially important when adding novel functionalities. A lack thereof increases the chances that correctness issues will be missed. It also results in more effort to establish basic correctness and increases the effort spent exploring edge cases, thereby increasing the chances of missing complex issues.

Moreover, the lack of repeated automated testing of the full specification increases the chances of introducing breaking changes and new vulnerabilities. This applies to both previously audited code and future changes to current code. This is particularly true in this project due to the pace, extent, and complexity of ongoing and planned changes across all parts of the stack (L1, L2, relayer, and zkEVM). Under-specified interfaces and assumptions increase the risk of subtle integration issues, which testing could reduce by enforcing an exhaustive specification.

We recommend implementing a comprehensive multi-level test suite consisting of contract-level tests with more than 90% coverage, per-layer deployment and integration tests that test the deployment scripts as well as the system as a whole, per-layer fork tests for planned upgrades and cross-chain full integration tests of the entire system. Crucially, the test suite should be documented in a way so that a reviewer can set up and run all these test layers independently from the development team. Some existing examples of such setups can be suggested for use as reference in a follow-up conversation. Implementing such a test suite should be a very high priority to ensure the system's robustness and reduce the risk of vulnerabilities and bugs.

Update: *Acknowledged. The Scroll team stated:*

> *More tests will be added later.*

Monitoring Recommendations

While audits help in identifying code-level issues in the current implementation and potentially the code deployed in production, the Scroll team is encouraged to consider incorporating

monitoring activities in the production environment. Ongoing monitoring of deployed contracts helps identify potential threats and issues affecting production environments. With the goal of providing a complete security assessment, the monitoring recommendations section raises several actions addressing trust assumptions and out-of-scope components that can benefit from on-chain monitoring.

Governance

Critical: There are several important contracts that use the Proxy Pattern and can be arbitrarily upgraded by the proxy owner. Consider monitoring for upgrade events on at least the following contracts:

- `ScrollChain`
- `L1ScrollMessenger`
- Gateway contracts

Access Control

Critical: `Ownable` allows implementing access control to prevent unauthorized parties making unintended changes, but it is important to monitor for events where the owner changes. Consider monitoring for the `OwnershipTransferred` event on all ownable contracts such as `ScrollChain` and `L1MessageQueue`.

Technical

High: The rollup contract contains sensitive functions that should be called only by the owner. Consider monitoring if any of the following events are emitted.

- `UpdateSequencer`
- `UpdateProver`
- `UpdateVerifier`

Medium: The `L1ScrollMessenger` contract includes a mechanism for pausing in case of an incident. Consider monitoring for the `Paused` since an unexpected pause may cause a disruption in the system.

Financial

Medium: Consider monitoring the size, cadence and token type of bridge transfers during normal operations to establish a baseline of healthy properties. Any large deviation, or unexpectedly large withdrawals may indicate unusual behavior of the contracts or an ongoing attack.

Update: *Acknowledged. The Scroll team stated:*

> *It is on our roadmap. We will have one before mainnet launch.*

Conclusion

This five-and-a-half-week audit had a somewhat challenging start due to scope changes, but it ultimately progressed smoothly and benefitted greatly from valuable insights provided by the Scroll team. The code is overall well documented which makes it easy to reason about. The architecture of the contracts is sound and inspired by other protocols.

In this report, we uncovered a few issues that indicate that the correctness of the protocol needs to be tested more thoroughly. This, together with the lack of refund mechanisms, suggests that the protocol is not yet ready for production. We recommend a more extensive test suite and another audit after the refund features have been implemented. Having said that, we see the great efforts by the Scroll team to realize this and launch the system responsibly.