OpenZeppelin | security

Scroll GasSwap, Multiple Verifier, Wrapped Ether and Diff Audit

August 31, 2023

This security assessment was prepared by OpenZeppelin.

Table of Contents

Summary

Type	zkEVM-based zkRollup, Bridge & Rollup
Timeline	From 2023-07-24 To 2023-08-11
Languages	Solidity

Total Issues	32 (13 resolved, 4 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	2 (0 resolved)
Medium Severity Issues	5 (0 resolved, 2 partially resolved)
Low Severity Issues	10 (8 resolved)
Notes & Additional Information	15 (5 resolved, 2 partially resolved)
Client Reported Issues	0 (0 resolved)

Scope

We audited the `scroll-tech/scroll` repository at the 2eb458c commit.

```
contracts
└── src
    ├── L1
    │   ├── gateways
    │   │   ├── usdc
    │   │   │   └── L1USDCGateway.sol
    │   ├── rollup
    │   │   └── MultipleVersionRollupVerifier.sol
    ├── L2
    │   ├── gateways
    │   │   ├── usdc
    │   │   │   └── L2USDCGateway.sol
    │   ├── predeploys
    │   │   └── WrappedEther.sol
    ├── gas-swap
    │   └── GasSwap.sol
    ├── interfaces
    │   └── IFiatToken.sol
    ├── libraries
    │   ├── callbacks
    │   │   └── IMessageDropCallback.sol
    │   ├── token
    │   │   ├── IScrollERC1155Extension.sol
    │   │   ├── IScrollERC20Extension.sol
    │   │   └── IScrollERC721Extension.sol
    │   ├── verifier
    │   │   └── IZkEvmVerifier.sol
    │   └── ScrollMessengerBase.sol
    ├── misc
    │   └── Fallback.sol
    └── External.sol
```

We also performed a diff audit of the `scroll-tech/scroll` repository at the 2eb458c commit against 3bc8a3f commit.

```
contracts
└── src
    ├── L1
    │   ├── gateways
    │   │   ├── EnforcedTxGateway.sol
    │   │   ├── IL1ERC1155Gateway.sol
    │   │   ├── IL1ERC20Gateway.sol
    │   │   ├── IL1ERC721Gateway.sol
    │   │   ├── IL1ETHGateway.sol
```

```
│   │   ├── IL1GatewayRouter.sol
│   │   ├── L1CustomERC20Gateway.sol
│   │   ├── L1ERC1155Gateway.sol
│   │   ├── L1ERC20Gateway.sol
│   │   ├── L1ERC721Gateway.sol
│   │   ├── L1ETHGateway.sol
│   │   ├── L1GatewayRouter.sol
│   │   ├── L1StandardERC20Gateway.sol
│   │   └── L1WETHGateway.sol
│   ├── rollup
│   │   ├── IL1MessageQueue.sol
│   │   ├── L1MessageQueue.sol
│   │   └── ScrollChain.sol
│   ├── IL1ScrollMessenger.sol
│   └── L1ScrollMessenger.sol
├── L2
│   ├── gateways
│   │   ├── L2ERC1155Gateway.sol
│   │   ├── L2ERC721Gateway.sol
│   │   └── L2ETHGateway.sol
│   ├── predeploys
│   │   └── L1GasPriceOracle.sol
│   ├── IL2ScrollMessenger.sol
│   └── L2ScrollMessenger.sol
├── gas-swap
│   └── GasSwap.sol
└── libraries
    ├── constants
    │   └── ScrollConstants.sol
    ├── gateway
    │   └── ScrollGatewayBase.sol
    ├── token
    │   ├── IScrollERC1155.sol
    │   ├── IScrollERC20.sol
    │   └── IScrollERC721.sol
    ├── verifier
    │   └── IRollupVerifier.sol
    └── ScrollMessengerBase.sol
```

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.

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 structures from Arbitrum's gateways and the `AddressAliasHelper.sol` contract are reused with minor modifications.

There were several major changes since the last audit. The first major change was to add a refund for skipped messages only and to remove the message retry logic from the L2 side. There was also an addition of the `MultipleVersionRollupVerifier` contract which allows for different verifiers to be used for different batches. A `GasSwap` contract was also added to help users swap tokens for L2 ETH. Aside from these major changes, this audit focused on diff audits across almost all contracts across this protocol. In this audit, we aimed to verify the correctness and security of the contracts, focusing on block finalization, message passing, and the process of depositing and withdrawing into/from the rollup.

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. These custom gateways are also required for ERC-721 and ERC-1155 tokens, which currently do not have a standard gateway provided. However, the GatewayRouter does not currently support ERC-721 or ERC-1155 custom gateways.

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 a refund mechanism for (1) only if the L1 initialized message is not provable and hence not executed and skipped from the L1 message queue. If an L1 message is not executed, reverted, or otherwise in a situation where it has succeeded on L1 but failed on L2 and yet is proven back onto L1, there are no refunds available. 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 and prover have 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.
- **Whitelist ownership**: The whitelist contract has an owner who can update the whitelist status of different addresses. This implies trust in the owner of the whitelist to manage this list correctly and in the best interest of the system and its users.

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`
  - `L2ScrollMessenger`
- **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.

    - `ScrollStandardERC20Factory`: Use the factory to deploy another instance of a standard ERC-20 token on L2.
    - `L2ScrollMessenger`: Pause relaying of L1 to L2 messages and L2 to L1 message requests.
    - `L2{CustomERC20|ERC721|ERC1155}Gateway`: Change the token mapping of which L2 token is bridged to which L1 token.
    - `L2GatewayRouter`: Set the respective gateway for ETH, custom ERC-20s and default ERC-20s.
    - `L2MessageQueue`: Update the address of the messenger.
    - `L2TxFeeVault`: Change the messenger address that is used to withdraw the funds from L1 to L2, the recipient address of the collected fees, and update the minimum amount of funds to withdraw.
    - `L1BlockContainer`: Initialize the starting block hash, block height, block timestamp, block base fee, and state root.
    - `L1GasPriceOracle`: Update the gas price and whitelist.
    - `Fallback`: Withdraw ERC20 tokens and ETH as well as execute arbitrary messages.
    - `Whitelist`: Accounts can be whitelisted to change the L2 base fee on L1 as well as the intrinsic gas parameters.
    - `GasSwap`: Withdraw stuck ERC20 tokens and ETH, update the fees, and set the approved targets to call
    - `MultipleVersionRollupVerifier`: Set the new verifier and their starting batch index
- **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.
- **Prover**: The prover role can interact with the `ScrollChain` contract to prove batches that have already been committed by the sequencer, thus finalizing them.

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 Lack of Refunds

The protocol allows users to bridge their ERC-20, ERC-721, ERC-1155, USDC, 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, USDC, 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:*

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

H-02 Potentially Stuck USDC From Pausing

Currently the `L2USDCGateway` has pausing functionality that its L1 counterpart does not have. This is a feature that will be used by the protocol to do supply locking in the future.

An issue arises when the `L2USDCGateway` is paused while the `L1USDCGateway` remains unpaused and usable. Users may still use the gateway to submit messages from L1 to L2, and while the transaction could succeed on L1, it will fail during finalization on L2 via the paused deposit check in the `finalizeDepositERC20` function. Therefore, the users who deposited L1 USDC will have their tokens locked in the gateway and unable to replay their transaction. Assuming that the transaction was not skipped, dropMessage cannot be called to obtain a refund.

As USDC is one of the most widely used tokens, this issue could cause a high volume of users to have locked funds. Consider adding the same pausing functionality to the `L1USDCGateway` and updating the pausing state on both sides at the same time. Furthermore, consider implementing a refund mechanism to unlock user funds when their message from L1 to L2 fails.

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

> *We will ensure that in L1 `pauseDeposit` will be called first and in the meantime, in L2, `pauseWithdraw` will be called. After this, the pending L1=>L2 or L2=>L1 messages are relayed. The pausing withdraw in L1 and pausing deposit in L2 will enable (actually not needed but just in case). We will help relay pending L2=>L1 message if users forget to withdraw the USDC. We will also help replay L1=>L2 messages if they fail in L2 due to running out of gas. Skipped messages are not possible in the USDC gateway, since we disabled deposit/withdraw with data.*

Medium Severity

M-01 Lack of Expiration for Retrying Transactions

The `L1ScrollMessenger` contract provides a mechanism to retry failed transactions from L1 to L2 at any time. Therefore, if a user creates an L1 to L2 transaction and it fails, it can be retried indefinitely at a later time. A scenario could occur where a user erroneously sends an L1 to L2 transaction, which fails at the L2 level. The user may be inclined to send another transaction, which would succeed at the L2 level. However, a malicious recipient could later retry the user's first transaction against their will, which could succeed.

More generally, if a transaction is sent from L1 to L2 and fails, it can be retried indefinitely at a later time. Consider adding an expiration time to replay failed transactions, limiting the timeframe during which a failed transaction can be retried.

Update: *Partially resolved in pull request #840 at commit 0d7d73f. The `L1ScrollMessenger` and `L2ScrollMessenger` contracts were updated to store the timestamp of when the message was sent. However, checking for expiration was not implemented. The Scroll team stated:*

> *Only the timestamp is added for each message, the expiration check will be added together with the refund feature.*

M-02 Potentially Stuck ETH in `L1ScrollMessenger`

The function `dropMessage` in `L1ScrollMessenger` allows the dropping of a message sent from L1 to L2 that has been skipped in the proof. In order for a sender to obtain their refund when their message is skipped, the `dropMessage` function is called and the `_value` of the message is returned to the sender using the `onDropMessage` function call. However, if the sender's address does not include the `onDropMessage` or a `payable fallback`, this callback function will fail. While the gateways provided by the protocol do have the `onDropMessage` implemented, it is possible that the user had sent a message by calling `L1ScrollMessenger` directly using an EOA or a smart wallet. In the future, it is also possible that a sender would use their own gateway that does not implement the `onDropMessage` function.

Consider allowing the `_value` to be returned to the sender even if the sender does not implement the `onDropMessage` function.

Update: *Acknowledged, will resolve. The Scroll team stated that they will resolve the issue:*

> *This will be resolved if we implement the refund feature.*

M-03 Potentially Stuck ETH from Incorrect Data Parameter

The protocol allows users to bridge their assets to the L2 rollup and back through the `L1ScrollMessenger` and the `L2ScrollMessenger`. When bridging from L1 to L2, the `_executeMessage` will be called. On line 198 of this function, the `_to` address is called and the `_message` is passed to this function. If a user accidentally sets a value in the `_message` field, but either the `_to` address is an EOA, the message is in an incorrect format, or the address is a contract that does not support the data in this field, the user's assets will be stuck on L1, as the L1 transaction has succeeded but the L2 transaction will fail.

The replaying of this message will not help, as the `_message` field cannot be changed for a replay, and assuming that the transaction was not skipped, `dropMessage` cannot be called to get a refund.

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, will resolve. The Scroll team stated that they will resolve the issue:*

> *This will be resolved if we implement the refund feature.*

M-04 Use of Non-Production-Ready Trusted Forwarder

The `GasSwap` contract inherits from `ERC2771Context` thereby allowing meta-transactions to work with its functions. It relies on a trusted forwarder that is set in the constructor. The trusted forwarder that it depends on is the `MinimalForwarder`, which is located in the `External` contract. However, the `MinimalForwarder` is not ready for production use and is mainly meant for testing.

By using the `MinimalForwarder`, ETH could potentially be lost. In addition, the `MinimalForwader`'s signed requests do not expire and lack batching, which is useful when dealing with a large volume of requests to be forwarded.

Consider using OpenZeppelin's `ERC2771Forwarder` instead. While this contract is not available until v5.0 is released, the source code can be obtained from the `master` branch and inserted into the codebase.

Update: *Partially resolved in pull request #843 at commit 709101a. The Scroll team replaced the `MinimalForwarder` with the `ERC2771Forwarder` contract but did not remove the `MinimalForwarder` import from the `External.sol` file*

M-05 `replayMessage` and `dropMessage` Can Be Front-Run

The `L1ScrollMessenger` contract is used to send messages from L1 to L2. Failed messages can be replayed using the `replayMessage` function. In addition, skipped messages can be dropped using the `dropMessage` function. A scenario could occur where a user sends an L1-to-L2 transaction that gets skipped. The user can now choose to either replay or drop the message.

However, an adversary could void the user's intention by front-running the user's intended action if they have an incentive to do so. If a user wants to replay their message, the adversary could front-run the user and call `dropMessage`. Similarly, if a user wants to drop their message, the adversary could front-run the user and call `replayMessage`.

Consider only allowing the sender of the original message to replay or drop their message to prevent front-running attacks.

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

> *Firstly, we do not think the fact that the transaction can be front-run is a big problem. Secondly, it is not easy to figure out which user is the original sender of the message. Thirdly, in some situations, we want to replay or drop message for some users, so we do not plan on fixing the issue.*

Low Severity

L-01 Error-Prone Call Encoding

On line 126 of `ScrollMessengerBase.sol` a call is encoded with `abi.encodeWithSignature`, which is prone to typographical errors. Instead, consider using the `abi.encodeCall` function that protects against mistakes. When making this change, ensure that at least Solidity version 0.8.13 or above is used, due to a bug encoding literals.

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

> *This is the only `encodeWithSignature` left in the contracts. The reason why we do not fix it is because the `relayMessage` is only used in `L2ScrollMessenger`. If we use `abi.encodeCall`, some L2 only interface will be introduced to the base contract, which is not good in our opinion.*

L-02 Anyone Can Steal ERC-20 Tokens From `GasSwap`

The `GasSwap` contract implements the `withdraw` function to allow the owner to `withdraw` ETH or ERC-20 tokens that are stuck within the contract. This contract also contains a `swap` function, which allows users to swap any ERC-20 token for a specified amount of ETH. This function performs the swap followed by a refund of the unswapped tokens to the user.

However, the amount refunded to the user is the total balance of the `GasSwap` contract of that ERC-20 token, which includes the amount of that ERC-20 token that has been stuck in the contract. Therefore, if there is any ERC-20 token that is stuck in `GasSwap`, any user can execute a swap of 0 tokens of that particular ERC-20 token and obtain these stuck tokens for themselves as the refund.

When swapping, consider taking into account the user's balance of the swapped token before and after the swap, before refunding the user.

Update: *Resolved in pull request #844 at commit 7a26dbc.*

L-03 Inconsistency of Allowing a Trusted Forwarder

In `GasSwap`, some parts of the code rely on `_msgSender` and others on `msg.sender`. This inconsistency can lead to confusion as to when meta-transactions are allowed. For example, `GasSwap` inherits from `OwnableBase` which has an `onlyOwner` check that uses `msg.sender`. Therefore, all functions with `onlyOwner` modifier will fail when submitted through a trusted forwarder.

This inconsistency is confusing and error-prone. Consider using `_msgSender()` everywhere in `GasSwap` and `OwnableBase`, which will automatically default to `msg.sender` if it is not sent from the `trustedForwarder` address.

Update: *Resolved in pull request #846 at commit 60de22b.*

L-04 Inconsistency of Reentrancy Guard

The `ScrollMessengerBase` and `ScrollGatewayBase` contracts implement a `nonReentrant` modifier. Instead, consider utilizing the ReentrancyGuardUpgradeable contract of the OpenZeppelin library which is a dependency in use. The reimplementation of such safety mechanisms is generally discouraged. In addition, this is inconsistent with other contracts such as `EnforcedTxGateway`, which does, in fact, use OpenZeppelin's reentrancy guard. In both cases, make sure to initialize the contracts properly if they are used for upgradeable contracts.

Update: *Resolved in pull request #698 at commit a798e4d.*

L-05 Potentially Stuck ETH in `GasSwap`

In the `GasSwap` contract, the `swap` function allows a user to swap ERC-20 tokens for ETH. The swap is executed by calling an approved target, which in return transfers ETH to the `GasSwap` contract through the `receive` function. The user is then reimbursed the output amount of ETH after deducting the fee.

However, any user can accidentally transfer ETH to the `GasSwap` contract without calling the `swap` function. Any ETH transferred outside of a swap will be locked in the contract and is only redeemable through the owner.

Consider only allowing ETH transfers to the `GasSwap` contract from approved targets to minimize the probability of having ETH accidentally locked in this contract. Alternatively, consider only allowing ETH transfers to the contract during a swap.

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

> *Our owner can rescue the stuck ETH as long as the user provides the transfer transaction.*

L-06 Possible Misleading `revert` Message When Swapping Non-`ERC20Permit` Tokens

The `GasSwap` contract allows users to sign an off-chain transaction and pass it to a forwarder to swap any `ERC20Permit` token on their behalf. However, since the `_permit.token` can be arbitrarily chosen by the user, there is no guarantee that the address of the token is an `ERC20Permit` token. If the token has a `fallback` function that fails silently, the call to the `permit` function will not revert. In this case, the transaction will fail when trying to call `safeTransferFrom`, emitting an insufficient allowance message.

Consider using the `safePermit` function from the imported `SafeERC20` library to fail with a more reasonable message.

Update: *Resolved in pull request #847 at commit 8daae8f.*

L-07 Potentially Misleading Verifier Event

The `MultipleVersionRollupVerifier` contract maps the batch index to the address of the verifier that was used. The owner of this contract can call the `updateVerifier` function in order to update the verifier. For this update to succeed, it requires that the provided `startBatchIndex` is greater than or equal to the previous verifier's `startBatchIndex`.

However, it does not check if that batch number has already been verified by the current verifier, which could be confusing. For example, if the current verifier's start index is 100, and it has verified batches up to batch 105, the new verifier could be set with a start index of 102. The event emitted would contain the index of 102, which would be confusing because it would appear to a user monitoring the events as though the new verifier was used to verify the batches 102-105 when in reality it was not. This could even be used maliciously by the owner to hide information relating to a faulty verifier.

Consider enforcing that the start index must be greater than the `lastFinalizedBatchIndex`.

Update: *Resolved in pull request #849 at commit 6527331.*

L-08 Redundancy of Replaying Messages in `L2ScrollMessenger`

The logic to retry messages sent from L1 to L2 lives solely in the `replayMessage` function in `L1ScrollMessenger`. In addition, the maximum number of times that a message can be replayed as well as the `replayStates` are also located in `L1ScrollMessenger`.

However, the `L2ScrollMessenger` also contains a `maxFailedExecutionTimes` state variable and a `l1MessageFailedTimes` mapping, which are tracking the same values as on the L1 side. When executing a message in `L2ScrollMessenger`, it uses these variables to determine whether or not a transaction will succeed. However, given that the maximum number of failed execution times can be updated to be a different value than that on the L1 side, this could cause confusion for a user who replays a transaction from L1. This duplication of function can also result in future issues when refactoring code, which could lead to future vulnerabilities.

Consider removing all code related to replaying messages in `L2ScrollMessenger` to reduce code duplication as well as the potential surface of future errors.

Update: *Resolved in pull request #850 at commit 51a74dd.*

L-09 Misleading and Incorrect Comments

Throughout the code, we found several comments that were either misleading or incorrect. Some examples are:

- This comment in `ScrollMessengerBase` suggests moving the declaration of `_lock_status` to `ScrollMessengerBase` in the next big refactor, which has already been done.
- The comments on lines 36 and 172 in `L1ERC721Gateway` should say `_l2Token` instead of `_l1Token`.
- The comment on line 134 in `L2USDCGateway.sol` should say `L2GatewayRouter` instead of `L1GatewayRouter`.

Consider resolving these instances of incorrect documentation to improve the clarity and readability of the codebase.

Update: *Resolved in pull request #851 at commit b933adb.*

L-10 `maxReplayTimes` is Not Initialized in `L1ScrollMessenger`

Currently `maxReplayTimes` is not being set in the `initialize` function of `L1ScrollMessenger`. In addition, the deployment script for `L1ScrollMessenger` does not call `updateMaxReplayTimes` to set `maxReplayTimes` to an appropriate value. An uninitialized value of `maxReplyTimes` would have the value of 0, which would prevent any user from calling `replayMessage` due to the max reply check reverting. Therefore, this will happen until `updateMaxReplayTimes` is called to set `maxReplayTimes`.

Consider setting `maxReplayTimes` to an appropriate value during the initialization of the `L1ScrollMessenger` contract.

Update: *Resolved in pull request #852 at commit 020d272.*

Notes & Additional Information

N-01 Tokens With Permit Functionality Can Be Front-Run

Tokens that are created on L2 through the standard flow of bridging from `L1StandardERC20Gateway` will be of type `ScrollStandardERC20`, which has `permit` functionality. In addition, the `WrappedEther` contract will also have `permit` functionality. If a user wants to call the `permit` function for these tokens from L1 to L2 through the `EnforcedTxGateway`, a malicious user could read it and execute the `permit` in L2 before it arrives.

The result would be that when this L1 to L2 message is executed on L2, it would fail since it would attempt to execute the same `permit` function with the same nonce. This griefing

attack could confuse the user, who may believe that the `permit` did not succeed on L2, and cause them to re-send the message, costing the user more gas.

Consider discouraging calling the `permit` function from L1 to L2 or documenting the possibility of front-running griefing attacks more thoroughly. Alternatively, consider removing the `permit` functionality from frequently used tokens in L2.

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

> *It seems not a big problem, we think it is not necessary to fix.*

N-02 Extraneous Use of `safeApprove`

In the `swap` function of `GasSwap`, the function `safeApprove` is used to approve an amount of 0 and once again to approve the desired value. While this will mitigate the attack against double increments, it is gas-inefficient and not the best practice. Furthermore, `safeApprove` will not be supported in future OpenZeppelin contracts.

Consider avoiding this anti-pattern with the use of `safeIncreaseAllowance` and `safeDecreaseAllowance` instead.

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

> *We want make sure only at most `amount` is used by the target contract, so safeApprove is our best choice.*

N-03 Variables Missing `immutable` Keyword

Throughout the codebase, there are variables that remain unchanged after initialization and can be optimized by declaring them as immutable variables and setting them in the constructor. This could increase code clarity as well as reduce gas costs. For instance:

- The variables `rollup` and `messageQueue` in `L1ScrollMessenger`
- The variable `counterpart` in `ScrollMessengerBase`
- The variables `messenger` and `scrollChain` in `L1MessageQueue`
- The variable `messageQueue` in `ScrollChain`

Consider declaring these variables as `immutable` if they are not meant to be updateable later.

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

> *Some of the addresses are not known until deploy, so current way works well for us.*

N-04 Inconsistency Between `maxReplayTimes` and `ReplayState.times`

The state variable `maxReplayTimes` is of type `uint256`, while the struct field `ReplayState.times` is of type `uint128` (for the purpose of packing the struct into one storage slot).

If `maxReplayTimes` is set to be greater than `type(uint128).max`, a user would be allowed to replay their message more than `type(uint128).max` times. Due to the use of `unchecked block`, the value of `_replayState.times` could overflow, causing unexpected behavior.

Consider changing `maxReplayTimes` from `uint256` to `uint128` to prevent this potential overflow.

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

> *The overflow is not likely to happen. If someone replays the message 1 time per second, it well need 1079028307080601418897052915499O years.*

N-05 Lack of Indexed Event Parameters

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

- The `UpdateMaxReplayTimes` event
- The `DequeueTransaction` event
- The `DropTransaction` event
- The `UpdateVerifier` event
- The `UpdateMaxNumL2TxInChunk` event
- The `UpdateMaxFailedExecutionTimes` event
- The `UpdateTokenMapping` events 1 2
- The `UpdateWhitelist` event
- The `UpdateFeeRatio` event
- The `UpdateApprovedTarget` event

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

Update: *Partially resolved in pull request #673 at commit b7a02fb. The client added in indexed parameters for a few events, but kept the following events unindexed:*

- *The `UpdateMaxReplayTimes` event*
- *The `DequeueTransaction` event*
- *The `DropTransaction` event*
- *The `UpdateVerifier` event*
- *The `UpdateMaxNumL2TxInChunk` event*
- *The `UpdateMaxFailedExecutionTimes` event*
- *The `UpdateWhitelist` event*
- *The `UpdateFeeRatio` event*
- *The `UpdateApprovedTarget` event*

N-06 Inconsistent Coding Style

Throughout the codebase, a couple of `solhint-disable no-empty-blocks` instances were found that are unnecessary. The `no-empty-blocks` rule should only be used to suppress `solhint` warnings if a code block has zero statements inside. However, multiple instances were found where the rule was implemented, but no empty blocks were found in the code. For instance: - Line 19 in `L1ERC20Gateway` - Line 7 in `L2ERC20Gateway`

Consider removing the `solhint-disable no-empty-blocks` statements listed above to improve the clarity of the codebase.

Update: *Resolved in pull request #874 at commit ba98a1a.*

N-07 Incorrect Function Visibility

The `depositETH` function is not called internally by any of the functions in the `L1ETHGateway` contract. Consider setting the visibility to `external` instead of `public`.

Update: *Resolved in pull request #875 at commit f5e5de1.*

N-08 Inconsistent Order of Event Emissions

Throughout the codebase, events are emitted after the corresponding storage has changed. However, in `revertBatch` in `ScrollChain`, the `RevertBatch` event is being emitted before resetting the batch value.

To improve the readability and consistency of the codebase, consider always following the same order of operations when emitting an event in response to a change in the state of the contract.

Update: *Resolved in pull request #876 at commit 4b8d9ce.*

N-09 Missing and Inconsistent Event Emissions

Throughout the codebase, several constructors and initializers do not emit events after initializing sensitive variables in the system, although when those variables are updated using setter functions, an event is emitted. For instance:

- In the `constructor` of `FeeVault`, both the storage variables `minWithdrawAmount` and `recipient` are changed without emitting the events `UpdateMinWithdrawAmount` and `UpdateRecipient` respectively.
- When initializing `ScrollMessengerBase`, the storage variable `feeVault` is changed without emitting the respective `UpdateFeeVault` event.
- In the `constructor` of `GasSwap`, setting the storage variable `owner` does not emit the `OwnershipTransferred` event.
- In the `constructor` of `MultipleVersionRollupVerifier`, the storage variable `latestVerifier` is changed without emitting the respective `UpdateVerifier` event.
- When initializing `EnforcedTxGateway`, the storage variable `feeVault` is changed without emitting the respective `UpdateFeeVault` event.

In addition, in `L1ScrollMessenger`, the functions `dropMessage` and `replayMessage` do not emit an event when a message is dropped or replayed. Also, the functions `pauseDeposit` and `pauseWithdraw` in `L2USDCGateway` do not emit an event when deposits or withdrawals are paused.

Consider emitting events when changing state variables. Moreover, consider replacing the manual variable declarations in constructors and initializers with the corresponding setter function to update those variables.

Update: *Acknowledged, will resolve. The Scroll team stated that they will resolve the issue:*

> *It will be fixed later on when we have more time.*

N-10 Code Duplication

There are instances of duplicated code within the codebase. Duplicating code can lead to issues later in the development lifecycle and leaves the project more prone to the introduction of errors. Such errors can inadvertently be introduced when functionality changes are not replicated across all instances of code that should be identical. For instance:

- In the `L1MessageQueue` contract the value while declaring `_queueIndex` can be replaced by calling `nextCrossDomainMessageIndex`. Consider marking `nextCrossDomainMessageIndex` as `public`.
- In the `L1CustomERC20Gateway` contract multiple instances, for example on lines 102 and 122, obtain the value of the L2 token address by accessing the mapping directly. Consider using the designated `getL2ERC20Address` function.
- In the `L1ERC20Gateway` contract, if `msg.sender != router`, the amount requested is calculated in the `else`-block. However, the calculation can be replaced by calling the designated `requestERC20` function.
- In the `L1ERC721Gateway` contract, lines 196 to 197 and 227 to 228 obtain the corresponding L2 address of the token being deposited and check if the address exists. Consider replacing these lines with a corresponding internal function.
- All instances of the `OwnableBase` contract can be replaced by OpenZeppelin's `Ownable` contract.
- During the initialization of the `L1GatewayRouter` contract, the declaration of the variables `defaultERC20Gateway` and `ethGateway` can be replaced by the `setDefaultERC20Gateway` and `setETHGateway` functions respectively.
- During the initialization of the `L2GatewayRouter` contract, the declaration of the variables `defaultERC20Gateway` and `ethGateway` can be replaced by the `setDefaultERC20Gateway` and `setETHGateway` functions respectively.

Consider removing the listed code duplications to improve the readability and consistency of the codebase.

Update: *Acknowledged, will resolve. The Scroll team stated that they will resolve the issue:*

> *It will be fixed later on when we have more time.*

N-11 Follow the Checks-Effects-Interactions Pattern

In the `relayMessageWithProof` function in `L1ScrollMessenger`, there is an external call which is made before a state update, thereby breaking the Checks-Effects-Interactions pattern. While currently there may not be a reentrancy risk, it is still considered best practice to follow this pattern to mitigate future issues that may occur when refactoring code.

Consider rewriting this function to follow the Checks-Effects-Interactions pattern.

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

> *It is hard to do so, that's why we add a reentrancy guard here.*

N-12 Unused Function Parameter

The `_receiver` parameter is never used in any implementations of the `_beforeDropMessage` function.

Consider removing the unused function parameter to avoid confusion.

Update: *Resolved. This is not an issue. The Scroll team stated:*

> *Yeah, it is not used in our gateways. But we keep it, just in case if thirdparty gateways need it.*

N-13 Unnecessary Usage of Upgradeable Interfaces

Upgradeable patterns are useful for contract upgradeability. While useful in the context of the implementation contract, it is unnecessary to use the following upgradeable interfaces and libraries to call into an external contract. Using upgradeable interfaces and libraries can introduce unnecessary complexity to your codebase and reduce code readability. For example: - `SafeERC20Upgradeable` and `IERC20Upgradeable` in `L1GatewayRouter`. - `SafeERC20Upgradeable` and `IERC20Upgradeable` in `L1CustomERC20Gateway`. - `SafeERC20Upgradeable` and `IERC20Upgradeable` in `L2USDCGateway`.

Consider using upgradeable patterns only for the code contract logic and implementing non-upgradeable interfaces and libraries for better code compatibility and consistency.

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

> *All the contracts mentioned are upgradeable, I think it is ok to use the upgradeable version of IERC20 and SafeERC20.*

N-14 Duplicate Imports

There are duplicate imports throughout the codebase. For instance:

- The duplicate import `ScrollGatewayBase` in `L1ERC20Gateway.sol`
- The duplicate import `IERC20` in `GasSwap.sol`

Consider removing duplicate imports to improve the readability of the codebase.

Update: *Resolved in pull request #877 at commit b73f7a9.*

N-15 Unused Imports

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

- Import `AddressAliasHelper` of `L1ScrollMessenger.sol`
- Import `IScrollMessenger` of `L1ERC20Gateway.sol`
- Import `ScrollConstants` of `L1ERC20Gateway.sol`
- Import `IScrollGateway` of `L1GatewayRouter.sol`
- Import `IL1ScrollMessenger` of `L1GatewayRouter.sol`
- Import `IL1BlockContainer` of `L2ScrollMessenger.sol`
- Import `IL1GasPriceOracle` of `L2ScrollMessenger.sol`
- Import `PatriciaMerkleTrieVerifier` of `L2ScrollMessenger.sol`
- Import `IERC1155Upgradeable` of `L2ERC1155Gateway.sol`
- Import `IScrollGateway` of `L2ERC1155Gateway.sol`
- Import `IERC721Upgradeable` of `L2ERC721Gateway.sol`
- Import `IScrollGateway` of `L2ERC721Gateway.sol`
- Import `IScrollGateway` of `L2USDCGateway.sol`
- Import `IL1BlockContainer` of `L1GasPriceOracle.sol`

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

Update: *Partially resolved in pull request #698 at commit 3e21edb. The client removed most unused imports, but kept the following:*

- *Import* `IScrollMessenger` *of* `L1ERC20Gateway.sol`
- *Import* `ScrollConstants` *of* `L1ERC20Gateway.sol`
- *Import* `PatriciaMerkleTrieVerifier` *of* `L2ScrollMessenger.sol`
- *Import* `IScrollGateway` *of* `L2USDCGateway.sol`

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 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 cannot 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 factory design that is future-proof 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 to from a single `UpgradeableBeacon` contract. This allows 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.

ERC-165 Support

While most of the codebase is comprised of 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 enable other parties to identify its interface and the standard it implements.

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 reduces 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). Underspecified 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 >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 of 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.

Custom Gateway Contracts

Developers who need to use custom gateway contracts should ensure that their contracts are designed to allow for the burning of tokens and the creation of the same tokens with the same ID. This is particularly relevant for non-fungible tokens (NFTs) and other unique asset types that rely on the token ID for maintaining uniqueness.

The burning of tokens on one layer (L1 or L2) and the subsequent creation of the same tokens on the other layer is a crucial feature for token bridging in layer 2 solutions like Scroll. However, most NFT contracts are designed to create new tokens with a sequential counter, which makes them incompatible with this requirement.

Ensure providing adequate documentation, and if possible code examples, so that developers in the Scroll ecosystem can properly implement these requirements.

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`
- `L2ScrollMessenger`
- Gateway contracts

Access Control

Critical: `Ownable` allows implementing access control to prevent unauthorized parties from 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`, `MultipleVersionRollupVerifier`, `GasSwap`, `L1BlockContainer`, `L1GasPriceOracle`, `L1MessageQueue`, `L2MessageQueue`, `L2TxFeeVault`, and `Whitelist`.

Technical

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

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

Medium: The `L1ScrollMessenger`, `L2ScrollMessenger`, and `L2USDCGateway` contracts include a mechanism for pausing in case of an incident. Consider monitoring for `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, such as an unexpectedly large withdrawal, may indicate unusual behavior of the contracts or an ongoing attack.

Conclusion

This three-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, who were responsive and thorough with their answers to our questions. The codebase of the Scroll protocol is well-documented and organized, making it easier to understand its functionality and potential vulnerabilities. The architecture of the contracts is sound and inspired by other protocols.

However, the fact that the protocol had not yet implemented a refund mechanism casts some doubts on the protocol's readiness for it's production use. The lack of refund mechanisms, except for skipped messages, is an important feature for safeguarding user assets when bridging them, which points towards the need for further development before the protocol can be considered ready for live deployment. In addition, we recommend a more exhaustive and rigorous test suite after this feature has been implemented. Particularly, we feel that the protocol would benefit from more extensive integration testing. Having said that, we see great efforts by the Scroll team to realize this and launch the system responsibly.