Light Protocol Refactor

Security Assessment

June 23rd, 2025 — Prepared by OtterSec

Nicola Vella nick0ve@osec.io

Filippo Barsanti barsa@osec.io

Tuyết Dương tuyet@osec.io

Robert Chen r@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

Light Protocol engaged OtterSec to assess code refactors. This assessment was conducted between May 22nd and June 6th, 2025. A follow-up review was performed on `batched-merkle-tree` program, between July 7th and July 24th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 6 findings throughout this audit engagement.

In particular, we identified a vulnerability where the tree capacity is never set or enforced, allowing the queue to exceed the Merkle tree's actual capacity (OS-LPR-ADV-00). Additionally, if root history capacity is too small, required Merkle roots may be evicted prematurely, resulting in proof verification failures (OS-LPR-ADV-01). Furthermore, read-only accounts may be incorrectly verified using stale state if they are also consumed as outputs in the same transaction, due to post-verification zeroing of their hashes (OS-LPR-ADV-02).

We also made recommendations for updating the codebase to improve overall clarity and functionality and mitigate potential security issues (OS-LPR-SUG-02). Additionally, we highlighted the fragility of hardcoding PDA bump values and advised adding a test to verify the bump for correctness and to prevent future mismatches (OS-LPR-SUG-02). Lastly, we suggested checking the next index before incrementing it, as the current capacity check only ensures space for one leaf, but batched updates insert multiple leaves (OS-LPR-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 commit 368f9f0. A follow-up review was performed against 6eaaeb5.

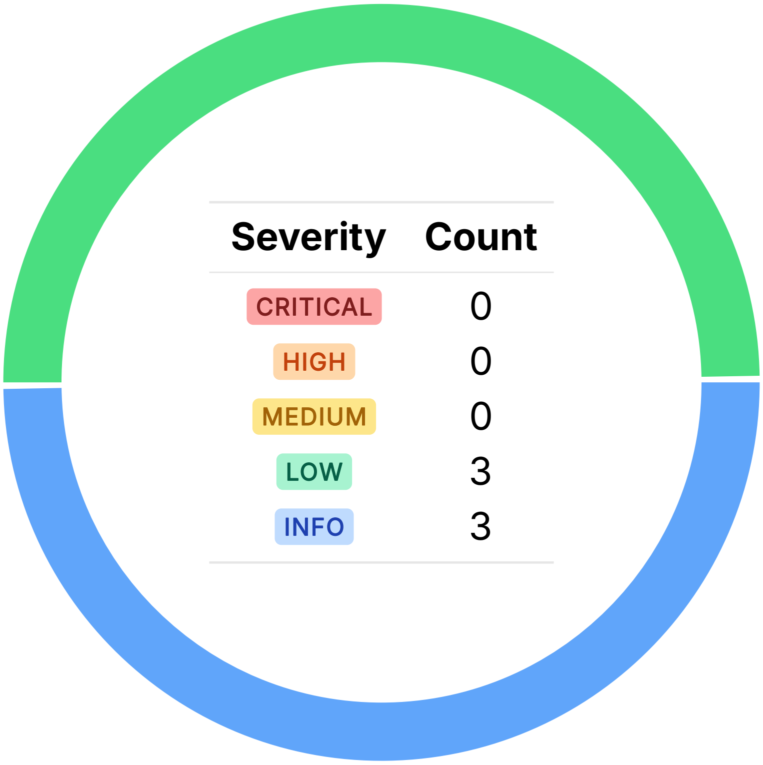
Brief descriptions of the programs is as follows:

Name	Description
refactors	Code refactors to utilize `account_info_init` instead of manually assigning the discriminator.
batched-merkle-tree	It efficiently queues and appends data entries into a Merkle tree in batches, optimizing for zero-knowledge proof generation and verification.

03 — Findings

Overall, we reported 6 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	0
LOW	3
INFO	3

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-LPR-ADV-00	LOW	RESOLVED ✓	The `tree_capacity` is never set or enforced, allowing the queue to exceed the Merkle tree's actual capacity.
OS-LPR-ADV-01	LOW	RESOLVED ✓	If `root_history_capacity` is too small, required Merkle roots may be evicted prematurely, resulting in proof verification failures.
OS-LPR-ADV-02	LOW	RESOLVED ✓	Read-only accounts may be incorrectly verified using stale state if they are also consumed as outputs in the same transaction, due to post-verification zeroing of their hashes.

Missing Tree Capacity Enforcement LOW OS-LPR-ADV-00

Description

The `queue::BatchedQueueMetadata` structure defines a `tree_capacity` field meant to cap the number of leaves that may be inserted into the Merkle tree (2^{height}), but this field is never initialized in `init`, and `check_tree_is_full`, which checks if the tree is full via the `tree_capacity`, is never utilized as a result. Thus, it is possible to insert as many as `2*output_queue_batch_size` elements into the queue, since the queue supports two batches. If this exceeds the Merkle tree's actual `tree_capacity`, leaves may be inserted into the queue but will fail to append to the Merkle tree. There are also no checks `output_queue_batch_size`.

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

```rust
/// Check if the tree is full.
pub fn check_tree_is_full(&self) -> Result<(), BatchedMerkleTreeError> {
    if self.tree_is_full() {
        return Err(BatchedMerkleTreeError::TreeIsFull);
    }
    Ok(())
}
```

Remediation

Set `tree_capacity` explicitly in `BatchedQueueMetadata::init`

Patch

Resolved in 50778df.

Failure to Enforce Minimum Root History Capacity LOW OS-LPR-ADV-01

Description

The `root_history` stores recent Merkle roots and is utilized by both input and output queues during zk-proof batch updates. Since each queue appends multiple roots per batch, the minimum required `root_history_capacity` is the sum of `input_batch_size / input_zkp_batch_size` and `output_batch_size / output_zkp_batch_size`. However, this is not currently verified. Thus, if its capacity is too small, older roots may be overwritten before all associated zk-proofs are finalized, breaking zk-proof verification.

Remediation

Ensure that `root_history_capacity` is set to at least the minimum required value, as outlined above.

Patch

Resolved in 50778df.

Stale Read-Only Account Verification LOW OS-LPR-ADV-02

Description

The vulnerability stems from a race condition between verifying read-only compressed accounts and consuming them as outputs in the same transaction. The protocol allows verifying inclusion of read-only compressed accounts in an output queue via the `verify_read_only_account_inclusion_by_index` . However, these accounts may also be part of the output queue being spent within the same transaction. When spent, their account hashes are zeroed out via a CPI call to the account-compression program, indicating they have been consumed.

The read-only verification (`verify_read_only_account_inclusion_by_index`) and `verify_proof` checks are executed before the CPI that spends (zeroes out) the output queue accounts. Thus, the verification step reads the account's original hash. Consequently, if a read-only account is also included in an output queue spent in the same call, its stale state will be read instead.

Remediation

Ensure read-only accounts are not allowed to be in the output queue of the same transaction.

Patch

Resolved in 50778df.

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-LPR-SUG-00	The current capacity check only ensures space for one leaf, but batched updates insert multiple leaves.
OS-LPR-SUG-01	Recommendation for updating the codebase to improve overall clarity and functionality and mitigate potential security issues.
OS-LPR-SUG-02	Suggestion to add a test to verify the bump for correctness and to prevent future mismatches.

Incorrect Capacity Check

OS-LPR-SUG-00

Description

Currently `merkle_tree::update_tree_from_output_queue_account` only checks if the Merkle tree has at least one slot left (`next_index < capacity`) utilizing `check_tree_is_full`. However, batched updates may insert multiple leaves at once (`batch_size`), and the `next_index` will be incremented by the size of the batch, which implies the tree will overflow if `next_index + batch_size > capacity`.

Remediation

Ensure to check the `next_index` before incrementing it.

Patch

Resolved in 50778df.

Code Refactoring OS-LPR-SUG-01

Description

1. `check_fee_payer` in `account_checks` only checks if the account is a signer, but not if it is writable. Since fee payment requires lamport deduction, the account must also be writable to avoid runtime errors.

   >_ *programs/system/src/accounts/account_checks.rs* RUST

   ```rust
   pub fn check_fee_payer(fee_payer: Option<&AccountInfo>) -> Result<&AccountInfo> {
       let fee_payer = fee_payer.ok_or(ProgramError::NotEnoughAccountKeys)?;
       check_signer(fee_payer).map_err(ProgramError::from)?;
       Ok(fee_payer)
   }
   ```

2. `initialize_address_tree::validate_batched_address_tree_params` includes an assertion message that lists unsupported ZKP batch sizes, such as 1, as valid options. However, only sizes such as 10, 100, 250, 500, 1000 are actually accepted. This inconsistency may mislead developers during testing or debugging.
3. The `sol_pool_pda` and `decompression_recipient` accounts must be writable when utilized, but this is not currently enforced in `from_account_infos`, risking runtime errors during `SOL` transfers.

Remediation

1. Add a check to ensure the account is writable.
2. Update the message to reflect the correct supported values to improve clarity.
3. Ensure that `InvokeCpiInstruction::from_account_infos` and `InvokeInstruction::from_account_infos` check that `sol_pool_pda` and `decompression_recipient` accounts are writable.

Patch

1. Resolved in e9c6594.
2. Resolved in 50778df.
3. Resolved in 50778df.

Hardcoded PDA Bump Validation OS-LPR-SUG-02

Description

`check_pda_seeds_with_bump` relies on hardcoded bumps, which may be fragile. While it is currently called solely by `check_anchor_option_sol_pool_pda`, which utilizes a hardcoded bump of 255, ensuring canonical behavior in this case, this approach may result in issues if re-utilized elsewhere with different bump values. It will be appropriate to add a test to catch potential mistakes if this function is re-utilized elsewhere in the future.

Remediation

Incorporate the above suggestion.

Patch

Resolved in 1c90f3e.

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.