OpenZeppelin

zkSync Layer 1 Audit

OPENZEPPELIN SECURITY | NOVEMBER 23, 2022

Security Audits

This first security assessment for zkSync 2.0 was prepared by OpenZeppelin, as part of an ongoing security partnership with Matter Labs.

Table of Contents

OpenZeppelin

Type	Rollup
Timeline	From 2022-09-05 To 2022-09-30
Languages	Solidity
Total Issues	35 (25 resolved, 1 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	4 (4 resolved)
Low Severity Issues	16 (14 resolved)
Notes & Additional Information	15 (7 resolved, 1 partially resolved)

Scope

In scope were the following contracts:

```
contracts/ethereum/contracts
├── zksync
│   ├── Storage.sol
│   ├── DiamondInit.sol
│   ├── DiamondProxy.sol
│   ├── Config.sol
│   ├── facets
│   │   ├── Executor.sol
│   │   ├── Mailbox.sol
│   │   ├── DiamondCut.sol
│   │   ├── Getters.sol
│   │   ├── Governance.sol
│   │   └── Base.sol
│   ├── libraries
│   │   ├── Diamond.sol
│   │   ├── PriorityQueue.sol
│   │   └── Merkle.sol
│   └── interfaces
│       ├── Mailbox.sol
│       ├── IExecutor.sol
│       ├── IGetters.sol
│       ├── IDiamondCut.sol
│       ├── IGovernance.sol
│       └── IZkSync.sol
└── common
    ├── ReentrancyGuard.sol
    ├── Dependencies.sol
    └── libraries/UnsafeBytes.sol
```

System Overview

In zkSync users sign and send transactions to validators who process them, include them into blocks, and create a cryptographic commitment of the updated state. This commitment (root hash) is then transferred to a smart contract on layer 1 along with a cryptographic proof (SNARK) proving that this new state was correctly calculated based on applying transactions to a previous state. A compressed state update is also sent to layer 1, allowing anyone to reconstruct the state at any moment. The layer 1 contract validates both the state update and the cryptographic proof, assuring the validity of the transactions included in the block and the data availability.

zkSync protocol implements the Diamond Proxy EIP-2535 as an upgrade mechanism, thereby splitting its functionality into four different facets:

- *Executor*: Processing rollups of layer 2 blocks
- *Mailbox*: Bidirectional communication between layer 1 and layer 2
- *DiamondCut*: Contract administration
- *Governance*: Role management

Note that while the aforementioned EIP does not contain any known issues, it is not yet considered finalized.

Executor

The Executor component allows validators to commit, prove, and execute blocks. All blocks are tentative before execution and can be removed by any validator. This component is central in extending the security guarantees of layer 1 to transactions on layer 2 through rollups.

Mailbox

The Mailbox component handles bi-directional communications between layer 1 (L1) and layer 2 (L2). To request an L2 transaction from L1, the transaction data is appended to a queue and removed from it upon final inclusion of the L2-rollup into the L1 contract.

In contrast, communication from L2 to L1 is divided in two parts: Sending a transaction on L2 and reading it from L1. To send information from L2, a special opcode `sendToL1` is implemented. Using this opcode, users can send logs or messages. Logs provide a key-value tuple of 32 bytes inclusion is made available on L1 by checking against the Merkle tree root. Upon committing a block, verifications are performed to ensure data availability, enabling anyone to prove message inclusion without additional help from the operator.

DiamondCut

Note: For a comprehensive explanation of the diamond update mechanism please refer to EIP-2535.

This component manages upgrade-related operations and freezing/unfreezing of facets. The upgrade mechanism is comprised of three stages:

1. Upgrade proposal – In this stage, the governor commits both a sequence of changes (add/replace/remove) to the supported facet functions and the fixed address of an initializer contract.
2. Upgrade notice period – zkSync users are given a constant timeframe to withdraw their funds if they are against the proposed upgrade, unless the Security Council approves an immediate emergency upgrade, thereby skipping the execution delay.
3. Upgrade execution – The governor can execute the upgrade and provide additional calldata to the initializer contract.

Freezing Mechanism

Matter Labs team has implemented a freezing feature. When defining the facet through diamond cuts, each facet can be set as freezable or not. The governor can freeze the diamond as a whole which affects all freezable facets. Therefore, it is possible to designate parts that shall remain operational in an emergency situation. It is crucial for the DiamondCut facet to remain operational in order to enable the governor to unfreeze the Diamond Proxy after resolving the emergency situation.

Governance

The Governance component allows the governor to assign and remove the valdiator role from addresses. It further enables the transition of contract administration to a new governor by

Operation Modes

Regular Operation

This operation mode allows only registered validators to commit and execute blocks with rolled-up layer 2 transaction information. Thus, users are dependent on the validators to not censor their transactions. To protect against malfunction and malice a second operation mode exists.

Priority Mode

Under specific conditions the system can enter a special operation mode called Priority Mode. This mechanism is intended as an escape hatch to allow users to withdraw their funds from zkSync protocol in the case an operator becomes malicious or unavailable. **During the course of this audit this feature was out of scope due to the fact that it was not yet implemented**.

Privileged Roles and Security Assumptions

The *governor* is a single address that can perform critical administrative actions such as proposing, canceling, and executing upgrades of the Diamond Proxy, as well as freezing and unfreezing the proxy. The governor is restricted by a time-delay between proposal and execution of any upgrade. However, no other entity can veto, postpone, or restrict the governor’s actions. Further, the governor can set and unset addresses as validators. The governor is considered a trusted entity.

The *security council* is a set of addresses that can skip the time-delay within the approval process of upgrades proposed by the governor to allow immediate incidence response actions. The security council has no power to propose, delay, or veto actions. The exact size of the security council, its members, or the threshold of security council votes to allow immediate execution of a proposal has not been determined at the time of the audit.

The *Verifier* is a smart contract on layer 1 in charge of verifying zk-proofs. It exposes a function that returns a Boolean value indicating whether a given proof is valid. During the course of this audit this feature was out of scope due to the fact that it was not yet implemented.

The *validators* are a set of layer 1 addresses in charge of bundling transactions into blocks, executing them on layer 2, committing their compressed information to layer 1, requesting their zk-proof, and finalizing them on layer 1. Additionally, they are in charge of forwarding messages mathematically prevents validators from spoofing transactions or relaying false information. At the time of the audit, the validators are a centralized entity, while future decentralization is planned.

A set of four *system contracts* on layer 2 has privileged roles and performs special operations:

- The *Contract Deployer* is in charge of deploying contracts on layer 2 via a hash-commitment to the contract's bytecode.
- The *IL1Messenger* is a contract that allows the transfer of arbitrary-length messages from layer 2 to layer 1 by using a hash commitment within the fixed-size data exchange struct `L2Log`.
- The *INonceHolder* stores transaction and deployment nonces for accounts and exposes them via view functions.
- The *Bootloader* acknowledges received requests which have been passed from layer 1 to layer 2 via the priority queue mechanism.

Medium Severity

Corruption of facets array on selector replacement

The `Diamond` library allows replacing a selector's facet with itself which is non-compliant with EIP-2535.

Moreover, an edge-case in which a facet only has one selector and this selector's facet is replaced with itself leads to corruption of the `DiamondStorage.facets` array. Consider the following scenario:

1. A facet has an array of selectors containing only one element `[s1]`. Through the function `diamondCut` a call to `_replaceFunctions` is initiated.
2. Inside of `_replaceFunctions`, the call to `_saveFacetIfNew` does not add any facet, because the facet is already registered.
3. Inside of the loop iterating through the selector array `[s1]`, the call to `_removeOneFunction` triggers a call to `_removeFacet` due to the last selector being removed. This in turn removes the facet from the `ds.facets` array.

To be fully compliant with the EIP-2535 spec and mitigate the edge-case leading to a corruption of the facets array, consider adding the requirement that `_facet` and `oldFacet.facetAddress` are distinct from each other.

Update: *Fixed in commit `f6cde78`. The team mitigated this edge-case by rearranging the code. However, with the missing distinction check, the implementation is not fully EIP-2535 compliant.*

Freezable property applies to individual selectors instead of facets

In the `DiamondProxy` contract, a selector is mapped to a facet via the mapping `diamondStorage.selectorToFacet` upon executing the `fallback` function. The received datastructure of type `Diamond.SelectorToFacet` contains the information

```
address facetAddress, uint16 selectorPosition, bool isFreezable
```

The flag `isFreezable` is used to determine whether the `delegatecall` of the given selector to the respective facet should be executed.

At the same time, it is the stated intent of the system to allow freezability on the granularity level of facets: While the Diamond shall have a global flag to determine whether it is frozen or not. Each facet shall be marked as either freezable or not.

An issue arises, because the `selectorToFacet` mapping allows different values for the flag `isFreezable` for different selectors of *the same facet*. Which would allow for freezability on the granularity of selectors instead of facets. Consider the following example of two different selectors, one freezable, one not, belonging to the same facet:

```
selectorToFacet[selector1] = SelectorToFacet(facet1, 0, true)
selectorToFacet[selector2] = SelectorToFacet(facet1, 1, false)
```

Moreover, the initialization of the `selectorToFacet` mapping within the `Diamond` library in function `diamondCut` actually allows the assignment of different values for `isFreezable`. Consider the following example for the `facetCuts` array:

which will lead to an initialization of the `selectorToFacet` mapping given in the example above.

To prevent selector-level granularity of the freezabilitiy property, consider removing the `isFreezable` property from the `Diamond.SelectorToFacet` datatype and add it to a datatype describing only the facet thereby establishing a 1:1 mapping between facet and freezability.

Update: *Fixed in commit* *`e39eb07`*.

Merkle library verifies intermediate inputs

The `Merkle` library enables verification of a Merkle proof by performing an inclusion check of an input against a binary tree. This works by consecutively hashing concatenated sibling nodes until a root hash is generated. The input is one of the leaf hash values, while the proof is a path through the tree containing the missing hash information to regenerate the root.

An issue arises in this library, due to the arbitrary length of the proof. This allows shorter paths to resolve to the same root. Hence, the known hash of an intermediate node is a valid input as well. To visualize, considering the leaf nodes `h0` and `h1`, the hashed concatenation `hash(h0 || h1)` of those hashes would be a valid input along a shorter path. An attacker could utilize the known pre-image to prove its inclusion in the tree. For the standalone library this is a critical problem.

In this particular codebase the `Merkle` library is solely used in the `MailboxFacet` contact to prove the inclusion of a transaction within a set of layer 2 logs. Thus, only inputs of type `L2Log` with a length of 88 bytes are legitimate, while the pre-images of size 64 bytes contained within the Merkle tree are not. However, any future usage on 64 bytes input would lead to a critical vulnerability.

It was also stated that the incomplete tree of fixed size is filled with the default hash `hash("")`. This allows an attacker to prove the inclusion of empty bytes by default. Although, no threat was identified for the contracts in scope.

"leaf" wording for variable naming.

Update: *Fixed in commit `7eb51d9`. Additional checks have been applied outside of the Merkle library to filter malicious inputs. The library itself remains vulnerable to the attack if used in a different context. A note about this problem was added to the function documentation with commit `5f02309`.*

Proof replayability

In the `proveBlocks` function of the `ExecutorFacet` contract, there is no linkage between the committed blocks and the proof. The respective check is commented out in line 213. However, as it is commented out, the following is applicable.

The provided proof data is self-contained. Hence, the validator verifies that the given proof is valid in itself. Seeing the validator as a black box, it is assumed that there is no back checking against the committed blocks provided during the call. Therefore, the independence between the committed blocks and proof suggests a replay attack. By providing any formerly valid proof the previously committed blocks would be validated. Thus, all users could verify committed blocks, whether valid or not.

As documented in the code, the necessary check is there but commented out, which is based on the argument that the `Verifier` contract is not yet implemented. However, the commitment check between the blocks and proof has nothing to do with the `Verifier`. Therefore, consider incorporating this crucial check as part of the finalized codebase.

Update: *Fixed in commit `64d6aec`.*

Low Severity

`_proveBlock` while loop could run out of gas

In the `ExecutorFacet` contract within the `proveBlocks` function, there is a while loop to skip already verified blocks. The loop condition is defined as:

Therefore, if the committed blocks do no contain the first unverified block, this loop will eventually run out of gas and revert.

Consider limiting the number of loop iterations to the length of the `_committedBlocks` array and reverting with an expressive error message in case the block was not found.

Update: *Fixed in commit* `df107f0`.

`DiamondInit` can be initialized itself

The `DiamondInit` contract is designed to initialize the `DiamondProxy` or any new facet via a `delegatecall` from the proxy contract. Therefore, the `DiamondInit` contract is deployed on its own with an unprotected `initialize` function.

Hence, anyone could initialize the deployed instance of the `DiamondInit` contract itself. While this isn't identified as a threat, it is good practice to prevent arbitrary callers from initializing contracts.

Consider initializing the `DiamondInit` contract via the `constructor` or adding a security mechanism to the `initialize` function.

Update: *Fixed in commit* `c9089a5`.

`lastDiamondFreezeTimestamp` is unused

In the `DiamondCutFacet` contract, the diamond can be frozen to allow inspection of the protocol's security. Currently, as part of the `emergencyFreezeDiamond` function, `s.diamondCutStorage.lastDiamondFreezeTimestamp` is set but not used elsewhere in the code.

Consider either implementing a use-case for this variable or removing it.

Update: *Acknowledged, not fixed. The Matter Labs team states:*

> While this feature was not included into this release we prefer to keep the variable to facilitate the rollout of the feature once it is ready.

standard foresees moving individual selectors from facet to facet. Hence, one logical component (e.g. the `ExecutorFacet`) could be split into two facets, due to patching a single function and migrating the selector to the new facet, while the rest of the logic is kept in the old facet. As the first selector to the new facet defines the freezability, this could result in two different freeze capabilities for one high-level logical component (`ExecutorFacet`).

Implementing checks to cover the joint freezability for the logical facet would introduce additional overhead. Instead, consider extensively documenting this behavior in the `DiamondCutFacet` contract.

Update: *Acknowledged, not fixed. The Matter Labs team states:*

> We acknowledge that this issue raises a valid concern and we are aware that overall documentation improvement is due. This has been in our backlog already and we have now adjusted the priority accordingly.

Gas optimizations

Throughout the codebase there are multiple instances where gas costs can be optimized:

- In line 162 of the `Diamond` library the `uint16` cast is unnecessary and can be removed.
- Using the `delete` keyword instead of overwriting with the default value saves gas in these instances:
  - line 33 of Governance facet.
  - line 119-121 of `DiamondCutFacet` contract.
- In the `approveEmergencyDiamondCutAsSecurityCouncilMember` function, the `s.diamondCutStorage.currentProposalId` variable can be written to stack and reused.
- In the `ExecutorFacet` contract on line 113 and 155 two `bytes32` values are encoded and hashed. Consider using the `abi.encode` function for the encoding to be more gas efficient.
- In the `ExecutorFacet` contract, consider writing the `_maxU256` return value to stack to save on the following `s.totalBlocksCommitted` storage reads.

within the function `approveEmergencyDiamondCutAsSecurityCouncilMember`.

Interface and contract function parameter mismatch

The `revertBlocks` function has a different parameter name in `IExecutor` compared to `ExecutorFacet`. While in the interface the `_blocksToRevert` parameter suggests reverting a relative amount of blocks, the logic sets an absolute `_newLastBlock` which is confusing.

Further, in the `requestL2Transaction` function two parameters have a mismatch between the `MailboxFacet` and the `IMailbox` interface.

Consider correcting the above mismatches in favor of consistency and clarity.

Update: *Fixed in commit `c0600e0`.*

Getter returns misleading value

In the `GettersFacet` contract, the function `isFunctionFreezable` returns a Boolean value indicating whether a given selector is freezable or not. This value is taken from storage without any prior validation. At the same time, any uninitialized storage in Solidity contains the default value zero/false.

Querying the function `isFunctionFreezable` for an unknown selector will return `false`, thereby misleading the user to believe that the selector is used within the Diamond and is not freezable.

Consider validating the existence of the selector by requiring that the facet address of the selector is registered.

Update: *Fixed in commit `cfe6f54`.*

Lack of Documentation

Docstrings improve readability and ease maintenance. They should explicitly explain the purpose or intention of the functions, the scenarios under which they can fail, the roles allowed to call them, docstrings, it hinders reviewers' understanding of the code's intention and increases the maintenance effort for contributors.

Throughout the zksync-2-dev codebase there are several parts that do not have docstrings. For instance:

In the `Storage.sol` contract the following identifiers lack sufficient documentation:

- The `DiamondCutStorage` struct including all fields
- The `L2Log` struct including all fields
- The storage information of the `L2Message` struct as well as its `txNumberInBlock` field

In the `IMailbox.sol` interface the following constructs lack sufficient documentation:

- The `L2CanonicalTransaction` struct including all fields
- The `NewPriorityRequest` event including all fields

In the `Mailbox.sol` contract the following functions lack sufficient documentation:

- The `proveL2MessageInclusion` function
- The `proveL2LogInclusion` function
- The `l2TransactionBaseCost` function
- The `requestL2Transaction` function
- The `serializeL2Transaction` function

In the `IExecutor.sol` interface the following structs lack sufficient documentation:

- The `StoredBlockInfo` struct, especially fields `blockHash`, `indexRepeatedStorageChanges`, `stateRoot`
- The `CommitBlockInfo` struct, especially field `indexRepeatedStorageChanges`
- The `ProofInput` struct including all fields

In the `DiamondCut.sol` contract the following functions lack sufficient documentation:

- The `emergencyFreezeDiamond` function

In the `IGetters.sol` interface the following structs lack sufficient documentation:

- The `Facet` struct including all fields
- `SelectorExtended` struct including all fields
- `FacetExtended` struct including all fields

In the `Diamond.sol` library the following identifiers lack sufficient documentation:

- The `DiamondStorage` struct and all of its fields
- The `FacetCut` struct and all of its fields
- The `DiamondCutData` struct and all of its fields
- The `diamondCut` function
- The `getDiamondStorage` function

In the `PriorityQueue.sol` library the following identifiers lack sufficient documentation:

- The `Queue` struct and all its fields
- The `getLastProcessedPriorityTx` function
- The `getTotalPriorityTxs` function
- The `getSize` function
- The `isEmpty` function
- The `pushBack` function
- The `front` function
- The `popFront` function

Consider thoroughly documenting all structs and functions (and their parameters) that are part of the contracts' public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec). While applying the NatSpec tags, make sure to be consistent with the usage of the respective tag. For instance, use `@notice` for a general description and `@dev` for technical aspects.

Update: *Fixed in commit* `8abb05c`.

Lack of event information

- In the `ExecutorFacet` contract
  - the `proveBlocks` function does not emit an event after altering the storage variable `s.totalBlocksVerified`. Consider creating a new event that can emit both the old and new value of this variable.
  - the `BlocksRevert` event is used as `BlocksRevert(s.totalBlocksExecuted, s.totalBlocksCommitted)` which differs from its definition `BlocksRevert(uint256 totalBlocksVerified, uint256 totalBlocksCommitted)` within the `IExecutor` interface. Consider replacing `s.totalBlocksExecuted` with `s.totalBlocksVerified`.
- In the `IExecutor` interface
  - the `BlockCommit` event only contains the `blockNumber`, which might not be unique due to block reversion. Consider adding indexed fields for `blockHash` and `commitment`.
  - the `BlockExecution` event only contains the `blockNumber`. Consider adding indexed fields for `blockHash` and `commitment`.
- In the `IGovernance` interface the `NewGovernor` event emits the new governor. To ease tracking the responsibility of this important role, consider emitting both – the old and new governor – as *indexed* addresses.
- In the `IDiamondCut` interface
  - the `EmergencyDiamondCutApproved` event only emits the address of the approver, but no information about the diamondcut proposal. Consider indexing the `address` field and adding the fields `currentProposalId`, `securityCouncilEmergencyApprovals` and the indexed field `proposedDiamondCutHash`.
  - the `Unfreeze` event does not emit additional information. Consider adding the `lastDiamondFreezeTimestamp` as an event field.
  - the `DiamondCutProposalCancelation` event does not emit additional information. Consider adding a `currentProposalId` field and the indexed field `proposedDiamondCutHash`.

Update: *Fixed in commit `79fd845`.*

Lack of `l2Logs` validation

In the `ExecutorFacet` contract within the `commitBlocks` function, the array `_newBlock.l2Logs` is processed in the helper function `_processL2Logs`. While any L2 user can be the sender of `l2Logs`, three special senders can determine information influencing the hash commitment of the block. Most notably, the sender `L2_SYSTEM_CONTEXT_ADDRESS` can set the `previousBlockHash` and `blockTimestamp` information. Moreover, multiple `L2Logs` of this sender within one block would override each other.

While the system implicitly assumes that exactly one `L2Log` of sender `L2_SYSTEM_CONTEXT_ADDRESS` is present in each block, this assumption is not enforced during block commitment.

Consider enforcing that only one `L2Log` with sender `L2_SYSTEM_CONTEXT_ADDRESS` is present in each block.

Update: *Fixed in commit `dfc6fe1`.*

Preimage hash collision protection for storage pointers

The Diamond Proxy makes use of the diamond storage pattern to track the facets and selectors in use. This is achieved through a `DiamondStorage` struct that contains the relevant facet and selector information. Because of the proxy setup, this struct is placed in an unstructured-storage-manner at a pseudo random storage slot calculated by hashing a hardcoded string.

In the event of introducing a dynamic slot calculation using hashing, the `DiamondStorage` storage slot could be specifically addressed to force a collision using the known input bytes from above.

To prevent this pre-image hash collision, consider applying a `-1` offset to the hash.

Throughout the codebase there are `require` statements that require multiple conditions to be satisfied. For instance:

- The `require` statement on line 37 of `Executor.sol`
- The `require` statement on line 251 of `Diamond.sol`
- The `require` statement on line 17 of `Merkle.sol`

To simplify the codebase and to raise the most helpful error messages for failing `require` statements, consider having a single require statement per condition.

Update: *Fixed in commit `b87267c`.*

Confusing event emission when executing diamond cut proposals

In the `DiamondCutFacet` contract, the `executeDiamondCutProposal` function is used to execute a previously proposed upgrade.

This function resets the scheduled diamond cut proposal, thereby re-purposing the `_resetProposal` function. When this function is successfully executed it triggers a `DiamondCutProposalCancelation` event. Afterwards, `DiamondCutProposalExecution` event is triggered by `executeDiamondCutProposal` function.

This dual event emission of cancellation followed by execution could lead to confusion in off-chain systems.

To prevent emitting a cancellation event during the execution of a diamond cut proposal, consider moving the emission of the `DiamondCutProposalCancelation` event from the `_resetProposal` function to the `cancelDiamondCutProposal` function.

Update: *Fixed in commit `fad57c5`.*

Unused input to commit blocks

variable [illegible] is calculated from the [illegible] and included in the block commitment.

Consider validating both values against each other.

Update: *Fixed in commit* `615b6a5`.

Unused L2Messages can be committed to L1

In the `ExecutorFacet` contract the validator provides multiple blocks of type `CommitBlockInfo` to the `commitBlocks` function, which are validated in several steps including `_processL2Logs`. In this function, the preimages contained in the `_newBlock.l2ArbitraryLengthMessages` array are checked against the hashes contained in `L2Logs` with sender `L2_TO_L1_MESSENGER`.

However, the counter `currentMessage` is incremented only based on the `L2Log` information, without taking the length of the `_newBlock.l2ArbitraryLengthMessages` into account. In effect, the `_newBlock.l2ArbitraryLengthMessages` array can be longer than the number of relevant `L2Log`s contained in the `_newBlock.l2Logs` parameter which might be confusing to the validator and to off-chain receivers of the respective call data.

Consider the addition of a final check of `currentMessage` against the length of the `_newBlock.l2ArbitraryLengthMessages` array at the end of the `_processL2Logs` function to ensure that only relevant preimages have been included in the calldata.

Update: *Fixed in commit* `25913e7`.

Unverified inputs during block commitment

In the `ExecutorFacet` contract the `_commitOneBlock` function takes the `_newBlock` parameter to validate, extract, and transform block information into a `StoredBlockInfo` struct, which is hashed and stored on chain as a commitment. With a valid zero-knowledge proof this data can later be verified and executed.

Consider preventing the commitment of unexecutable blocks by:

- Validating the field `numberOfLayer1Txs` against the number of `L2Log`s with sender `L2_BOOTLOADER_ADDRESS`.
- Validating the field `l2LogsTreeRoot` against a reconstruction of the Merkle tree from the `l2Logs` array.
- Validating the field `timestamp` against the local variable `blockTimestamp`.

Update: *Fixed in commit `230f400`. The Matter Labs team states:*

> We have applied the recommendations #1 and #3. The recommendation #2 is redundant as it is already covered by zero knowledge proofs. Verifying this on Layer 1 would be too expensive so by design this is entrusted to zero knowledge cryptography.

Notes & Additional Information

AppStorage partially lacks getter functions

The `GettersFacet` contract does not expose the entire `AppStorage` via `view` functions. The following storage parts remain inconvenient to read for an outside actor:

- Every aspect of `diamondCutStorage`
- The `pendingGovernor` address
- The `storedBlockHashes` mapping
- The functions `getSize` and `front` of `priorityQueue` as well as a function to determine the position of elements within the queue

Additionally, there is an input size mismatch between the `l2LogsRootHashes` mapping, which takes an `uint256` key, and the `l2LogsRootHash` getter function, which takes a `uint32` parameter.

Consider exposing all relevant information via getter functions and ensure that function parameters and mapping keys are type-identical.

Update: *Fixed in commit `6b99055`.*

OpenZeppelin

the `IExecutor` interface have the following parameters in common:

- `blockNumber`
- `indexRepeatedStorageChanges`
- `numberOfLayer1Txs`
- `priorityOperationsHash`
- `l2LogsTreeRoot`
- `timestamp`

Consider moving these parameters to a separate `BaseBlockInfo` struct which is then included into `StoredBlockInfo` and `CommitBlockInfo` respectively. Note, this will affect the way the variables are accessed.

Update: *Acknowledged, not fixed. The Matter Labs team states:*

> We acknowledge that this issue raises a valid concern. It does not pose a security risk, so we have added it to our development backlog.

Confusing identifier names

Throughout the codebase we found multiple occurrences of identifier names creating confusion:

- In the `IExecutor` interface the parameter name used in the signature of the `revertBlocks` function is `_blocksToRevert`. However, the implementation of said function within `ExecutorFacet` calls the same parameter `_newLastBlock` which is consistent with its usage.
- In the `PriorityQueue` library the variables `head` and `tail` account for where to add and remove items from the list. Unintuitively, the `head` points to the end of the queue and `tail` points to the front of the queue.

Consider renaming confusing identifiers to prevent misunderstandings and incorrect usage.

Update: *Fixed in commit `03e04cb` and `c0600e0`.*

Direct usage of library struct fields

However, it is best practice to decouple the internal structure of a library from the functionality it exposes to other contracts through its functions.

Consider replacing the direct access of struct field `head` with a call to the getter function `getTotalPriorityTxs` that exposes the same information.

Update: *Fixed in commit `8b97af`.*

Lack of ERC-165 support

External contracts and third-party integrations have no means to discover interfaces supported by the codebase.

Consider implementing the `supportsInterface` function of the `ERC-165` standard to expose information about implemented interfaces.

Update: *Acknowledged, not fixed. The Matter Labs team states:*

> We acknowledge that this issue raises a valid concern. It does not pose a security risk and comes with additional maintenance overhead if the system is expected to change in the future. We have added it to our development backlog to be addressed once we are out of alpha version.

File and contract name mismatch

There is a general mismatch between the facet contract names and their file names. While the contracts have a “Facet” suffix, the files have not. The following contracts are affected:

- `DiamondCutFacet`
- `ExecutorFacet`
- `GettersFacet`
- `GovernanceFacet`
- `MailboxFacet`

Regarding the interfaces, the individual interface names are based on the filename, e.g. “IDiamondCut”, but instead should align with the contract name.

Update: *Acknowledged, not fixed. The Matter Labs team states:*

> We acknowledge that this issue raises a valid concern. It does not pose a security risk, so we have added it to our development backlog.

Inconsistent NatSpec tags

Throughout the codebase the NatSpec docstring tags `@notice` and `@dev` are used interchangeably. The Solidity NatSpec documentation describes the tags as the following:

- `@notice` – Explain to an end user what this does
- `@dev` – Explain to a developer any extra details

Consider applying the respective descriptions across the documentation to be consistent.

Update: *Acknoledged, not fixed. The Matter Labs team states:*

> We acknowledge that this issue raises a valid concerns and we are aware that overall documentation improvement is due. This has been in our backlog already and we have now adjusted the priority accordingly.

Misleading documentation

The docstring documentation of the `_executeOneBlock` function is misleading by stating the following:

> Processes all pending operations (**Send Exits**, Complete priority requests)

However, sending exits is not part of the implementation. Consider revising the comments to accurately reflect the logic.

Update: *Fixed in commit `ce78828`.*

Uninformative reason strings

The codebase uses short alphanumeric codes instead of understandable reason strings in require statements.

Consider including specific, informative error messages in `require` statements to improve overall code clarity and to facilitate troubleshooting whenever a requirement is not satisfied.

Update: *Partially fixed in commit `79f6a36` by adding the missing error message. In addition, the Matter Labs team states:*

> Improving the error messaging is a large effort that we have added to the backlog for now.

Solidity compiler version is not pinned

Throughout the codebase there are `pragma` statements that allow multiple versions of the Solidity compiler, including outdated versions.

Consider taking advantage of the latest Solidity version to improve the overall readability and security of the codebase. Regardless of which version of Solidity is used, consider pinning the version consistently throughout the codebase to prevent bugs due to incompatible future releases and take into account the list of known compiler bugs.

Update: *Acknowledged, not fixed. The Matter Labs team states:*

> The version is pinned, but not directly in the source files to speed up the development: https://github.com/matter-labs/zksync-2-dev/blob/openzeppelin-audit/contracts/ethereum/hardhat.config.ts#L82. It is in the backlog to pin it in the source files after the release.

TODO comments in the code base

We found the following instances of TODO comments in the codebase that should be tracked in the project's issues backlog and resolved before the system is deployed:

- Line 17 of `Config.sol`
- Line 50 of `Storage.sol`
- Line 212 of `Executor.sol`
- Lines 70, 94 and 136 of `Mailbox.sol`
- Line 20 of `IDiamondCut.sol`

Consider tracking all instances of TODO comments in the issues backlog and linking each inline TODO to the corresponding backlog entry. Resolve all TODOs before deploying to a production environment.

Update: *Acknowledged, not fixed. The Matter Labs team states:*

> We acknowledge that this issue raises a valid concern. It does not pose a security risk, so we have added it to our development backlog.

Typographical errors

Throughout the codebase there were a few typographical errors. For instance:

- line 105 of the `Diamond` library: "Add facet to the list of facets if the facet address is ***a*** new one"
- line 192 of the `Diamond` library: "It is expected but NOT enforced that `_facet` is ***a*** NON-ZERO address"
- line 295 of the `ExecutorFacet` contract: "_blockAuxilaryOutput" should be "_blockAuxiliaryOutput"

Consider correcting the above and any other typos in favor of correctness and readability.

Update: *Acknowledged, not fixed. The Matter Labs team states:*

> We acknowledge that this issue raises a valid concerns and we are aware that overall documentation improvement is due. This has been in our backlog already and we have now adjusted the priority accordingly.

Unorganized file layout

The `Diamond` library has an unorganized layout of code contents, which mixes functions, structs, enums, and events in no particular order. More specifically, we found the following code order in the file:

> constants -> structs -> function -> enum -> structs -> function -> event -> functions

Unused named return variable

The `requestL2Transaction` function declares a named return variable `bytes32 canonicalTxHash` in its signature, but uses an explicit `return` statement in its body.

Consider either using or removing the named return as well as applying a consistent style of returning variables.

Update: *Fixed in commit `923b3c3`.*

Write array length to stack to save gas

In the EVM it is more gas-efficient to read values from stack than from memory or storage. As values are read repeatedly within for loops, it makes sense to write the length of an array to the stack and reuse the stack-variable.

Throughout the codebase we found multiple instances to which this optimization could be applied:

- On line 100 of file `Executor.sol`
- On line 69 of file `Diamond.sol`
- On line 108 of file `Diamond.sol`
- On line 131 of file `Diamond.sol`
- On line 148 of file `Diamond.sol`

Consider writing the array length to stack by assigning it to a local variable and then using the variable to reduce gas consumption.

Update: *Fixed in commit `2987c69`.*

Conclusions

Over the course of four weeks we audited this layer 1 building block of the zkSync 2.0 project. We're excited to see Matter Labs making this step and developing the first EVM-equivalent zero-knowledge-based rollup. It goes without saying that this protocol is highly complex, but Matter Labs was really responsive and helpful clarifying any doubts we had and provided dedicated

Appendix

Monitoring Recommendations

While audits help in identifying code-level issues in the current implementation and potentially the code deployed in production; we encourage Matter Labs team to consider incorporating monitoring activities in the production environment. Ongoing monitoring of deployed contracts helps in identifying potential threats and issues affecting production environment. Hence, with the goal of providing a complete security assessment we want to raise several actions addressing trust assumptions and out-of-scope components that can benefit from on-chain monitoring.

Upgrades – The Diamond pattern that defines the code structure allows upgrading the logic of this protocol. Any upgrade must be initiated by the governor via a diamond cut proposal and a subsequent execution. This process must undergo a time delay or requires the security council's approval for a quicker upgrade. In this context, consider monitoring these events:

- `DiamondCutProposal`
- `DiamondCutProposalCancelation`
- `DiamondCutProposalExecution`
- `EmergencyDiamondCutApproved`

This would allow the detection of the following suspicious activities:

- The introduction of malicious code either as part of a facet or as part of the initializer contract.
- Any diamond cut proposal including an initializer address at which no contract has been deployed so far.
- The init calldata on proposal execution is malicious.
- An unrealistically short time delay between council members' approvals.

Freezability – Further, a governor controlled mechanism was implemented to freeze all freezable facets. In that case the `EmergencyFreeze` and `Unfreeze` events are emitted. An unplanned emergency freeze outside of incident response measures could indicate that the governor role is compromised and performing a DoS attack, therefore consider monitoring the respective events.

address accepts the administrative rights, a `NewGovernor` event will be emitted. Finally, whether a new address is set as a validator or an existing validator changes its state, a `ValidatorStatusUpdate` event will be emitted. Consider monitoring these events to detect unexpected changes to the governor or validators, both of which could signal a compromised governor role.

Executor – Blocks submitted to the layer 1 contract typically undergo a three stage process: commit, prove, and execute. Consider monitoring any deviation from this process as it might indicate the following malicious activities performed by rogue validators (censoring or DoS attacks):

- A transaction reverts due to wrong data while aiming to prove or execute a block.
- Any usage of the block reversion function firing the `BlocksRevert` event.

Mailbox – To ensure the correct and timely operation of layer 1 (L1) to layer 2 (L2) communication, consider monitoring each invocation of the `requestL2Transaction` function via the `NewPriorityRequest` event, as well as the calldata of each `Executor.commitBlocks` and `Executor.executeBlocks` invocation. This will allow the computation of time deltas between request and inclusion for each L1->L2 transaction. Furthermore, the detection of censorship through dropped transactions will be possible.

Related Posts

Beefy

Zap Audit

OpenZeppelin

BRUSHFAM

OpenBrush Contracts Library Security Review

OpenZeppelin

Linea

Bridge Audit

OpenZeppelin

BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

Security Audits

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits

OpenZeppelin

Defender Platform

Secure Code & Audit
Secure Deploy
Threat Monitoring
Incident Response
Operation and Automation

Services

Smart Contract Security Audit
Incident Response
Zero Knowledge Proof Practice

Learn

Docs
Ethernaut CTF
Blog

Company

About us
Jobs
Blog

Contracts Library

Docs