# Security Audit - Light Protocol

conducted by Neodyme AG

| | |
|---|---|
| Lead Auditor: | Robert Reith |
| Second Auditor: | Mathias Scherer |
| Administrative Lead: | Thomas Lambertz |

August 28, 2024

# Table of Contents



## Appendices

# 1 | Executive Summary

**Neodyme** audited **Light Protocol's** on-chain ZK-Layer during July and August 2024.

The scope of this audit included implementation security, overall design and architecture. The auditors found that Light Protocol's staking program comprised a clean design and far-above-standard code quality. According to Neodymes Rating Classification, **1 high and 3 low- or informational-severity issues** were found. The number of findings identified throughout the audit, grouped by severity, can be seen in Figure 1.

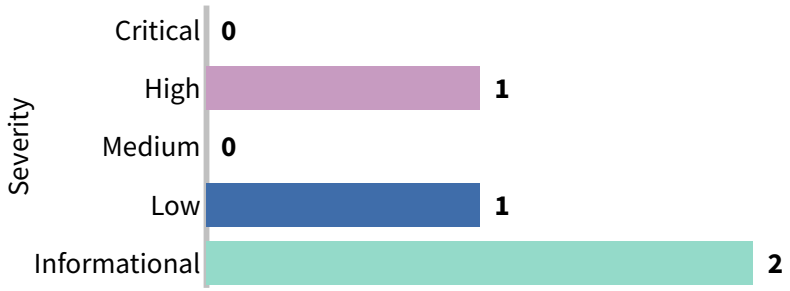


**Figure 1:** Overview of Findings

The auditors reported all findings to the Light Protocol developers, who addressed them promptly. The security fixes were verified for completeness by Neodyme. In addition to these findings, Neodyme delivered the Light Protocol team a list of nit-picks and additional notes that are not part of this report.

# 2 | Introduction

During the Summer of 2024, Light Protocol engaged Neodyme to do a detailed security analysis of their on-chain ZK-Layer programs. Two senior security researchers from Neodyme conducted independent full audits of the contract between the 1st of July and the 23rd of August 2024. Both auditors have a long track record of finding critical and other vulnerabilities in Solana programs.

The audit focused on the contract's technical security. In the following sections, we present our findings and discuss worst-case scenarios for authority compromise, and provide some general notes for considerations that may be useful in the future.

Neodyme would like to emphasize the high quality of Light Protocol's work. Light Protocol's team always responded quickly and **competently** to findings of any kind. Their **in-depth knowledge** of Solana programs was apparent during all stages of the cooperation, including excellent. Evidently, Light Protocol invested significant effort and resources into their product's security. Some technical debt was apparent in the contracts, but only to a minor extent. The contract's source code has no unnecessary dependencies, relying mainly on the well-established Anchor framework.

## Summary of Findings

During the audit, **2 security-relevant** and **2 informational** findings were identified. Light Protocol remediated all of those findings before the protocol's launch.

In total, the audit revealed:

**0** critical • **1** high-severity • **0** medium-severity • **1** low-severity • **2** informational

issues.

# 3 | Scope

The contract audit's scope comprised of two major components:

- **Implementation** security of the source code
- Security of the **overall design**

Neodyme considers the source code, located at https://github.com/Lightprotocol/light-protocol/, in scope for this audit. Third-party dependencies are not in scope. Light Protocol only relies on the Anchor library, the spl-token program, arksworks (a collection of libraries for zkSNARK), all of which are well-established. During the audit, minor changes and fixes were made by Light Protocol, which the auditors also reviewed in-depth.

Relevant source code revisions are:

- `6accb201910fb561b5ef15c66c9a3b7aebf24549` · Start of the audit
- `1c2ed8fa9c205a52443f9dcf3f422f1140d8e9d3` · Handover Compressed token program
- `2b4a6fadd7491f6a9f05479d0ebf09c96e4d8e8c` · Handover Registry program
- `a877a4f7345c48fd09f4b75fcaf4c4fdeac0a262` · Last reviewed revision

# 4 | Project Overview

This section briefly outlines Light Protocol's functionality, design, and architecture, followed by a detailed discussion of all related authorities.

## Functionality

Light Protocol's zk compression primitive aims to enable Solana developers to compress their on-chain state, reducing state costs while inheriting the security, performance, and composability of the Solana L1.

At its core, the protocol implements an unspent transaction output (UTXO) model, where transaction outputs are arbitrary data stored in Merkle trees. To "spend" these outputs, which means to use them as inputs in further transactions, users need to call compression programs and prove that their inputs are included in a state Merkle tree. Then, the compression programs can create new outputs using arbitrary logic.

The idea is that no data except Merkle tree roots needs to be stored in the Solana state itself. Instead, call data are used as a cheap resource to store data for indexing.

To implement this the developers created multiple programs and libraries which we describe now.

### Account Compression

The Account Compression Program is responsible for creating, authenticating access to, updating and rolling over Merkle trees and associated queues. Each account is immutable because the data is stored within the Merkle trees. This results in multiple accounts being invalidated and created with each transaction.

For a program to use the account compression program it needs to be registered by a group authority. This group authority can be created by anyone. After registration, the program can create the necessary Merkle trees for states and addresses.

The program uses sparse Merkle trees to efficiently update the trees used by programs. Sparse Merkle trees have a limited size so the developers implemented a roll-over functionality that allows programs to roll over to a new tree when needed. To prevent transactions from failing because of an outdated tree root, the program also stores a changelog of previous changes and updates the root hashes within transactions. This allows the system to be easier to use by users. Because transactions don't have to include the latest root, which can be difficult for programs that are highly used.

### Light System Program

This program verifies the input state (via Merkle tree inclusion ZKP), hashes input and output compressed accounts, and invokes the Account Compression Program to write into Merkle trees owned by

the Group Authority that it is registered to. It also supports the creation of compressed accounts with permanent unique addresses, and SOL compression, decompression, and compressed-SOL transfers. It forms the base layer of the Light Protocol.

### Light Compressed Token

The compressed token program allows any SPL-Token to be compressed and moved into the Light Protocol. It supports the basic token program functionalities such as transfer, mint, burn, freeze, and thaw but not the extensions introduced by Token 2022. Compresses tokens are stored in a token pool which is unique for each mint. Tokens can be compressed and decompressed with the transfer function. Because of how the Light System and Account Compression program were built it allows for multiple accounts to transfer tokens of the same mint within the same instruction. Each instruction invalidates the input accounts and creates new accounts with the updated balances and parameters.

### Light Registry

The light registry program implements functionality to govern the protocol and its state, allowing for protocol configuration updates and the addition of new protocol features by registering new system programs. Additionally, the program regulates the contention of foresters by implementing logic to register foresters with weights and providing wrapper functions to regulate access to the account compression program. This includes emptying queues and rolling over trees. Registered foresters can empty queues of address and state Merkle trees by registering for epochs. Each epoch starts with a registration phase. During the active phase, foresters can perform their duties and report their performance during the report work phase. Rewards can be claimed post-epoch, although this feature is not yet implemented.

## On-Chain Data and Accounts

### Account Compression

The Account Compression program is responsible for the Merkle trees and queues used by the protocol.

- `AddressMerkleTreeAccount` stores the data and metadata such as owner, height, changelog, associated queue, etc that are necessary for the Merkle tree, which stores the Solana account data, to operate.
- `StateMerkleTreeAccount` stores the data and metadata such as owner, height, changelog, associated queue, etc that are necessary for the Merkle tree, which stores the PDA data, to operate.
- `RegisteredProgram` stores the registered program ID and the associated group authority. It is seeded from the public key of the program.
- `GroupAuthority` stores the authority that is allowed to sign for that group. The seeds used for it are `"group_authority" + user_defined_bytes`.

- `QueueAccount` stores the type of the queue, necessary metadata, and to which Merkle tree it is associated.

### Light System Program

The Light System Program has only 3 accounts:

- `CpiContextAccount` stores instruction data without executing the compressed transaction.
- CPIAuthority is derived from the seed `"cpi_authority"` and is used to sign the instruction to the Account Compression program.
- The SOL Pool PDA is seeded from `"sol_pool_pda"` and stores all the compressed SOL.

### Light Compressed Token

This program uses two accounts to function.

- CPIAuthority is derived from the seed `"cpi_authority"` and is used to sign the instruction to the Account Compression program.
- The token pool stores all compressed SPL tokens and is derived from `"pool" + mint.key().as_ref()`. It is a normal Token Account

### Light Registry

The registry has various accounts to store the necessary data for the foresters (crankers).

- CPIAuthority is derived from the seed `"cpi_authority"` and is used to sign the instruction to the Account Compression program.
- `ProtocolConfigPda` holds the protocol configurations and the authority of this program. It is seeded from `"authority"`.
- `ForesterPda` uses the seed `"forester" + forester_authority.as_ref()` and registers an authority that is allowed to act as a forester in the protocol.
- `ForesterEpochPda` is seeded from `"forester_epoch" + forester_pda.key().to_bytes().as_slice() + current_epoch.to_le_bytes().as_slice()` and registers that forester for the upcoming epoch. It stores the protocol configuration and forester weight for this epoch.
- `EpochPda` gets created each epoch and is seeded from `current_epoch.to_le_bytes().as_slice()`. It holds the registered forester weights, forester's work reports, and the protocol configuration used for this epoch.

# Instructions

The contracts have the following instructions, which we briefly summarized.

### Account Compression

| Instruction | Access Control | Summary |
|---|---|---|
| initializeAddressMerkleTreeAndQueue | ProgramAuthority | Initializes the address Merkle tree and queue for the program. |
| insertAddresses | ProgramAuthority | Inserts addresses into the address Merkle tree queue. |
| updateAddressMerkleTree | ProgramAuthority | Applies pending address updates from the queue to the Merkle tree. |
| rolloverAddressMerkleTreeAndQueue | ProgramAuthority | Rolls over the address Merkle tree and queue. |
| initializeGroupAuthority | Unrestricted | Creates a new GroupAuthority account. |
| updateGroupAuthority | GroupAuthority | Updates the given GroupAuthority. |
| registerProgramToGroup | GroupAuthority | Registers a program with the given GroupAuthority. |
| deregisterProgram | GroupAuthority | Deregisters a program from the given GroupAuthority. |
| initializeStateMerkleTreeAndNullifierQueue | ProgramAuthority | Initializes the state Merkle tree and queue for the program. |
| appendLeavesToMerkleTree | ProgramAuthority | Applies state updates to the state Merkle tree. |
| nullifyLeaves | ProgramAuthority | Applies pending nullifications in the queue to the state Merkle tree. |
| insertIntoNullfierQueues | ProgramAuthority | Inserts nullifications into the state Merkle tree queue. |
| rolloverStateMerkleTreeAndNullifierQueue | ProgramAuthority | Rolls over the state Merkle tree and queue. |

### Light System Program

| Instruction | Access Control | Summary |
|---|---|---|
| initCpiContextAccount | Unrestricted | Initializes a new CPI context account for the given Merkle tree. |

| Instruction | Access Control | Summary |
|---|---|---|
| invoke | AccountOwner | Verifies that the authority is the owner of the given accounts, de-/compresses lamports, and updates the program state. |
| invokeCpi | ProgramAuthority | Verifies that the authority is the owner of the given accounts, de-/compresses lamports, and updates the program state. |

### Light Compressed Token

The Light Protocol's architecture, allows most of these instructions to do batch processing.

| Instruction | Access Control | Summary |
|---|---|---|
| createTokenPool | Unrestricted | Creates the token pool. |
| mintTo | MintAuthority | Mints new tokens to the token pool and compresses them into the target account. |
| transfer | AccountOwner | Transfers, de-/compresses tokens. |
| approve | AccountOwner | Delegates tokens to a delegate. Delegated tokens are stored in a separate account. |
| revoke | AccountOwner | Revokes delegations and merges all accounts into one. |
| freeze | FreezeAuthority | Freezes the given accounts. |
| thaw | FreezeAuthority | Unfreezes or thaws the given accounts. |
| burn | AccountOwner | Burns tokens from the given accounts and the token pool. |

### Light Registry

| Instruction | Access Control | Summary |
|---|---|---|
| initializeProtocolConfig | Deployer | Initializes the protocol configuration PDA. |
| updateProtocolConfig | ProtocolAuthority | Updates the protocol configurations. |
| registerSystemProgram | ProtocolAuthority | Registers a new program in the Account Compression program. |
| deregisterSystemProgram | ProtocolAuthority | Deregisters a program from the Account Compression program. |

| Instruction | Access Control | Summary |
|---|---|---|
| registerForester | ProtocolAuthority | Registers a new forester in the program. |
| updateForesterPDA | ForesterAuthority | Updates the forester authority and configuration. |
| updateForesterPDAWeight | ProtocolAuthority | Updates the forester's assigned weight in the protocol. |
| registerForesterEpoch | ForesterAuthority | Registers the forester for the next epoch. Only allowed during the registration phase. |
| finalizeRegistration | ForesterAuthority | Finalizes the forester registration. Only allowed during the active phase of the epoch. |
| reportWork | ForesterAuthority | Reports the total work done during the previous epoch. Only allowed during the report-work phase of the epoch. |
| initializeAddressMerkleTree | Unrestricted | Creates a new address Merkle tree and queue in the Account Compression program. |
| initializeStateMerkleTree | Unrestricted | Creates a new state Merkle tree, nullifier queue, and CPI context in the Account Compression program. |
| nullify | Forester | Nullifies leaves for the given program in the state Merkle tree. |
| updateAddressMerkleTree | Forester | Processes address updates for the given program in the address Merkle tree. |
| rolloverAddressMerkleTreeAndQueue | Forester | Rolls over the address Merkle tree and queue with the Account Compression program. |
| rolloverStateMerkleTreeAndQueue | Forester | Rolls over the state Merkle tree and nullifier queue in the Account Compression program. It also updates the CPI context account with the new addresses. |

# Authority Structure and Off-Chain Components

## Light Compressed Token

The Light Compressed Token program has no authorities of its own but makes have use of the SPL-Token programs authorities defined in the mint. The developers made sure that privileges can not be escalated and use the authorities, defined in the mints, as they are intended to.

## Light Registry

The Light Registry is the only program in the protocol that has a global configuration. Therefore it has a `ProtocolAuthority` which has a small risk profile, due to the configurations possible. The `ProtocolAuthority` can update how epochs are configured (the length of phases), register new foresters, and update the weights assigned to foresters. A malicious actor holding that authority could bring the registry to a halt but this doesn't impact the functionality of the protocol besides not having foresters that work as crankers. At the time of writing foresters don't receive a reward for their duties, the developers intend to introduce rewards in the future, which could increase the risk of the `ProtocolAuthority`.

## Upgrade Authority

As with any contract, the upgrade authority has complete control over the program and the funds it controls. It is hence one of the most critical components of the security of the protocol. By maliciously upgrading the contract, the upgrade authority can irreversibly transfer control of all funds in the reserve, in the liquidity pool, and in all stake accounts to itself or other parties.

Light Protocol is aware of this and uses a multisig with cold wallets to secure the upgrade authorities used for their protocol.

# 5 | Findings

This section outlines all of our findings. They are classified into one of five severity levels, detailed in Appendix C. In addition to these findings, Neodyme delivered the Light Protocol team a list of nit-picks and additional notes that are not part of this report.

All findings are listed in Table 5 and further described in the following sections.

| Identifier | Name | Severity | Status |
|---|---|---|---|
| `ND-LIG01-HI-01` | Compression Program Allows Deregistering Arbitrary Programs | HIGH | Fixed |
| `ND-LIG01-LO-01` | Insufficient Noop Program Check | LOW | Fixed |
| `ND-LIG01-IN-01` | Users may create Empty Token Accounts for Invalid Mints | INFORMATIONAL | Fixed |
| `ND-LIG01-IN-02` | Active phase length increase leads to DOS | INFORMATIONAL | Fixed |

**Table 5:** Findings

## [ND-LIG01-HI-01] Compression Program Allows Deregistering Arbitrary Programs

| Severity | Impact | Affected Component | Status |
|---|---|---|---|
| HIGH | DOS | Account Compression | Fixed |

### Description

We found that the Account Compression program has insufficient checks for program deregistration. The `DeregisterProgram` Accounts structure ensures that the given signer belongs to the given group authority. However, there is no check that the given group authority is the authority of the registered program. This enables an attacker to create a new group authority and deregister all registered programs, temporarily creating a DoS requiring a program upgrade and stealing rent for the registered program accounts.

### Relevant Code

deregister_program.rs, lines 5-18

```
#[derive(Accounts)]
pub struct DeregisterProgram<'info> {
    /// CHECK: Signer is checked according to authority pda in instruction.
    #[account(mut, constraint= authority.key() == group_authority_pda.authority @AccountCompressionErrorCode::InvalidAuthority)]
    pub authority: Signer<'info>,
    #[account(
        mut, close=close_recipient
    )]
    pub registered_program_pda: Account<'info, RegisteredProgram>,
    pub group_authority_pda: Account<'info, GroupAuthority>,
    /// CHECK: recipient is not checked.
    #[account(mut)]
    pub close_recipient: AccountInfo<'info>,
}
```

### Mitigation Suggestion

We recommend adding another constraint that verifies that the `group_authority_pda` matches the group authority saved in the `registered_program_pda`.

### Resolution

The team implemented the additional constraint with commit `a0c2edb14bd42edad5e07cc8e47e12cd457b01be`.

## [ND-LIG01-LO-01] Insufficient Noop Program Check

| Severity | Impact | Affected Component | Status |
|---|---|---|---|
| **LOW** | Event confusion | Light System Program | Fixed |

### Description

To emit events, the system program calls uses the noop program pattern, where event data are stored in the call data of a CPI. To make sure that the called program is the expected one, a check is implemented that throws an error when the given noop program key is not the expected hardcoded noop program AND the given program is not executable. This means that an executable program that is not the expected noop program should pass the check. The CPI is done without passing additional signatures, so the invocation is not critical. However, an attacker could call their own program and potentially emit a fake, or incorrect event confusing any off-chain software that listens for events.

### Relevant Code

emit_event.rs, lines 39-43

```
if ctx.accounts.get_noop_program().key() != Pubkey::new_from_array(NOOP_PUBKEY)
        && !ctx.accounts.get_noop_program().executable
    {
        return err!(SystemProgramError::InvalidNoopPubkey);
    }
```

### Mitigation Suggestion

The intended check was probably to use an OR instead of an AND operator.

### Resolution

`a91cc3d35dae56145d98e8d3bf94784d346cef95` fixes the issue.

## [ND-LIG01-IN-01] Users may create Empty Token Accounts for Invalid Mints

| Severity | Impact | Affected Component | Status |
|---|---|---|---|
| INFORMATIONAL | Inconsistency between accounts | Compressed Token | Fixed |

### Description

We found that the Token Program allows users to create empty token accounts for invalid mints through the revoke mechanism. To do this, they would create a revoke instruction with an empty `InputTokenDataWithContext` Vector, and an arbitrary Mint, which is not checked, because there are no inputs to be verified. The revoke instruction always creates one output account for the given authority, and assigns it the sum of Lamports and amounts of all given input accounts, in this case both being zero. While the impact of this issue is insignificant as the accounts have zero-amounts, we believe that it may be inconsistent that token accounts could be created for invalid Mints this way.

### Relevant Code

delegation.rs, lines 161-242

```
#[derive(Debug, Clone, AnchorSerialize, AnchorDeserialize)]
pub struct CompressedTokenInstructionDataRevoke {
    pub proof: CompressedProof,
    pub mint: Pubkey,
    pub input_token_data_with_context: Vec<InputTokenDataWithContext>,
    pub cpi_context: Option<CompressedCpiContext>,
    pub output_account_merkle_tree_index: u8,
}

pub fn process_revoke<'a, 'b, 'c, 'info: 'b + 'c>(
    ctx: Context<'a, 'b, 'c, 'info, GenericInstruction<'info>>,
    inputs: Vec<u8>,
) -> Result<()> {
    let inputs: CompressedTokenInstructionDataRevoke =
                        CompressedTokenInstructionDataRevoke::deserialize(&mut
inputs.as_slice())?;
    let (compressed_input_accounts, output_compressed_accounts) =
        create_input_and_output_accounts_revoke(
            &inputs,
            &ctx.accounts.authority.key(),
            ctx.remaining_accounts,
        )?;
    cpi_execute_compressed_transaction_transfer(
        ctx.accounts,
        compressed_input_accounts,
        &output_compressed_accounts,
```

```
        Some(inputs.proof),
        inputs.cpi_context,
        ctx.accounts.cpi_authority_pda.to_account_info(),
        ctx.accounts.light_system_program.to_account_info(),
        ctx.accounts.self_program.to_account_info(),
        ctx.remaining_accounts,
    )?;
    Ok(())
}

pub fn create_input_and_output_accounts_revoke(
    inputs: &CompressedTokenInstructionDataRevoke,
    authority: &Pubkey,
    remaining_accounts: &[AccountInfo<'_>],
) -> Result<(
    Vec<PackedCompressedAccountWithMerkleContext>,
    Vec<OutputCompressedAccountWithPackedContext>,
)> {
    let (mut compressed_input_accounts, input_token_data, sum_lamports) =
           get_input_compressed_accounts_with_merkle_context_and_check_signer::
<NOT_FROZEN>(
            authority,
            &None,
            remaining_accounts,
            &inputs.input_token_data_with_context,
            &inputs.mint,
        )?;
    let sum_inputs = input_token_data.iter().map(|x| x.amount).sum::<u64>();
    let lamports = if sum_lamports != 0 {
        Some(vec![Some(sum_lamports)])
    } else {
        None
    };
    let mut output_compressed_accounts =
        vec![OutputCompressedAccountWithPackedContext::default(); 1];
    let hashed_mint = match hash_to_bn254_field_size_be(&inputs.mint.to_bytes())
{
        Some(hashed_mint) => hashed_mint.0,
        None => return err!(ErrorCode::HashToFieldError),
    };

    create_output_compressed_accounts(
        &mut output_compressed_accounts,
        inputs.mint,
        &[*authority; 1],
        None,
        None,
```

```
71          &[sum_inputs],
72          lamports,
73          &hashed_mint,
74          &[inputs.output_account_merkle_tree_index],
75      )?;
76      add_token_data_to_input_compressed_accounts::<NOT_FROZEN>(
77          &mut compressed_input_accounts,
78          input_token_data.as_slice(),
79          &hashed_mint,
80      )?;
81      Ok((compressed_input_accounts, output_compressed_accounts))
82  }
```

### Mitigation Suggestion

We recommend validating the supplied Mint account, or enforcing that at least one input account is given so that the Mint is verified implicitly by inclusion proofs.

### Resolution

The team enforced that at least one input account is given with commit `b9f686ce64f28a7cb2754aea4fc19e43cda53a29`. This resolves the issue.

## [ND-LIG01-IN-02] Active phase length increase leads to DOS

| Severity | Impact | Affected Component | Status |
|---|---|---|---|
| INFORMATIONAL | DOS | Light Registry | Fixed |

### Description

The system authority is allowed to update the `ProtocolConfig` of the Light Registry program. This includes the `active_phase_length` parameter which controls for how many slots an epoch is active. By increasing this value the system tries to repeat already existing epochs. Because the program only stores the protocol configuration in the `EpochPDA` and the `ForesterEpochPDA` during the initialization of these accounts, without a way to update the configuration at a later date, the epochs try to operate on expired configuration parameters. This results in the program seizing operation until the protocol configuration has caught up and creates new epoch accounts.

### Relevant Code

lib.rs, lines 160-198

```
/// Registers the forester for the epoch.
/// 1. Only the forester can register herself for the epoch.
/// 2. Protocol config is copied.
/// 3. Epoch account is created if needed.
pub fn register_forester_epoch<'info>(
    ctx: Context<'_, '_, '_, 'info, RegisterForesterEpoch<'info>>,
    epoch: u64,
) -> Result<()> {
    // Only init if not initialized
    if ctx.accounts.epoch_pda.registered_weight == 0 {
        (*ctx.accounts.epoch_pda).clone_from(&EpochPda {
            epoch,
            protocol_config: ctx.accounts.protocol_config.config,
            total_work: 0,
            registered_weight: 0,
        });
    }
                                    let         current_solana_slot         =
anchor_lang::solana_program::clock::Clock::get()?.slot;
    // Init epoch account if not initialized
    let current_epoch = ctx
        .accounts
        .epoch_pda
        .protocol_config
        .get_latest_register_epoch(current_solana_slot)?;

    if current_epoch != epoch {
```

```
            return err!(RegistryError::InvalidEpoch);
        }
        // check that epoch is in registration phase is in process_register_for_epoch
        process_register_for_epoch(
            &ctx.accounts.authority.key(),
            &mut ctx.accounts.forester_pda,
            &mut ctx.accounts.forester_epoch_pda,
            &mut ctx.accounts.epoch_pda,
            current_solana_slot,
        )?;
        Ok(())
    }
```

### Mitigation Suggestion

Either don't allow `active_phase_length` to be increased or allow `EpochPDA` and `ForesterEpochPDA` to update the configuration and reset its values if necessary.

### Resolution

The team implemented a fix with commit `a877a4f7345c48fd09f4b75fcaf4c4fdeac0a262`. This prevents changes to the `active_phase_length` parameter. In the future, they will implement a more robust fix.

# A | About Neodyme

Security is difficult.

To understand and break complex things, you need a certain type of people. People who thrive in complexity, who love to play around with code, and who don't stop exploring until they fully understand every aspect of it. That's us.

Our team never outsources audits. Having found over 80 High or Critical bugs in Solana's core code itself, we believe that Neodyme hosts the most qualified auditors for Solana programs. We've also found and disclosed critical vulnerabilities in many of Solana's top projects and have responsibly disclosed issues that could have resulted in the theft of over $10B in TVL on the Solana blockchain.

All of our team members have a background in competitive hacking. During such hacking competitions, called CTFs, we competed and collaborated while finding vulnerabilities, breaking encryption, reverse engineering complicated algorithms, and much more. Through the years, many of our team members have won national and international hacking competitions, and keep ranking highly among some of the hardest CTF events worldwide. In 2020, some of our members started experimenting with validators and became active members in the early Solana community. With the prospect of an interesting technical challenge and bug bounties, they quickly encouraged others from our CTF team to look for security issues in Solana. The result was so successful that after reporting several bugs, in 2021, the Solana Foundation contracted us for source code auditing. As a result, Neodyme was born.

# B | Methodology

We are not checklist auditors.

In fact, we pride ourselves on that. We adapt our approach to each audit, investing considerable time into understanding the program upfront and exploring its expected behavior, edge cases, invariants, and ways in which the latter could be violated.

We use our uniquely deep knowledge of Solana internals, and our years-long experience in auditing Solana programs to find bugs that others miss. We often extend our audit to cover off-chain components in order to see how users could be tricked or the contract affected by bugs in those components.

Nonetheless, we also have a list of common vulnerability classes, which we always exhaustively look for. We provide a sample of this list here.

## Select Common Vulnerabilities

Our most common findings are still specific to Solana itself. Among these are vulnerabilities such as the ones listed below:

- Insufficient validation, such as:
  - Missing ownership checks
  - Missing signer checks
  - Signed invocation of unverified programs
  - Account confusions
  - Missing freeze authority checks
  - Insufficient SPL account verification
  - Dangerous user-controlled bumps
  - Insufficient Anchor account linkage
- Account reinitialization vulnerabilities
- Account creation DoS
- Redeployment with cross-instance confusion
- Missing rent exemption assertion
- Casting truncation
- Arithmetic over- or underflows
- Numerical precision and rounding errors
- Anchor pitfalls, such as accounts not being reloaded
- Non-unique seeds
- Issues arising from CPI recursion
- Log truncation vulnerabilities
- Vulnerabilities specific to integration of Token Extensions, for example unexpected external token hook calls

Apart from such Solana-specific findings, some of the most common vulnerabilities relate to the general logical structure of the contract. Specifically, such findings may be:

- Errors in business logic
- Mismatches between contract logic and project specifications
- General denial-of-service attacks
- Sybil attacks
- Incorrect usage of on-chain randomness
- Contract-specific low-level vulnerabilities, such as incorrect account memory management
- Vulnerability to economic attacks
- Allowing front-running or sandwiching attacks

Miscellaneous other findings are also routinely checked for, among them:

- Unsafe design decisions that might lead to vulnerabilities being introduced in the future
    - Additionally, any findings related to code consistency and cleanliness
- Rug pull mechanisms or hidden backdoors

Often, we also examine the authority structure of a contract, investigating their security as well as the impact on contract operations should they be compromised.

Over the years, we have found hundreds of high and critical severity findings, many of which are highly nontrivial and do not fall into the strict categories above. This is why our approach has always gone way beyond simply checking for common vulnerabilities. We believe that the only way to truly secure a program is a deep and tailored exploration that covers all aspects of a program, from small low-level bugs to complex logical vulnerabilities.

# C | Vulnerability Severity Rating

We use the following guideline to classify the severity of vulnerabilities. Note that we assess each vulnerability on an individual basis and may deviate from these guidelines in cases where it is well-founded. In such cases, we always provide an explanation.

| Severity | Description |
|---|---|
| **CRITICAL** | Vulnerabilities that will likely cause loss of funds. An attacker can trigger them with little or no preparation, or they are expected to happen accidentally. Effects are difficult to undo after they are detected. |
| **HIGH** | Bugs that can be used to set up loss of funds in a more limited capacity, or to render the contract unusable. |
| **MEDIUM** | Bugs that do not cause direct loss of funds but that may lead to other exploitable mechanisms, or that could be exploited to render the contract partially unusable. |
| **LOW** | Bugs that do not have a significant immediate impact and could be fixed easily after detection. |
| **INFORMATIONAL** | Bugs or inconsistencies that have little to no security impact, but are still noteworthy. |

Additionally, we often provide the client with a list of nit-picks, i.e. findings whose severity lies below Informational. In general, these findings are not part of the report.

**Neodyme AG**

Dirnismaning 55
Halle 13
85748 Garching
Germany

E-Mail: contact@neodyme.io

https://neodyme.io