OpenZeppelin | security

# Scroll ZKTrieVerifier Audit


March 6, 2024

# Table of Contents

# Summary

| | |
|---|---|
| Type | Layer 2 |
| Timeline | From 2024-01-25<br>To 2024-02-16 |
| Languages | Solidity |

| | |
|---|---|
| Total Issues | 8 (4 resolved, 1 partially resolved) |
| Critical Severity Issues | 0 (0 resolved) |
| High Severity Issues | 0 (0 resolved) |
| Medium Severity Issues | 1 (1 resolved) |
| Low Severity Issues | 4 (1 resolved) |
| Notes & Additional Information | 3 (2 resolved, 1 partially resolved) |

# Scope

We audited the `scroll-tech/scroll` repository at the `c68f428` commit.

In scope were the following files:

```
contracts/src
├── L1/rollup
│   └── ScrollChainCommitmentVerifier.sol
└── libraries/verifier
    └── ZkTrieVerifier.sol
```

# System Overview

The `ZkTrieVerifier` library consists in an implementation that defines a single method used to verify if a certain proof is contained inside a root hash.

The `ScrollChainCommitmentVerifier` contract is the entrypoint to verify the proof. The contract has 2 external functions: the `verifyZkTrieProof` function consist in a simple wrapper of the library's method, which does not provide extra validations, and the `verifyStateCommitment` function performs the necessary checks against the `ScrollChain` contract to validate if the proof is contained inside a particular batch by comparing the recreated `rootHash` against the one in the finalized batch. Furthermore, it returns the storage value for the particular proof in the passed account. These contracts can then be used to validate if a certain proof for a storage in L2 is committed in L1.

However, one particularity of the protocol, is that it uses a Poseidon Hash Algorithm to calculate more efficiently the hashes. This means that all hashing operations are being redirected to a circuit with the implementation of the Poseidon hash. A characteristic of its use is that as the secure key space does not occupy the full range of `2 ** 256` values, the protocol truncated the key to the lower 248 bits, which means that the depth of the Trie cannot exceed such level.

## Limitations of the 'zkTrieVerfier' Verification

The `ZkTrieVerifier` library is meant to validate certain scenarios with the queried proofs. However, there might be a few situations in which the library will end up in revertion, even when using the proofs from the node. In particular, some of the cases seen were:

- Accounts that do not have code and its nonce is zero
- Accounts that do not have code, its nonce is zero, and it has zero balance
- Proofs from contracts on empty storage slots

Moreover, users and developers must be aware that the on-chain usage of the `ZkTrieVerifier` library on any other protocol to validate a proof before taking an action will increase the gas execution by a significant amount, which depends on the depth of the proof.

# Advanced Testing Analysis

In addition to manual review of the codebase, the engagement also incorporated advanced testing analysis to enhance the coverage of the audit and assess how the `ZkTrieVerifier` library will behave under various scenarios. Sections below outline the key steps of the work performed.

## Analysis of the state of the Original Tests

The test suite found for the respective codebase is built on Hardhat in, mainly, a TypeScript file. Such set of tests aim at validate a few set of cases. In particular, the test suite:

- Takes proofs from 5 queries from the Scroll mainnet, and verifies that such proofs are valid using the library.
- Edge cases of malformed proofs are created based on the first example of the hardcoded cases. Then, after alterations are being made, the proof is sent to validate that the verification will revert. Between those, it is being tested:
  - 5 successful single cases of non-empty storage in a contract and empty storage in a contract.
  - Reverted single cases of invalid node types, invalid branch hash, invalid keys, invalid flags, invalid leaf hashes, invalid preimage and preimage lengths, and invalid Magic Bytes separators.
- All edge cases are single situations, and it is not being tested a diversity of situations that could alter the behavior during the execution.
- Cases like the 4th one from the hardcoded examples, aim at verifying a "random empty storage in some contract", whereas specification given by the Scroll team was that such scenarios should revert. Moreover, similar scenarios created on a local environment has shown that the `storageProof` for those kind of queries would return no elements such as the `nodeType` and the "Magic Bytes", ending up in a revert scenario.
- Tests for the protocol's contract are written in Foundry, but the ones for the `ZkTrieVerifier` library are created with a different environment, which makes it harder to maintain.

Even though both positive and negative cases had been implemented, such tests were created to tackle specific cases and values, reducing the possibility of finding an input combination that could break the assumptions made with each case.

Moreover, the cases follow the pattern which the developer has programmed to do, meaning that those cannot deviate from the path chosen to verify either its successful outcome or not, reducing the possibility of finding unexpected situations.

# Objective

For this engagement:

- We explored the current test suite to identify improvement opportunities or gaps.
- We implemented fuzzing tests on such gaps so as to reduce the attack surface by using inputs that could trigger undesired behaviors.
- We bring recommendations on how to reduce the attack surface by increasing the testing scenarios.

As mentioned above, this effort is a step forward in the direction of increasing the security of the protocol by the usage of tools, rather than a complete solution.

# Test Details

In order to test more broad scenarios, the focus of the engagement has been put into identifying, from the manual review and the current test suite, the potential spots where more cases need to be tested.

As an initial approach, due to being a library that does not interact with the storage, fuzzing implementations were added on the most critical cases. Moreover, as the `ScrollChainCommitmentVerifier` contract only wraps the library while doing a few checks, the tests were targeted directly at the library without using the latter contract, in a way to simplify the deployment setup, as it would be needed to have a `ScrollChain` contract deployed plus a system that would replace the relayer and sequencers.

Due to the characteristics of the node towards the usage of the Poseidon Hash algorithm to construct the zkTrie and the proofs, a local Scroll L2 node was used to create transactions and be able to modify the storage of the chain. With it, it was possible to get later the proofs of the particular slots/accounts that would then later allow us to feed back into the verifier to find possible edge cases.

Moreover, it was needed the deployment of such Poseidon bytecode into the Foundry testing chain to link the verifier to such contract, as it was used by the verifier during the verification stage to reconstruct the hashes.

In order to interact between the tests and the node, external endpoints have been created that would handle the parameters from the fuzzing runs while also deploying the contracts, setting on storage, and getting the proofs.

The additional tests were divided into the groups mentioned below.

### Fuzzing

The behavior was tested and divided into different situations:

- ZkTrieVerifierTest.test_singleProof: reproduce one of the tests written in TypeScript but on Foundry.
- ZkTrieVerifierTest.test_maxProofDepth: isolate the `walkTree` functionality and test the depth limits.
- ZkTrieVerifierTest.test_deployContractAndValidateTheStorage: a contract is deployed which sets in storage random values. The test then grabs the proof of one of those slots and verifies it against the library.
- ZkTrieVerifierTest.testFail_accountProofLengthChange: similarly to the previous one, although the proof is altered by changing the length associated to the `accountProof` set, in which the test should revert unless the length is the original.
- ZkTrieVerifierTest.test_deployOnceAndTest: test in charge of grouping alterations on the "Extra" element in the proof array (element before the "Magic Bytes" element). The deployment and proof generation is still similar to the cases from above, but each proof is tested individually against the following situations:
    - Account's nodeType has been swapped for the other possible case
    - Account's nodeType has been changed to an erroneous constant
    - Account's nodeKey has been changed to an erroneous constant
    - Account's compressedFlag has been changed to an erroneous constant
    - Account's nonce and codesize slot has been changed to an erroneous constant
    - Account's balance has been changed to an erroneous constant
    - StorageRoot has been changed to an erroneous constant
    - Acccount's keccakCodeHash has been changed to an erroneous constant
    - Account's poseidonCodeHash has been changed to an erroneous constant
    - Account's keyPreimageLength has been changed to an erroneous constant
    - Account's keyPreimage has been changed to an erroneous constant
    - Account's keyPreimage padding has been changed to an erroneous constant
    - Storage's nodeType has been swapped for the other possible case
    - Storage's nodeType has been changed to an erroneous constant
    - Storage's nodeKey has been changed to an erroneous constant
    - Storage's compressedFlag has been changed to an erroneous constant

    - Storage's value has been changed to an erroneous constant
    - Storage's keyPreimageLength has been changed to an erroneous constant
    - Storage's keyPreimage has been changed to an erroneous constant

All fuzzing tests were run over the course of 24 hrs when testing up to 10000 runs, depending on the test. To increase that value, change the number for the runs parameter in the foundry.toml file under the [fuzz] section. The details of the set-up and work performed can be found on the zkverifier-fuzz testing github repo.

# Limitations

Although this is a step to increase the covered attack surface by randomizing both interactions and parameters over the protocol, there are a few limitations. In particular:

- Finding cases is a highly resource intensive process, requiring a suitable machine to increase the number of runs.
- Having to adapt an external node to the tests caused that the outgoing external calls through the FFI calls increased the time needed for each run/operation.
- The usage of a different hashing algorithm caused that other tools used for proofs generations might have not worked without any core alteration on the way the hashes and structure were created, meaning that it took effort in setting up the environment of the protocol to then be used for fuzzing. Moreover, the nodes resource consumption meant that not all machines were able to take the task of running such nodes at the same time of running the fuzz test suite.
- There is no time value that guarantees no case will cause undesired behaviors -- the more time the suite is run, the more likely the suite will find a possible case/exploit.
- Certain limitations on the verifier were imposed by the specifications, meaning that outputs that came from the node's `getProof` might not work directly on the verifier (as mentioned earlier).
- There are more fuzzing test cases opportunities that can be tested but have not been implemented due to time constrains.
- It was seen that several of the test cases found in the original suite could be converted into fuzzing scenarios, which might be beneficial to catch possible inputs that could cause issues in the implementation without much additional effort.
- The usage of 2 different setup environments for running the tests introduces more friction at the time of maintaining the code, but also at the time of implementing new tests that might cover both sides.

These set of tests are not meant to be comprehensive enough to validate the entire codebase, nor all different aspects of the mentioned implementation, and it should be taken as a guide to improve the current set of test.

## Additional Testing Recommendations

As for future improvements, consider implementing further tests on the following places:

- Fuzz the Poseidon contract directly
- Instead of using the constants in the "Extra" element, it should be random bytes for each case
- Craft transactions that would use the "Magic Bytes" inside of either the proofs or inside the "Extra" element, while adding extra data that would make the methods pass the separation checks.
- Test the scenario of having an account or storage proof length that would underflow the `walkTree` node's variable, while adding random data to the proofs (in order to make it pass).
- Adapt the current test cases to have randomized alterations to the proofs for each of the edge cases presented there. Moreover, introduce bit-flips that could attempt to pass the verification on wrong proofs.
- Craft proofs whose length prefix parameter is one, and assert that later requirements catch undesired cases that have the default zero values returned from the `walkTree` method.

## Summary

Initially identified key gaps in the original test suite coverage have been addressed by fuzzing the most critical scenarios, especially around the nodeType switches and alterations in the "Extra" element on possible checks that would still go through. Although, additional testing scenarios were added, the coverage of the test suite can still be expanded. It is recommended to leverage the current set-up provided that can be re-used with minor alterations, to continue the execution of more test with powerful devices to find potential paths to exploit.

The current setup of the test environment due to the protocol's characteristics and the usage of the Poseidon Hash Algorithm for the Trie and the proofs, increased the friction for getting the fuzzing test to work. Moreover, having different environments for testing adds friction and can possibly raise maintenance issues.

During the course of several hours of computation, none of the tests have shown any attack vector or exploitation case, instead, list of proof types that were not able to run with the verifier were identified (refer to **Limitations of the 'zkTrieVerfier' Verification** section above). Please note that our testing work was not fully exhaustive and cannot be taken as a guarantee that no unexpected values might be found by continuing to run the suite.

# Medium Severity

## M-01 Malicious User Can Increase the Gas Cost of Verification

The `ZkTrieVerifier` library verifies proofs that come from the Scroll L2 node which uses a sparse binary Trie as the data structure. For the verification process, the proofs have the necessary node hashes to reconstruct the root hash based on the queried leaf. The `walkTree` method used for going through all the levels of the Trie uses the bits of the key provided to define where the path should continue (left or right). However, a malicious user can influence how many levels the proof must go through by predicting addresses (or storage slots) that would branch the Trie until a certain depth.

Even though artificially increasing the proof's depth of a certain account or storage will not cause a DoS scenario, since the depth can still reach the maximum depth size in the worst-case scenario, artificially increasing the proof's depth will increase the number of iterations the `walkTree` method has to perform in order to reach the respective leaf. If protocols that use this verifier limit the gas used on-chain to perform such a verification (to a reasonable value), then a malicious user might be able to increase it for a particular transaction by reaching a similar hashed key to a certain depth in the Trie.

As such, consider letting users and developers know about this possible attack vector so as to not limit the gas used in such scenarios. Otherwise, the verification might fail.

***Update:*** *Resolved in pull request #1135 at commit `265800f`.*

# Low Severity

## L-01 Node Type Check Uses Underflow to Define Range

In the `walkTree` function of the `ZkTrieVerifier` library, a requirement checks whether the `nodeType` is within the expected range. However, the operation is done by subtracting 6 units from the `nodeType` and then checking if the resulting value is less than 4. In the expected

range of 6 to 9, this operation does not present an issue, but for node types below 6, the requirement relies on an underflow that would make the result greater than 4, reverting the validation.

To follow the best practices for reducing error-proneness, lowering the attack surface, and improving the readability of the code, consider splitting the requirement into two different operations to validate the respective `nodeType` range.

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

> *No node types are below 6, so the current code works as expected.*

## L-02 Use of Implicit Default `rootHash` and `expectedHash`

When no proofs are passed to the `walkTree` function from the `ZkTrieVerifier` library, the function skips the `for` loop over the nodes and implicitly returns the default value of 0 for the `rootHash` and `expectedHash` outputs. These values are then used as in the `verifyStorageProof` function to continue with the validation. However, using the default value without any explicit assignment, especially when handling assembly code and memory pointers, might result in undesired outcomes.

As such, to reduce the attack surface, consider explicitly handling the scenario in which the `walkTree` function does not have a proof to go through.

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

> *We are sure that the default value of `rootHash` and `expectedHash` is zero, so no assignment is needed.*

## L-03 Unbounded `walkTree` Due to Underflow

In the `ZkTrieVerifier` library, the `walkTree` function calculates the nodes as the length passed with the proofs, minus one. However, if a user sets the length as if there are no proofs, the `nodes` variable will be `type(uint256).max` as it will underflow due to a lack of safe arithmetic operations in `assembly`. This would cause a situation whereby the loop handles data beyond the range of the calldata where the respective proofs should end.

Although this issue does not cause a problem by itself, as it will probably revert due to checks during the `walkTree` function, it can still be used as a tool to craft a malicious proof with the intention of triggering another issue.

In favor of reducing the attack surface, consider checking the lengths against the passed proofs.

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

> *If a user sets the length as if there are no proofs, the function will revert. So, it will not be a problem.*

## L-04 Trie Depth Is Not Explicitly Capped

By using the Poseidon Hash, the node uses a maximum depth of 248 levels. However, the `ZkTrieVerifier` library does not impose a limit on the first byte that represents the length added to the proofs, making it possible to pass proofs with a length that exceeds the limit and continue with the operation. Even though it is challenging to craft a proof that would allow a user to pass the validations performed when constructing the leaf hash, not asserting the length exposes some un-mitigated attack surface which could be targeted by a malicious user in a different attack.

Consider asserting that the maximum depth is not exceeded by the crafted proof.

***Update:*** *Resolved in pull request #1137 at commit* `daaf600`.

# Notes & Additional Information

## N-01 Inconsistent Naming Convention

The `ZkTrieVerifier` library uses two naming conventions for the functions' names. In particular, functions associated with the hashing operation (`poseidon_hash` and the `hash_uint256` functions) are written in snake case, whereas other functions such as the `walkTree` function are written in camel case.

Consider using a consistent naming style throughout the codebase.

***Update:*** *Resolved in pull request #1138 at commit `db8f65d`.*

## N-02 Inconsistent Integer Base in Inline Assembly

The `ZkTrieVerifier` library makes use of inline assembly for multiple features. When performing these calculations, the decimal and hexadecimal integer bases are used interchangeably. For instance, `1` and `0x1` are used to move the memory pointer.

Consider sticking to one integer base for memory pointer movements and any other operations to improve the readability of the codebase and prevent calculation errors.

***Update:*** *Partially resolved in pull request #1139 at commit `d280b6c`. There are still cases, such as the addition in line 99, that are not consistent with the rest of the code. The Scroll team stated:*

> *`depth` is not pointer, so we use `1` instead of `0x01`.*

## N-03 Incorrect Function Visibility

The `verifyZkTrieProof` function of the `ScrollChainCommitmentVerifier` contract is not called internally by this contract.

Consider setting this function's visibility to `external` instead of `public` in order to reduce the attack surface.

***Update:*** *Resolved in pull request #1140 at commit `bb781f1`. The `verifyStateCommitment` function now uses the `verifyZkTrieProof` function from the `ScrollChainCommitmentVerifier` contract instead of calling the library directly.*

# Conclusion

The audit and the additional testing advisory engagement yielded recommendations to improve the overall quality and health of the codebase.

The codebase is well-written and has proper documentation. However, it could benefit from a more descriptive reasoning about the integration with the rest of the protocol, specially with the node's code, and its limitations. Furthermore, there are opportunities to improve the test suite, and to increase the test coverage.

The Scroll team was very responsive throughout the audit period and provided us with information regarding the operation of the node.