OtterSec

Light Protocol Zero Copy

Security Assessment

March 18th, 2025 — Prepared by OtterSec

Ajay Shankar Kunapareddy d1r3wolf@osec.io

Robert Chen r@osec.io

Table of Contents

Executive Summary **2**

Overview 2

Key Findings 2

Scope **3**

Findings **4**

Vulnerabilities **5**

OS-LZC-ADV-00 | Denial of Service due to Strict Comparison Checks 6

OS-LZC-ADV-01 | Incorrect Generic Type Utilization 7

General Findings **8**

OS-LZC-SUG-00 | Code Maturity 9

Appendices

Vulnerability Rating Scale **11**

Procedure **12**

01 — Executive Summary

Overview

Light Protocol engaged OtterSec to assess the `batched-merkle-tree` and `zero-copy` crates. This assessment was conducted between February 22nd and March 13th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 3 findings throughout this audit engagement.

In particular, we identified a vulnerability in the rollover process, where a strict comparison is utilized for the lamports of old accounts, which may result in a denial of service scenario as anyone may send extra lamports to these accounts, potentially disrupting the rollover process (OS-LZC-ADV-00). Additionally, there is a mismatch between the generic type utilized in `from_account_info` and the expected queue type, which may result in type inconsistencies (OS-LZC-ADV-01).

We also made recommendations regarding modifications to the codebase for improved clarity, and maintainability, and to ensure adherence to coding best practices (OS-LZC-SUG-00).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/Lightprotocol/light-protocol. This audit was performed against commits b9eebe4.

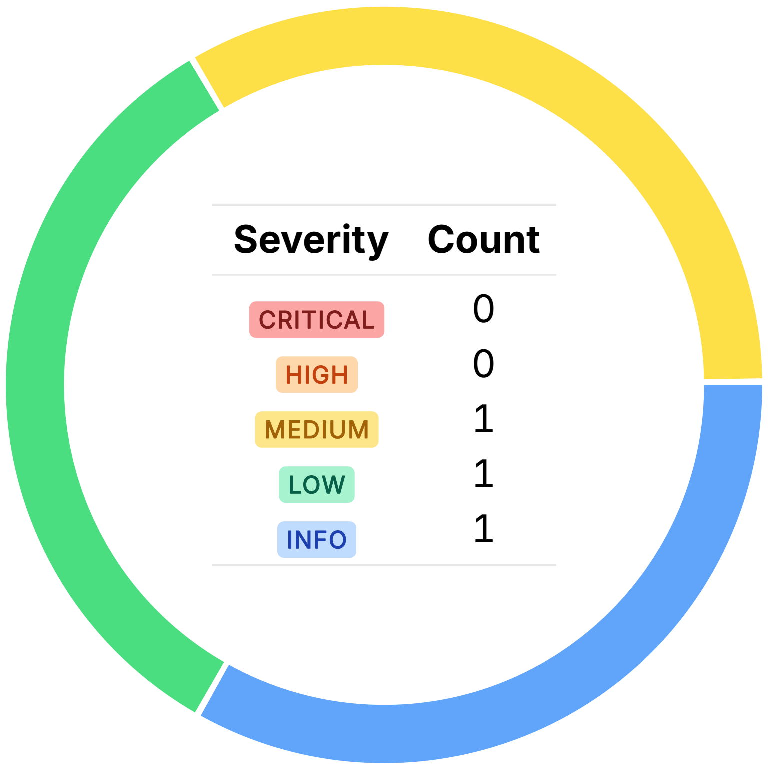
A brief description of the programs is as follows:

Name	Description
batched-merkle-tree	efficiently batches and inserts elements into a Merkle tree using zero-knowledge proofs (ZKPs).
zero-copy	A wrapper around Rust's `zerocopy` library to enable true zero-copy operations.

03 — Findings

Overall, we reported 3 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	0
MEDIUM	1
LOW	1
INFO	1

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-LZC-ADV-00	MEDIUM	RESOLVED ✓	During the rollover process, a strict comparison is utilized for the lamports of old accounts, which may result in a denial of service scenario, as anyone may send extra lamports to these accounts, potentially disrupting the rollover process.
OS-LZC-ADV-01	LOW	RESOLVED ✓	There is a mismatch between the generic type utilized in `from_account_info` and the expected queue type, which may result in type inconsistencies.

Denial of Service due to Strict Comparison Checks MEDIUM OS-LZC-ADV-00

Description

The issue arises from the strict comparison utilized for verifying rent exemption in the old Merkle tree and output queue accounts when performing a rollover in `rollover_batched_address_tree_from_account_info` and `rollover_batched_state_tree_from_account_info`.

>_ *program-libs/batched-merkle-tree/src/rollover_address_tree.rs* RUST

```rust
pub fn rollover_batched_address_tree_from_account_info<'a>(
    old_account: &AccountInfo<'a>,
    new_account: &AccountInfo<'a>,
    network_fee: Option<u64>,
) -> Result<u64, BatchedMerkleTreeError> {
    let new_mt_rent = check_account_balance_is_rent_exempt(new_account,
        ↪  old_account.data_len())?;
    #[cfg(target_os = "solana")]
    if old_account.lamports().checked_sub(new_mt_rent).unwrap() == 0 {
        return Err(MerkleTreeMetadataError::NotReadyForRollover.into());
    }
    [...]
}
```

This strict comparison assumes that the old accounts contain exactly the expected number of lamports before and after the rollover. However, this assumption is flawed because anyone may send additional lamports to these accounts. Consequently, it enables an attacker to disrupt the rollover process by artificially inflating the balance of the old accounts.

Remediation

The function should ensure that the balance is at least rent-exempt, instead of checking for the exact balance.

Patch

Resolved in 119a26e.

Incorrect Generic Type Utilization LOW OS-LZC-ADV-01

Description

In `from_account_info`, the generic type for the `from_bytes` method is set to `OUTPUT_QUEUE_TYPE`. However, `QUEUE_TYPE` is the generic type parameter for `from_account_info` itself. This implies that, in the current design, the queue type (`QUEUE_TYPE`) is never actually utilized in the call to `from_bytes`.

>_ *program-libs/batched-merkle-tree/src/queue.rs* RUST

```rust
pub fn output_from_account_info(
    account_info: &AccountInfo<'a>,
) -> Result<BatchedQueueAccount<'a>, BatchedMerkleTreeError> {
    Self::from_account_info::<OUTPUT_QUEUE_TYPE>(&ACCOUNT_COMPRESSION_PROGRAM_ID, account_info)
}

fn from_account_info<const QUEUE_TYPE: u64>(
    program_id: &solana_program::pubkey::Pubkey,
    account_info: &AccountInfo<'a>,
) -> Result<BatchedQueueAccount<'a>, BatchedMerkleTreeError> {
    [...]
    let account_data: &'a mut [u8] = unsafe {
        std::slice::from_raw_parts_mut(account_data.as_mut_ptr(), account_data.len())
    };
    Self::from_bytes::<OUTPUT_QUEUE_TYPE>(account_data, (*account_info.key).into())
}
```

This does not result in an immediate issue since it is only called in `output_from_account_info`, which passes `OUTPUT_QUEUE_TYPE`, the `QUEUE_TYPE` generic argument is meant to provide flexibility to `from_account_info`, allowing it to handle different queue types. If `from_account_info` were to be called with a queue type other than `OUTPUT_QUEUE_TYPE` in the future, the deserialization logic will still expect an output queue type.

Remediation

Utilize the `QUEUE_TYPE` generic instead of `OUTPUT_QUEUE_TYPE`.

Patch

Resolved in b45e0d1.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-LZC-SUG-00	Suggestions regarding inconsistencies in the codebase and ensuring adherence to coding best practices.

Code Maturity OS-LZC-SUG-00

Description

1. Validate metadata in `from_bytes_at` in both `ZeroCopyVec` and `ZeroCopyCyclicVec` to ensure that the metadata provided in the function is consistent with the actual data and constraints of the underlying container, such as verifying the length of the vector does not exceed its capacity.

   >_ *program-libs/zero-copy/src/cyclic_vec.rs* RUST

   ```rust
   #[inline]
   pub fn from_bytes_at(bytes: &'a mut [u8]) -> Result<(Self, &'a mut [u8]), ZeroCopyError> {
       let metadata_size = Self::metadata_size();
       if bytes.len() < metadata_size {
           return Err(ZeroCopyError::InsufficientMemoryAllocated(
               bytes.len(),
               metadata_size,
           ));
       }
       let (meta_data, bytes) = bytes.split_at_mut(metadata_size);
       let (metadata, _padding) = Ref::<&mut [u8], [L; 3]>::from_prefix(meta_data)?;
       let usize_len: usize = u64::from(metadata[CAPACITY_INDEX]) as usize;
       let full_vector_size = Self::data_size(metadata[CAPACITY_INDEX]);
       [...]
   }
   ```

2. Replace the hardcoded value 40 with the constant `DEFAULT_BATCH_ADDRESS_TREE_HEIGHT` in `initialize_address_trees::validate_batched_address_tree_params` to improve maintainability and readability. Similarly, in `queue::get_output_queue_account_size_default` and `BatchedMerkleTreeMetadata::default`, the `TEST_DEFAULT_ZKP_BATCH_SIZE` constant should be utilized instead of the hardcoded value 10.
3. Annotate functions that are utilized only in tests with `#[test_only]` in `batch-merkle-tree` and `zero-copy` (especially memory-intensive functions such as `to_vec` and `try_into_array`), to improve code clarity by clearly marking which functions are intended for testing purposes only. This also prevents the utilization of these test-specific functions in production code, ensuring better separation between test logic and actual application code.

Remediation

Implement the above-mentioned suggestions.

Patch

1. Issue #1 resolved in dbbf1fd.
2. Issue #2 resolved in b9eebe4 and a604e5a.
3. Issue #3 resolved in 394e562 and a604e5a.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL	Vulnerabilities that immediately result in a loss of user funds with minimal preconditions. Examples: • Misconfigured authority or access control validation. • Improperly designed economic incentives leading to loss of funds.
HIGH	Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit. Examples: • Loss of funds requiring specific victim interactions. • Exploitation involving high capital requirement with respect to payout.
MEDIUM	Vulnerabilities that may result in denial of service scenarios or degraded usability. Examples: • Computational limit exhaustion through malicious input. • Forced exceptions in the normal user flow.
LOW	Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions.
INFO	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants. • Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.