Light Protocol CPI Context Refactor

Security Assessment

October 28th, 2025 — Prepared by OtterSec

Nicola Vella nick0ve@osec.io

Filippo Barsanti barsa@osec.io

Robert Chen r@osec.io

Table of Contents

01 — Executive Summary

Overview

Light Protocol engaged OtterSec to assess cpi-context refactors. This assessment was conducted between October 8th and October 27th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 2 findings throughout this audit engagement.

In particular, we identified a vulnerability in the function that reinitializes the CPI context, where the current logic fails to reallocate the account before reinitializing. Consequently, older accounts may lack enough space for the new layout, resulting in failures (OS-CCR-SUG-00).

We also made recommendations for updating the codebase to improve overall clarity and functionality and mitigate potential security issues (OS-CCR-SUG-01).

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 a3497b1.

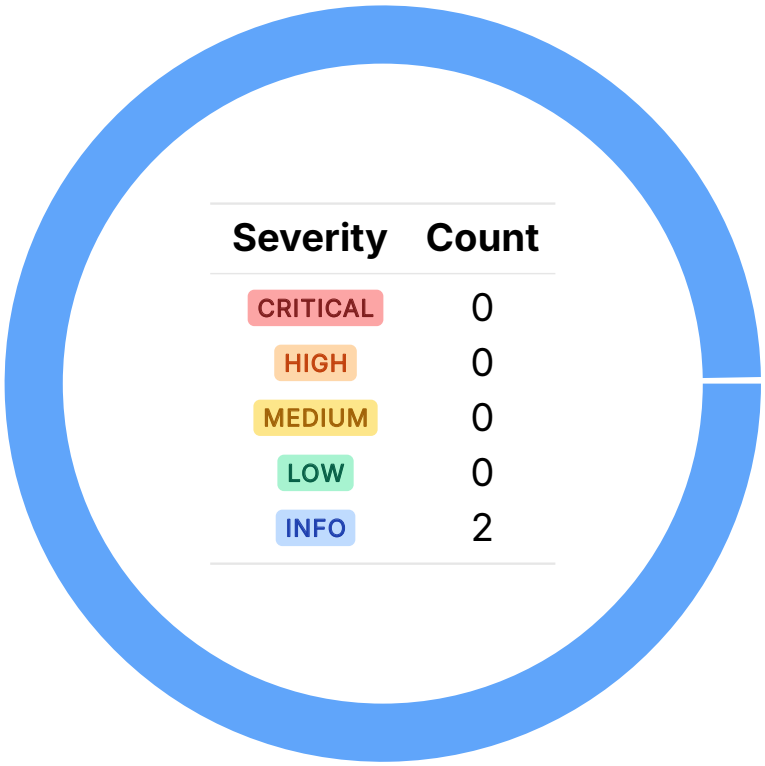
Brief descriptions of the programs is as follows:

Name	Description
cpi-context refactors	A refactor of the cpi context account, system program account checks, and account compression program nullify create output order.

03 — Findings

Overall, we reported 2 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	0
INFO	2

04 — 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-CCR-SUG-00	While reinitializing the CPI context, the function does not account for the size change of the account before reinitializing.
OS-CCR-SUG-01	Recommendation for updating the codebase to improve overall clarity and functionality and mitigate potential issues.

Missing Account Reallocation Check OS-CCR-SUG-00

Description

`init_context_account::reinit_cpi_context_account` reinitializes an existing CPI context account but does not handle the account size change before reinitializing the account to the v2 layout. If the account layout changes, the existing account buffer may be too small/big to fit the new layout.

Remediation

Ensure to resize the account before reinitialization.

Patch

Resolved in 4da1aaa by resizing the account to the v2 size.

Code Refactoring OS-CCR-SUG-01

Description

1. `remaining` in `AccountIterator` returns all unprocessed accounts but does not advance the iterator's position, rendering the internal state unchanged. This behaviour may be error-prone/ambiguous, specifically in `process_cpi_context`, if `next_account` is called afterwards. Ensure `remaining` consumes the iterator by advancing its position after returning the remaining accounts.

>_ light-protocol/program-libs/account-checks/src/account_iterator.rs RUST

```rust
#[inline(always)]
#[track_caller]
pub fn remaining(&self) -> Result<&'info [T], AccountError> {
    if self.position >= self.accounts.len() {
        let location = Location::caller();
        let account_name = "remaining accounts";
        solana_msg::msg!(
            "ERROR: Not enough accounts. Requested '{}' at index {} but only {} accounts
                ↪  available. {}{}:{}",
            account_name, self.position, self.accounts.len(), location.file(),
                ↪  location.line(),location.column()
        );
        return Err(AccountError::NotEnoughAccountKeys);
    }
    Ok(&self.accounts[self.position..])
}
```

2. `copy_cpi_context_outputs` still contains a debug assertion (`assert_eq!(bytes.len(), 4)`) meant for debugging purposes. Ensure to remove the assertion before mainnet deployment.

>_ light-protocol/programs/system/src/cpi_context/process_cpi_context.rs RUST

```rust
#[profile]
pub fn copy_cpi_context_outputs(
    cpi_context_account: &Option<ZCpiContextAccount2<'_>>,
    bytes: &mut [u8],
) -> Result<()> {
    if let Some(cpi_context) = cpi_context_account {
        [...]
        // Debug assert TODO: remove pre mainnet deployment
        assert_eq!(bytes.len(), 4);
    }
    Ok(())
}
```

Remediation

Incorporate the above refactors.

Patch

Resolved in 669cfa9.

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.