Accretion

SECURITY ASSESSMENT

Provided by Accretion Labs Pte Ltd. for Light Protocol
April 04, 2025
A25LIG2

Light Protocol

AUDITORS

Role	Name
Lead Auditor	Robert Reith (robert@accretion.xyz)

CLIENT

Light Protocol (https://lightprotocol.com/) builds a ZK compression layer on top of the Solana blockchain. They engaged Accretion Labs to audit an update which includes a switch to batched merkle trees and efficiency improvements.

ENGAGEMENT SCOPE

Crates: light-zero-copy, light-batched-merkle-tree

Link: https://github.com/Lightprotocol/light-protocol

Commit: 8aa0a1abb176149a586af3698a25d55222ce2ca4, 159dd679b5881239edaae77ff5f581d282c26382

ProgramID: compr6CUsB5m2jS4Y3831ztGSTnDpnKJTKS95d64XVq

Programs: System, Compression, Registry

Link: https://github.com/Lightprotocol/light-protocol

Commit: b9eebe4c6184e170723bf6f063157e07d17378a6

ProgramID:
SySTEM1eSU2p4BGQfQpimFEWWSC1XDFeun3Nqzz3rT7,
compr6CUsB5m2jS4Y3831ztGSTnDpnKJTKS95d64XVq,
Lighton6oQpVkeewmo2mcPTQQp7kYHr4fWpAgJyEmDX

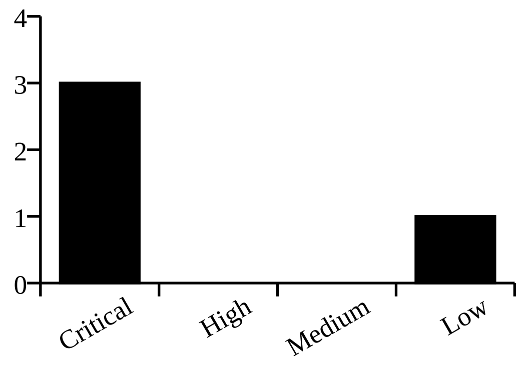
ENGAGEMENT TIMELINE

02 Feb	Project Kickoff Initial planning and scope definition
04 Feb	Assessment Begins Security review and testing phase
01 Mar	Review Fixes Security recommendations are given and implemented
04 Apr	Project Completion Report delivery and on-chain confirmation

ASSESSMENT

The Light Protocol upgrade demonstrated excellent code quality with strong engineering and thorough testing. However, we identified several similar critical vulnerabilities exclusively in the account compression program, a notable anomaly within an excellent codebase. These related issues were promptly fixed by the Light Protocol team.

SEVERITY DISTRIBUTION



AUDITED CODE

Program 1

ProgramID: SySTEM1eSU2p4BGQfQpimFEWWSC1XDFeun3Nqzz3rT7

Repository: https://github.com/Lightprotocol/light-protocol

Build Hash: c151d7e8acd5c32297869f6ffb94993c22abdcc3ea6dfa9be4ede111f6a71708

Commit: 4a59296c1b29c7117e6820c848fad0802025b156

Program 2

ProgramID: compr6CUsB5m2jS4Y3831ztGSTnDpnKJTKS95d64XVq

Repository: https://github.com/Lightprotocol/light-protocol

Build Hash: 59a235f76b7ebf573d2aa377218bc588ba838000647aa6b3a3e95b696ca10aa0

Commit: 4a59296c1b29c7117e6820c848fad0802025b156

Program 3

ProgramID: Lighton6oQpVkeewmo2mcPTQQp7kYHr4fWpAgJyEmDX

Repository: https://github.com/Lightprotocol/light-protocol

Build Hash: 2fc1ecee647467fae23be7842101d67eff59eda4869fab6ae50d5e5798bc419f

Commit: 4a59296c1b29c7117e6820c848fad0802025b156

ISSUES SUMMARY

ID	TITLE	**SEVERITY**	**STATUS**
ACC-C1	Migrating State doesn't check output queue belongs to authority	critical	fixed
ACC-C2	Missing Signer check when inserting into Address BatchedMerkleTreeAccount	critical	fixed
ACC-C3	Missing ownership check for RegisteredProgram accounts when inserting into queues	critical	fixed
ACC-L1	check_account_balance_is_rent_exempt asserts exact lamport amount	low	fixed

DETAILED ISSUES

ID	ACC-C1
Title	Migrating State doesn't check output queue belongs to authority
Severity	critical
Status	fixed

Description

When migrating leaves from a legacy `StateMerkleTreeAccount` using the `migrate_state` instruction, the code nullifies the leaf in the merkle tree, and then inserts it into the given batch output queue. In this process, the program correctly checks that the Signer is allowed to operate on the merkle tree, but there is no check that the output_queue belongs to the same authority. This means that we can create an arbitrary leaf in our own legacy merkle tree, and then insert it into any output queue belonging to other programs, for example allowing us to create fake deposits that we can then withdraw leading to Loss of Funds.

Location

https://github.com/Lightprotocol/light-protocol/blob/9d63d84e99505332575705e6da2d58b5cb0b7932/programs/account-compression/src/instructions/migrate_state.rs#L30-L32 https://github.com/Lightprotocol/light-protocol/blob/9d63d84e99505332575705e6da2d58b5cb0b7932/programs/account-compression/src/instructions/migrate_state.rs#L79-L88

Relevant Code

```rust
#[derive(Accounts)]
pub struct MigrateState<'info> {

    // [...]

    /// CHECK: with from_account_info.
    #[account(mut)]
    pub output_queue: AccountInfo<'info>,
    // [...]
}

// [...]

    let output_queue =
        &mut BatchedQueueAccount::output_from_account_info(&ctx.accounts.output_queue)
            .map_err(ProgramError::from)?;
    // 2. Migrate state
    let nullify_event = migrate_state(
        migrate_leaf_params,
        &mut zero_copy_merkle_tree,
        &merkle_tree.key(),
        output_queue,
    )?;
```

Mitigation Suggestion

When migrating, check that the given authority is also the authority for the output queue.

Remediation

Fixed in 42184a2b2132be94d36141c5710e506c3c381bb2.

ID	ACC-C2
Title	Missing Signer check when inserting into Address BatchedMerkleTreeAccount
Severity	critical
Status	fixed

Description

When calling the `insert_into_queues` instruction, the Signer of the Authority is not checked to correspond to the authority for an Address Tree, when the account in Question is a \`BatchedMerkleTreeAccount\` with \`TreeType::BatchedAddress\`. This means that anyone can insert arbitrary addresses into this tree.

Location

https://github.com/Lightprotocol/light-protocol/blob/b9eebe4c6184e170723bf6f063157e07d17378a6/programs/account-compression/src/context.rs#L134-L156

Relevant Code

```
BatchedMerkleTreeAccount::DISCRIMINATOR => {
                let mut tree_type = [0u8; 8];
                tree_type.copy_from_slice(&account_info.try_borrow_data()?[8..16]);
                let tree_type = TreeType::from(u64::from_le_bytes(tree_type));
                match tree_type {
                    TreeType::BatchedAddress => Ok(AcpAccount::BatchedAddressTree(
                        BatchedMerkleTreeAccount::address_from_account_info(account_info)
                            .map_err(ProgramError::from)?,
                    )),
                    TreeType::BatchedState => {
                        let tree = BatchedMerkleTreeAccount::state_from_account_info(account_info)
                            .map_err(ProgramError::from)?;
                        manual_check_signer_is_registered_or_authority::<BatchedMerkleTreeAccount>(
                            registered_program_pda,
                            authority,
                            &tree,
                        )?;

                        Ok(AcpAccount::BatchedStateTree(tree))
                    }
                    _ => Err(ProgramError::from(AccountError::BorrowAccountDataFailed).into()),
                }
            }
```

Mitigation Suggestion

Add the call to `manual_check_signer_is_registered_or_authority` to the \`TreeType::BatchedAddress\` case.

Remediation

Fixed in b9362d8adb46bbb3c3cba46e462969c3477e9363.

ID	ACC-C3
Title	Missing ownership check for RegisteredProgram accounts when inserting into queues
Severity	critical
Status	fixed

Description

The `insert_into_queues` instruction inserts nullifiers, leaves and addresses into queue accounts. For this, it offers a bool to be set when this instruction is called by a program. In this case, the first given account after the authority is interpreted as a `RegisteredProgram` account. A `RegisteredProgram` account is created by a Group authority, has to also be signed by the Program, and adds a Program to a Group of programs that belong together. From this first account `RegisteredProgram` account, a signer PDA is calculated that belongs to the registered program, and the group authority PDA is read as well. In this process, the owner of the `RegisteredProgram` account however is not checked. This means that those two values can be manipulated and a fake program registration be provided. Later on, all following accounts, such as Merkle Trees and Queues, are checked by confirming that that their owner is the provided `group_authority_pda`, and that the derived signer matches the given signer. In other words, the instruction confirms that we have the correct program signer, and allows this signer to write to any accounts that belong to the Group. Now, because we can fake a program registration, this means an attacker can create a completely new unregistered Program calling this instruction with the correct `PDA_AUTHORITY_PDA_SEED` signature, and a fake `RegisteredProgram` account, which contains itself as the registered program, and a victim Group's `group_authority_pda`. In the malicious call, the attacker Program can now insert arbitrary elements into all queues belonging to the victim Group. This includes nullifiers, leaves and addresses. In practice, this most likely means complete Loss of Funds for all program Groups registered with the protocol.

Location

https://github.com/Lightprotocol/light-protocol/blob/b9eebe4c6184e170723bf6f063157e07d17378a6/programs/account-compression/src/context.rs#L75-L93

Relevant Code

```rust
let derived_address = match invoked_by_program {
            true => {
                let account_info = &account_infos[0];
                let data = account_info.try_borrow_data()?;
                if RegisteredProgram::DISCRIMINATOR.as_slice() != &data[..8] {
                    return Err(AccountError::InvalidDiscriminator.into());
                }
                let account = bytemuck::from_bytes::<RegisteredProgram>(&data[8..]);
                // 1,670 CU
                // TODO: get from RegisteredProgram account and compare
                let derived_address = Pubkey::create_program_address(
                    &[CPI_AUTHORITY_PDA_SEED, &[bump]],
                    &account.registered_program_id,
                )?;
                skip += 1;
                Some((derived_address, account.group_authority_pda))
            }
            false => None,
        };
```

Mitigation Suggestion

Before checking the Discriminator of `RegisteredProgram`, check that its account ownership.

Remediation

Fixed in 07b3666f53d8035a1e5877381eae0fc31f48f6b1.

ID	ACC-L1
Title	check_account_balance_is_rent_exempt asserts exact lamport amount
Severity	low
Status	fixed

Description

The function `check_account_balance_is_rent_exempt` checks the account balance of an account, and if it matches the required rent amount for the expected account size. However, this function is extremely strict and asserts that the account's lamports match the the required rent exactly. This means that an attacker could supply this account with too many lamports, which would lead to the account being blocked from creation. This would require the account to be at a predictable address, for example a PDA.

Location

https://github.com/Lightprotocol/light-protocol/blob/159dd679b5881239edaae77ff5f581d282c26382/program-libs/utils/src/account.rs#L100-L133

Relevant Code

```
pub fn check_account_balance_is_rent_exempt(
    account_info: &AccountInfo,
    expected_size: usize,
) -> Result<u64, UtilsError> {
    let account_size = account_info.data_len();
    if account_size != expected_size {
        #[cfg(target_os = "solana")]
        solana_program::msg!(
            "Account {:?} size not equal to expected size. size: {}, expected size {}",
            account_info.key,
            account_size,
            expected_size
        );
        return Err(UtilsError::InvalidAccountSize);
    }
    let lamports = account_info.lamports();
    #[cfg(target_os = "solana")]
    {
        let rent_exemption = (Rent::get().map_err(|_| UtilsError::FailedBorrowRentSysvar))?
            .minimum_balance(expected_size);
        if lamports != rent_exemption {
            solana_program::msg!(
            "Account {:?} lamports is not equal to rentexemption: lamports {}, rent exemption {}",
            account_info.key,
            lamports,
            rent_exemption
        );
            return Err(UtilsError::InvalidAccountBalance);
        }
    }
    #[cfg(not(target_os = "solana"))]
    println!("Rent exemption check skipped since not target_os solana.");
    Ok(lamports)
}
```

Mitigation Suggestion

We suggest changing the fail condition `if lamports != rent_exemption` to `if lamports < rent_exemption`, so that the function succeeds when too many lamports exists in the account.

Remediation

Fixed in commit `4a59296c1b29c7117e6820c848fad0802025b156`

APPENDIX

Vulnerability Classification

We rate our issues according to the following scale. Informational issues are reported informally to the developers and are not included in this report.

Severity	Description
Critical	Vulnerabilities that can be easily exploited and result in loss of user funds, or directly violate the protocol's integrity. Immediate action is required.
High	Vulnerabilities that can lead to loss of user funds under non-trivial preconditions, loss of fees, or permanent denial of service that requires a program upgrade. These issues require attention and should be resolved in the short term.
Medium	Vulnerabilities that may be more difficult to exploit but could still lead to some compromise of the system's functionality. For example, partial denial of service attacks, or such attacks that do not require a program upgrade to resolve, but may require manual intervention. These issues should be addressed as part of the normal development cycle.
Low	Vulnerabilities that have a minimal impact on the system's operations and can be fixed over time. These issues may include inconsistencies in state, or require such high capital investments that they are not exploitable profitably.
Informational	Findings that do not pose an immediate risk but could affect the system's efficiency, maintainability, or best practices.

Audit Methodology

Accretion is a boutique security auditor specializing in Solana's ecosystem. We employ a customized approach for each client, strategically allocating our resources to maximize code review effectiveness. Our auditors dedicate substantial time to developing a comprehensive understanding of each program under review, examining design decisions, expected and edge-case behaviors, invariants, optimizations, and data structures, while meticulously verifying mathematical correctness—all within the context of the developers' intentions.

Our audit scope extends beyond on-chain components to include associated infrastructure, such as user interfaces and supporting systems. Every audit encompasses both a holistic protocol design review and detailed line-by-line code analysis.

During our assessment, we focus on identifying:
- Solana-specific vulnerabilities
- Access control issues
- Arithmetic errors and precision loss
- Race conditions and MEV opportunities
- Logic errors and edge cases
- Performance optimization opportunities
- Invariant violations
- Account confusion vulnerabilities
- Authority check omissions
- Token22 implementation risks and SPL-related pitfalls
- Deviations from best practices

Our approach transcends conventional vulnerability classifications. We continuously conduct ecosystem-wide security research to identify and mitigate emerging threat vectors, ensuring our audits remain at the forefront of Solana security practices.