OtterSec

Light Protocol

Security Assessment

August 15th, 2024 — Prepared by OtterSec

Ajay Shankar Kunapareddy	d1r3wolf@osec.io
Tuyết Dương	tuyet@osec.io
Robert Chen	r@osec.io

Table of Contents

01 — Executive Summary

Overview

Light Protocol engaged OtterSec to assess the `light-protocol` and `groth16-solana` programs. This assessment was conducted between June 19th and July 30th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 30 findings throughout this audit engagement.

In particular, we identified several critical vulnerabilities, including a lack of seed validation in Program Derived Address in mint and transfer instructions, allowing an attacker to substitute their own token account (OS-LPL-ADV-00), and the utilization of incorrect authority for token transfers, enabling the transfer of SPL tokens from a specified token account to a token pool account (OS-LPL-ADV-01). We also identified multiple high-risk issues concerning an incorrect update in the batch append functionality, where the last iteration of the loop erroneously includes an extra node (OS-LPL-ADV-02), and the absence of a check to ensure that the new element is inserted at its designated index (OS-LPL-ADV-03). There is also an inconsistency in the prime number calculation process, which fails to correctly handle prime numbers of a specific form (OS-LPL-ADV-15), and the inadequate validation of non-executable programs (OS-LPL-ADV-07).

Additionally, it is possible to append duplicate compressed output account hashes to the Merkle tree (OS-LPL-ADV-19) and to utilize a program's private key to act as a signer in the invoke instruction, potentially enabling the program admin to misuse the authority to manipulate program-owned accounts or transfer lamports (OS-LPL-ADV-12). Furthermore, the dequeue functionality allows the removal of elements at index zero, which disrupts the expected behavior where index zero is reserved for a special zero value (OS-LPL-ADV-20).

We also made recommendations regarding modifications to the codebase for improved efficiency (OS-LPL-SUG-05) and suggested the need to ensure adherence to coding best practices (OS-LPL-SUG-06). Moreover, we advised the removal of unutilized and redundant code within the system for increased readability (OS-LPL-SUG-07) and the inclusion of additional safety checks within the codebase to make it more robust and secure (OS-LPL-SUG-03).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/Lightprotocol. This audit was performed against commits cbdfe13 and 5f1a152.

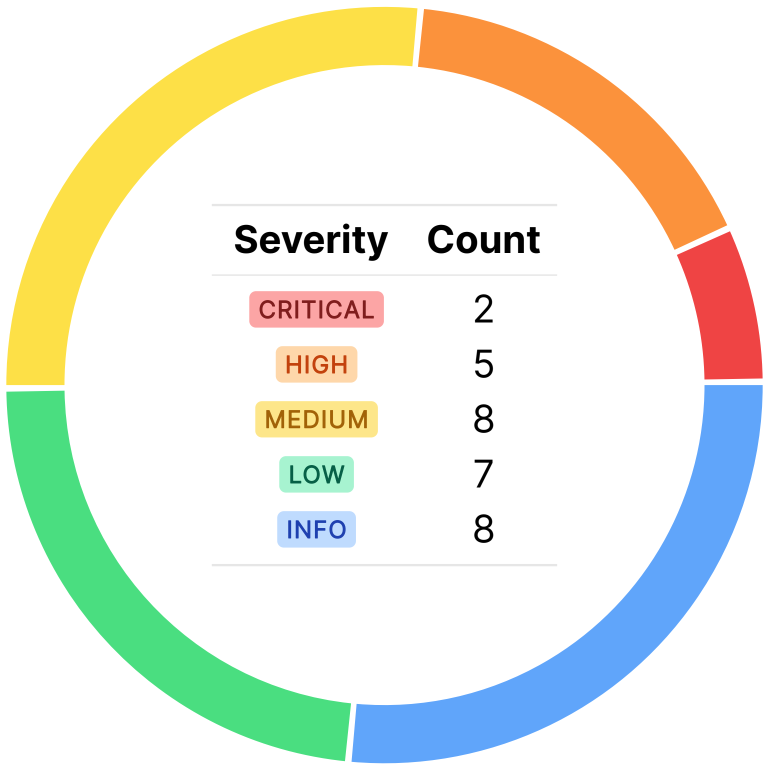
A brief description of the programs is as follows:

Name	Description
light-protocol	A zkLayer enabling stateless program execution, purpose-built for Solana.
groth16-solana	A zero-knowledge proof verification using Solana altbn254 syscalls.

03 — Findings

Overall, we reported 30 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	2
HIGH	5
MEDIUM	8
LOW	7
INFO	8

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-LPL-ADV-00	CRITICAL	RESOLVED ✓	`token_pool_pda` in `MintToInstruction` and `TransferInstruction` lack seed validation, allowing an attacker to substitute their own token account.
OS-LPL-ADV-01	CRITICAL	RESOLVED ✓	`compress_spl_tokens` incorrectly utilizes `cpi_authority_pda` as the authority for token transfers instead of `ctx.accounts.authority`.
OS-LPL-ADV-02	HIGH	RESOLVED ✓	There is an incorrect update in `append_batch` in `ConcurrentMerkleTree`, where the last iteration of the loop erroneously includes an extra node for `changelog` path updates.
OS-LPL-ADV-03	HIGH	RESOLVED ✓	There is no check in `IndexedMerkleTree::update` to ensure that the new element (`new_element`) is inserted at its designated index.
OS-LPL-ADV-04	HIGH	RESOLVED ✓	The rollover fee calculation incorrectly includes lamports as rent, resulting in the collection of excessive fees and potential unauthorized account closures by the authority owner.
OS-LPL-ADV-05	HIGH	RESOLVED ✓	`update` in `ConcurrentMerkleTree` fails to update canopy nodes when altering tree nodes, resulting in discrepancies in proof generation involving the canopy.

OS-LPL-ADV-06	HIGH	RESOLVED ✓	It is possible for duplicate Merkle tree accounts to be appended to the remaining accounts vector, resulting in identical leaf indices being utilized across different output accounts.
OS-LPL-ADV-07	MEDIUM	RESOLVED ✓	`emit_indexer_event` incorrectly allows non-xecutable programs to pass as valid `noop` programs, which should be flagged as `InvalidNoopPubkey`.
OS-LPL-ADV-08	MEDIUM	RESOLVED ✓	`process_cpi_context` lacks checks to verify that all input and output compressed accounts are part of the same Merkle tree, as specified by `CpiContextAccount.associated_merkle_tree`.
OS-LPL-ADV-09	MEDIUM	RESOLVED ✓	`InvokeCpiInstruction` lacks access control for `CpiContextAccount`, enabling an attacker to exploit the context by injecting malicious proofs if the context is set in multiple transactions.
OS-LPL-ADV-10	MEDIUM	RESOLVED ✓	`dequeue_at_with_low_element_index` misses to check if `elements[low_element_index].next_index` matches the index, which may result in dequeuing the wrong elements.
OS-LPL-ADV-11	MEDIUM	RESOLVED ✓	`CyclicBoundedVec::eq` compares the entire data slices directly, resulting in incorrect equality checks due to the unordered nature of elements affected by `first_index`.
OS-LPL-ADV-12	MEDIUM	RESOLVED ✓	The `invoke` instruction allows a program's private key to act as a `Signer`, potentially enabling the program admin to misuse their authority to manipulate program-owned accounts or transfer lamports.

OS-LPL-ADV-13	MEDIUM	RESOLVED ✓	`copy_from_bytes` in `ConcurrentMerkleTree` does not properly handle the `first_index` and `last_index` when copying `CyclicBoundedVec`, potentially resulting in a misordering of items.
OS-LPL-ADV-14	MEDIUM	RESOLVED ✓	`BoundedVec` and `CyclicBoundedVec` implementations do not deallocate memory upon `Drop`, which may result in double-free errors when utilizing `from_raw_parts` to create these vectors.
OS-LPL-ADV-15	LOW	RESOLVED ✓	`find_next_prime` fails to handle prime numbers of the form $6k + 1$ correctly when finding the next prime greater than a given number. Specifically, it skips over primes that are of this form.
OS-LPL-ADV-16	LOW	RESOLVED ✓	In `aligned_sized`, the `size_of` value for a field is recalculated, even if the field already has a `#[size]` attribute specifying its size.
OS-LPL-ADV-17	LOW	RESOLVED ✓	`ConcurrentMerkleTree40` is defined with a `HEIGHT` of 40, which is not supported by `ZeroBytes` as it has a height limit of 32.
OS-LPL-ADV-18	LOW	RESOLVED ✓	`append_batch` assigns `current_node` as the root node for both the `changelog` and `roots` array. This assignment may be incorrect if the computation of the Merkle path is interrupted before reaching the actual root.
OS-LPL-ADV-19	LOW	RESOLVED ✓	It is possible to append duplicate compressed output account hashes to the Merkle tree.

OS-LPL-ADV-20	LOW	RESOLVED ✓	`IndexedArray::dequeue_at` allows the removal of elements at index zero, which disrupts the expected behavior, as index zero is reserved for a special zero value.
OS-LPL-ADV-21	LOW	RESOLVED ✓	`HashSet::from_bytes_copy` copies the `next_value_index` from an incorrect offset of 24 (`bytes[24..32]`), from which `HashSetCell` elements begin.

Lack of PDA Validation CRITICAL OS-LPL-ADV-00

Description

In both `MintToInstruction` and `TransferInstruction`, the `token_pool_pda` account is included as a mutable account but does not undergo any seed validation or authority checks. This implies that there are no checks to ensure that `token_pool_pda` was generated utilizing the expected seeds and bump values. An attacker may provide their own token account as `token_pool_pda` instead of the intended token pool Program Derived Address (PDA). Since there are no checks to validate that this account is derived from the expected seeds, the program will treat the attacker's account as if it were the legitimate token pool PDA.

>_ *compressed-token/src/process_transfer.rs* rust

```rust
#[derive(Accounts)]
pub struct TransferInstruction<'info> {
    [...]
    #[account(mut)]
    pub token_pool_pda: Option<Account<'info, TokenAccount>>,
    #[account(mut)]
    pub compress_or_decompress_token_account: Option<Account<'info, TokenAccount>>,
    pub token_program: Option<Program<'info, Token>>,
    pub system_program: Program<'info, System>,
}
```

The attacker may utilize this manipulated `token_pool_pda` to compress tokens, which involves sending tokens to this account. Since the program does not validate the seeds, the tokens would be added to the attacker's account. Consequently, when decompressing tokens, the attacker will utilize the same `token_pool_pda` account to withdraw the tokens that were originally in the legitimate token pool. This would allow the attacker to steal tokens from the genuine token pool account and move them to their own account.

Remediation

Ensure that the `token_pool_pda` account is validated against the expected seeds and bump values to confirm that it is derived correctly.

Patch

Resolved in f59ffa1.

Utilization of Incorrect Authority for Token Transfers CRITICAL OS-LPL-ADV-01

Description

`spl_compression::compress_spl_tokens` is intended to transfer `SPL` tokens from a specified token account (`compress_or_decompress_token_account`) to a token pool account (`token_pool_pda`) to compress tokens. In the function, `transfer` is called with `cpi_authority_pda` as the authority for the token transfer. This implies that `cpi_authority_pda` is utilized to authorize the transfer instead of `ctx.accounts.authority`, which is typically expected to be the signer or authority responsible for the action.

>_ *compressed-token/src/spl_compression.rs* rust

```rust
pub fn compress_spl_tokens<'info>(
    inputs: &CompressedTokenInstructionDataTransfer,
    ctx: &Context<'_, '_, '_, 'info, TransferInstruction<'info>>,
) -> Result<()> {
    [...]
    transfer(
        &ctx.accounts
            .compress_or_decompress_token_account
            .as_ref()
            .unwrap()
            .to_account_info(),
        &recipient,
        &ctx.accounts.cpi_authority_pda.to_account_info(),
        [...]
    )
}
```

The attacker may set up both the `token_pool_pda` and the `compress_or_decompress_token_account` to point to the same token account by creating a token account and utilizing it as both the source and destination for the transfer. Since `cpi_authority_pda` is utilized as the authority, the attacker may authorize the transfer even if it is a self-transfer. With tokens effectively self-transferred to a pool account, the attacker may then decompress the tokens. Since the attacker controls both accounts involved in the transfer, they may effectively steal tokens from the pool.

Remediation

Ensure that `ctx.accounts.authority` is utilized as the authority for token transfers rather than `cpi_authority_pda`. This ensures that only authorized signers may approve transfers.

Patch

Resolved in 043e22a.

Changelog Path Update Error HIGH OS-LPL-ADV-02

Description

In the original implementation of `append_batch`, the `fillup_index` is calculated to determine the upper limit for computing the Merkle path. It includes the condition `self.next_index.trailing_ones() as usize + 1`, which consists of the last node in the Merkle path computation. However, during the loop that updates the `changelog` paths, there is a check `if i < self.height - 1` to update the paths. This implies it attempts to update the path even for the last node. The issue arises because the `fillup_index` calculation considers the last node (leaf) to be part of the Merkle path computation.

>_ *merkle-tree/concurrent/src/lib.rs* rust

```rust
pub fn append_batch(
    &mut self,
    leaves: &[&[u8; 32]],
) -> Result<(usize, usize), ConcurrentMerkleTreeError> {
    [...]
    for (leaf_i, leaf) in leaves.iter().enumerate() {
        // Limit until which we fill up the current Merkle path.
        let fillup_index = if leaf_i < (leaves.len() - 1) {
            self.next_index.trailing_ones() as usize + 1
        } [...]
        for i in 0..fillup_index {
            let is_left = current_index % 2 == 0;
            current_node = if is_left {
               [...]
            } else {
                H::hashv(&[&self.filled_subtrees[i], &current_node])?
            };
            if i < self.height - 1 {
                self.changelog[self.current_changelog_index].path[i + 1] = current_node;
                for leaf_j in 0..leaf_i {
                    let changelog_index =
                        (first_changelog_index + leaf_j) % self.changelog_capacity;
                    if self.changelog[changelog_index].path[i + 1] == [0u8; 32] {
                        self.changelog[changelog_index].path[i + 1] = current_node;
                    }
                }
            }
            current_index /= 2;
        }
        [...]
    }
    [...]
}
```

Therefore, when updating the `changelog` paths, it incorrectly attempts to update the path for the last node, which should not be updated. This inconsistency may result in incorrect Merkle proofs or paths stored in the `changelog`.

Remediation

Update the loop to exclude the last iteration (`fillup_index - 1`) for the `changelog` update. The `fillup_index - 1` condition should be in the `i < self.height - 1` check.

Patch

Resolved in 216b6e5.

Absence of Index Validation and Initialization HIGH OS-LPL-ADV-03

Description

`IndexedMerkleTree::update` lacks any checks to ensure that the new element (`new_element`) is inserted at its designated index (`new_element.index`). This is particularly important because the `leaf_index` in the Merkle tree is determined based on `new_element.index`. The Merkle tree relies on the correct placement of elements at their respective indices. If an element is inserted at an incorrect index, or if the index does not match the expected value, it will result in inconsistencies in the Merkle tree structure.

Remediation

Verify that `new_element` is indeed inserted at `new_element.index` in `update`. Additionally, since `init_value` is inserted at index one, ensure that `add_highest_element` is invoked as part of the tree initialization process.

Patch

Resolved in 0210a9a by passing only value instead of `IndexedElement`.

Inclusion of Lamports as Rent for Rollover Fee Calculation HIGH OS-LPL-ADV-04

Description

In the current implementation of the account-compression program, the lamports in the tree and queue are considered as rent for rollover fee calculations. This gives rise to several issues:

During the rollover process, the rent lamports are considered for withdrawal from the old Merkle tree. `compute_rollover_fee` utilizes ceiling division to calculate the fee based on the rent. Due to ceiling division, the protocol may charge slightly more in fees than required to cover the rent. For instance, if 10 lamports are the rent and the tree supports 3 node insertions, the ideal fee would be $10/3≈3.33$ lamports per insertion. However, with ceiling division, the fee is rounded up to 4 lamports per insertion. Thus, for 3 insertions, the protocol will collect $4*3 = 12$ lamports in fees instead of 10.

These extra lamports collected (2 lamports in this case) are not extracted and remain locked in the tree/queue accounts, effectively becoming inaccessible or unusable funds.

>_ *utils/src/fee.rs* rust

```rust
pub fn compute_rollover_fee(
    rollover_threshold: u64,
    tree_height: u32,
    rent: u64,
) -> Result<u64, UtilsError> {
    let number_of_transactions = 1 << tree_height;
    if rollover_threshold > 100 {
        return Err(UtilsError::InvalidRolloverThreshold);
    }
    // rent / (total_number_of_leaves * (rollover_threshold / 100))
    // (with ceil division)
    Ok((rent * 100).div_ceil(number_of_transactions * rollover_threshold))
}
```

Moreover, the rollover process considers the lamports in the new accounts as rent, allowing that amount to be fetched from the old tree/queue accounts to the fee account. Normally, the program is designed to avoid closing old tree accounts by default (as `close_threshold` is set to `None`), but by passing extra lamports to new accounts, the owner may withdraw the total lamports from the old account, resulting in the deletion of the old account. This manipulation allows the authority owner to bypass the program's default behavior, prematurely closing and deleting old tree accounts.

Remediation

These issues may be rectified by transferring `total_lamports - minimum_rent` to the `fee` account, ensuring that all lamports except the required rent are received as a fee. However, this solution may result in a rug pull. The tree authority may initiate the tree with more lamports to increase the rollover fee and eventually receive all the fees and extra lamports. Consequently, it would be better to consider the rent amount as follows: `let minimum_rent = (Rent::get()?).minimum_balance(size);` for the rollover fee calculation.

Patch

Resolved in afafb28.

Failure to Update Canopy Nodes HIGH OS-LPL-ADV-05

Description

`update` in `ConcurrentMerkleTree` is responsible for updating a leaf node and ensuring that the associated proof remains valid. However, it does not include the functionality to update the canopy nodes when a leaf node is updated. In Merkle trees, a canopy is a subset of nodes that are used in proof generation but are not part of the main tree structure. When generating Merkle proofs, canopy nodes are utilized to compute intermediate hashes.

>_ *merkle-tree/concurrent/src/lib.rs* rust

```rust
pub fn update(
    &mut self,
    changelog_index: usize,
    old_leaf: &[u8; 32],
    new_leaf: &[u8; 32],
    leaf_index: usize,
    proof: &mut BoundedVec<[u8; 32]>,
) -> Result<(usize, usize), ConcurrentMerkleTreeError> {
    let expected_proof_len = self.height - self.canopy_depth;
    if proof.len() != expected_proof_len {
        return Err(ConcurrentMerkleTreeError::InvalidProofLength(
            expected_proof_len,
            proof.len(),
        ));
    }
    if leaf_index >= self.next_index() {
        return Err(ConcurrentMerkleTreeError::CannotUpdateEmpty);
    }
    if self.canopy_depth > 0 {
        self.update_proof_from_canopy(leaf_index, proof)?;
    }
    if self.changelog_capacity > 0 && changelog_index != self.changelog_index() {
        self.update_proof_from_changelog(changelog_index, leaf_index, proof)?;
    }
    self.validate_proof(old_leaf, leaf_index, proof)?;
    self.update_leaf_in_tree(new_leaf, leaf_index, proof)
}
```

If canopy nodes are not updated along with the main tree structure, the generated proofs will include outdated or incorrect canopy node hashes. This inconsistency will result subsequent proof validation failing.

Remediation

Include logic to update the canopy nodes in `update`, along with the main tree structure, when a leaf node is updated.

Patch

Resolved in 587ca5f.

Hash Collision In Merkle Tree HIGH OS-LPL-ADV-06

Description

The vulnerability stems from the assumption in `create_cpi_accounts_and_instruction_data` that all remaining accounts in the `ctx.remaining_accounts` vector are unique. Specifically, it assumes that different Merkle tree accounts should only appear once in the vector. However, if an attacker manipulates `ctx.remaining_accounts` to include the same Merkle tree multiple times (`remaining_accounts[0] = tree_key` and `remaining_accounts[1] = tree_key`), then the `merkle_tree_index` values for different `output_compressed_accounts` may both refer to the same tree but at different indices.

>_ *merkle-append.rs* rust

```rust
pub fn create_cpi_accounts_and_instruction_data ([...]) -> Result<()> {
    [...]
    output_compressed_account_indices[j] = mt_next_index + num_leaves_in_tree;
    num_leaves_in_tree += 1;
    [...]
    // Compute output compressed account hash.
    output_compressed_account_hashes[j] = account
        .compressed_account
        .hash_with_hashed_values::<Poseidon>(
            &hashed_owner,
            &hashed_merkle_tree,
            &output_compressed_account_indices[j],
        )?;
    [...]
}
```

When `merkle_tree_index` changes between iterations, `num_leaves_in_tree` is reset to zero. Thus, `output_compressed_account_indices[j]` for both compressed accounts will be the same, resulting in identical leaf indices in the same tree. This, in turn, will generate the same hash for different compressed accounts, resulting in a hash collision in the Merkle tree.

Remediation

Introduce a validation step to ensure that each Merkle tree in `ctx.remaining_accounts` is unique before processing.

Patch

Resolved in bb222d4.

Inadequate Verification of Non-executable Programs MEDIUM OS-LPL-ADV-07

Description

In `emit_indexer_event` , the purpose of verifying the public key of the `noop_program` and its executability is to ensure that the event is emitted through a legitimate and intended no-operation (`noop`) program. However, the current condition for validating the `noop_program` has a logical flaw. It checks if the public key is correct and the program is executable simultaneously, which may allow non-executable programs to pass. This may result in a failed invocation when trying to emit the event.

>_ *merkle-tree/concurrent/src/lib.rs* rust

```rust
pub fn emit_indexer_event(data: Vec<u8>, noop_program: &AccountInfo) -> Result<()> {
    if noop_program.key() != Pubkey::new_from_array(NOOP_PUBKEY) && noop_program.executable {
        return err!(AccountCompressionErrorCode::InvalidNoopPubkey);
    }
    let instruction = Instruction {
        program_id: noop_program.key(),
        accounts: vec![],
        data,
    };
    invoke(&instruction, &[noop_program.to_account_info()])?;
    Ok(())
}
```

Remediation

Update the condition in `emit_indexer_event` to reject a `noop_program` if it is not the correct `noop` program (`noop_program.key() != Pubkey::new_from_array(NOOP_PUBKEY)`) and if it is not executable (`!noop_program.executable`).

Patch

Resolved in a91cc3d.

Lack of Merkle Tree Association Check MEDIUM OS-LPL-ADV-08

Description

In `invoke_cpi::process_cpi_context`, the lack of checks to ensure that all the input and output compressed accounts belong to the same Merkle tree as specified by `CpiContextAccount.associated_merkle_tree` introduces a significant vulnerability. The `CpiContextAccount` is expected to maintain a reference to a specific Merkle tree. This Merkle tree is utilized to ensure that the compressed accounts are validated correctly through the context's proof.

Without checking that the input and output compressed accounts are associated with the same Merkle tree as the `CpiContextAccount`, it is possible to utilize accounts from different Merkle trees. This may result in incorrect or invalid proof verifications because the proofs and the associated Merkle tree hashes may not match.

Remediation

Verify that the Merkle tree associated with the compressed accounts is the same as the `associated_merkle_tree` in `CpiContextAccount`. This ensures that all accounts are validated against the same Merkle tree.

Patch

Resolved in 0e0fec6.

Insecure Context Management MEDIUM OS-LPL-ADV-09

Description

The vulnerability stems from the lack of explicit access control when utilizing `CpiContextAccount` in `invoke_cpi`. In `InvokeCpiInstruction`, `CpiContextAccount` is meant to hold or manage context information required for cross-program invocation (CPI). However, there are no specific checks or restrictions on who may modify or access this `CpiContextAccount`. This lack of control may be exploited by injecting a different proof during the utilization of `CpiContextAccount`, especially when the context setting spans multiple transactions instead of one transaction, resetting the context account and removing the unutilized invoke inputs.

Remediation

Implement strict access control mechanisms to ensure that only authorized entities may modify or interact with `CpiContextAccount`.

Patch

Resolved in 0e0fec6.

Incorrect Element Dequeuing MEDIUM OS-LPL-ADV-10

Description

`dequeue_at_with_low_element_index` in the `IndexedArray` module is designed to remove an element from the array at a given index and update references accordingly. However, the function is missing a check to ensure that `elements[low_element_index].next_index == index`. Without this check, the function may update the `next_index` of `elements[low_element_index]` to point to an incorrect element. If the `next_index` of the element at `low_element_index` does not point to the element at `index`, the function may end up removing an element that should not be removed. This will result in unintended dequeuing and corruption of the internal structure of the `IndexedArray`.

Remediation

Add a check to ensure that the `next_index` of the element at `low_element_index` actually points to the element at `index` before proceeding with the removal.

Patch

Resolved in 92930cd by removing the dequeue method.

Improper Equality Comparison MEDIUM OS-LPL-ADV-11

Description

`CyclicBoundedVec::eq` compares the underlying data of two vectors without accounting for the cyclic nature of the vector, which may result in incorrect outcomes. This is because the data in `CyclicBoundedVec` may be stored in a non-contiguous manner due to the cyclic indexing, implying that the order of elements in the data array may not match the logical order of elements in the vector.

>_ *merkle-tree/bounded-vec/src/lib.rs* rust

```rust
impl<'a, T> PartialEq for CyclicBoundedVec<'a, T>
where
    T: Clone + PartialEq,
{
    fn eq(&self, other: &Self) -> bool {
        self.data[..self.length].iter().eq(other.data.iter())
    }
}
```

The function creates a slice from the start of `self.data` up to `self.length` and iterates over the elements of the slice. It compares the elements of the iterator from `self.data` with those from `other.data`. This approach assumes that `self.data` and `other.data` are in the same order, but in a `CyclicBoundedVec`, the logical order may be different due to the cyclic indexing, specifically the `first_index` and `last_index`.

Remediation

Ensure the comparison accounts for the cyclic ordering by iterating through the elements in their logical order to properly compare two `CyclicBoundedVec` instances.

Patch

Resolved in 534d90f.

Program Account Misuse MEDIUM OS-LPL-ADV-12

Description

The `authority` field is a `Signer<'info>`. This implies it is expected to be an account that has signed the transaction and, therefore, is responsible for authorizing the action. If the authority signer is a program, the program admin may misuse their authority. Since the admin has the corresponding private key for the program's public key, they may generate the `Signer` for the program's `pubkey` and utilize the `invoke` instruction to delete program-owned accounts or transfer lamports.

>_ *system/src/invoke/instruction.rs* rust

```rust
#[derive(Accounts)]
pub struct InvokeInstruction<'info> {
    #[account(mut)]
    pub fee_payer: Signer<'info>,
    pub authority: Signer<'info>,
    [...]
}
```

Remediation

Implement checks to ensure that the authority signer is not a program account.

Patch

Resolved in 203296b by checking that accounts used in invoke cannot have any data, and accounts owned by a program must contain data.

Inaccurate Reconstruction of Cyclic Bounded Vector MEDIUM OS-LPL-ADV-13

Description

The vulnerability in `ConcurrentMerkleTree::copy_from_bytes` is related to the reconstruction of `CyclicBoundedVec` structures from byte slices. `CyclicBoundedVec` maintains its elements in a circular buffer with specific indices to keep track of the start (`first_index`) and end (`last_index`) of the valid elements. When reconstructing a `CyclicBoundedVec` from a byte slice, it is crucial to consider these indices to preserve the correct order of elements.

In the current implementation of `copy_from_bytes`, the reconstruction of `CyclicBoundedVec` does not account for these indices. It only copies elements based on their positions in the byte slice, resulting in potential misordering. This issue is present in various places in both concurrent and indexed Merkle trees.

Remediation

Ensure the reconstruction process restores the `first_index` and `last_index` of the `CyclicBoundedVec`.

Patch

Resolved in 534d90f.

Risk of Double Free Errors MEDIUM OS-LPL-ADV-14

Description

The vulnerability concerns deallocation and the potential for double-free errors in the context of `BoundedVec` and `CyclicBoundedVec` created via `from_raw_parts`. When a `BoundedVec` or `CyclicBoundedVec` is created utilizing `from_raw_parts`, the memory referenced by the raw pointer is expected to be managed externally. If these structures attempt to deallocate this memory when they are dropped, it may result in a double-free error if the original owner of the memory also tries to deallocate it.

Remediation

Avoid deallocation to prevent any possibility of double-free errors.

Patch

Resolved in 534d90f.

Inconsistency in Prime Number Calculation LOW OS-LPL-ADV-15

Description

The vulnerability arises from the way the `find_next_prime` adjusts `n` to find the next prime number greater than `n`. All prime numbers greater than three are either of the form $6k + 1$ or $6k + 5$. In the case where the (`remainder != 0.0`), the function currently adjusts `n` to `n + 6.0 - remainder`. This adjustment ensures that `n` becomes the nearest number of the form $6k + 5$. However, it does not explicitly check for a prime of the $6k + 1$ form. This results in a situation where the function incorrectly identifies the next prime number, jumping to the next prime when the given number is prime and in the form $6k + 1$.

>_ *utils/src/prime.rs* rust

```rust
pub fn find_next_prime(mut n: f64) -> f64 {
    [...]
    // Ensure the candidate is of the form 6k - 1 or 6k + 1.
    let remainder = n % 6.0;
    if remainder != 0.0 {
        n = n + 6.0 - remainder;

        let candidate = n - 1.0;
        if is_prime(candidate) {
            return candidate;
        }
    }
    loop {
        let candidate = n + 1.0;
        if is_prime(candidate) {
            return candidate;
        }
        let candidate = n + 5.0;
        if is_prime(candidate) {
            return candidate;
        }

        n += 6.0;
    }
}
```

Remediation

Explicitly check both $n - 1.0$ and $n + 5.0$ after adjusting `n` to $6k + 5$.

Patch

Resolved in f0a7b5b.

Redundant Size Calculation LOW OS-LPL-ADV-16

Description

`expand::aligned_sized` is designed to compute the size of a structure by summing up the sizes of its individual fields. It does this by either utilizing `core::mem::size_of::<T>` to compute the size of each field (if no specific size is provided) or by utilizing a size specified by the `#[size = value]` attribute on the field if it is present. However, if a field already has a `#[size = value]` attribute, the function should utilize this size directly and not recalculate it via `size_of`. But in the current implementation, the function adds both the specified size and the `size_of` result for the same field.

Thus, the total size reported by the `LEN` constant will be larger than the actual size of the structure, as it erroneously includes both the specified size and the calculated size for fields with a `#[size = value]` attribute.

Remediation

Check if a `#[size]` attribute is present for a field. If it is present, utilize the specified size and skip adding `size_of` for that field. Ensure that only one size calculation (either specified or computed) is included for each field.

Patch

Resolved in b81cc4c.

Utilization of Incompatible Tree Height LOW OS-LPL-ADV-17

Description

`concurrent` defines `ConcurrentMerkleTree40` as a type alias for `ConcurrentMerkleTree` with the generic constant `HEIGHT` set to 40. This suggests the intention to create Merkle trees with a height of 40. However, `ZeroBytes` only supports Merkle trees up to a height of 32.

>_ *merkle-tree/concurrent/src/lib.rs* rust

```rust
pub type ConcurrentMerkleTree22<'a, H> = ConcurrentMerkleTree<'a, H, 22>;
pub type ConcurrentMerkleTree26<'a, H> = ConcurrentMerkleTree<'a, H, 26>;
pub type ConcurrentMerkleTree32<'a, H> = ConcurrentMerkleTree<'a, H, 32>;
pub type ConcurrentMerkleTree40<'a, H> = ConcurrentMerkleTree<'a, H, 40>;
```

Remediation

Ensure that the Merkle tree height does not exceed the height limitation of `ZeroBytes`.

Patch

Resolved in 93f2971.

Incorrect Root Assignment LOW OS-LPL-ADV-18

Description

In `concurrent::append_batch`, `current_node` is updated iteratively to represent the hash of nodes along the path from the leaf to the root. After computing the Merkle path for a leaf, the `current_node` is assigned as the root node for both the `changelog` entry and the list of roots. The vulnerability arises if the computation of the Merkle path does not complete correctly, particularly if the iteration breaks before reaching the root.

>_ *merkle-tree/concurrent/src/lib.rs* rust

```rust
pub fn append_batch(
    &mut self,
    leaves: &[&[u8; 32]],
) -> Result<(usize, usize), ConcurrentMerkleTreeError> {
    [...]
        for i in 0..fillup_index {
                [...]
                if i < self.height - 1 {
                    self.changelog[self.current_changelog_index].path[i + 1] = current_node;
                    [...]
                }
                current_index /= 2;
            }
            self.changelog[self.current_changelog_index].root = current_node;
            self.inc_current_root_index()?;
            self.roots.push(current_node);
            [...]
        }
    [...]
```

As a result, `current_node` may not accurately represent the actual root node of the Merkle tree but instead reflect an intermediate node or an incomplete computation. Consequently, the root node would be incorrectly recorded in the `changelog` and `roots` arrays.

Remediation

Ensure that `current_node` represents the true root of the Merkle tree before assigning it to the `changelog` and `roots` arrays.

Patch

Resolved in 93f2971 by emitting `[0u8;32]` instead of incorrect roots.

Inclusion of Duplicate Account Hash LOW OS-LPL-ADV-19

Description

The same compressed output account hash may be appended multiple times to the Merkle tree. This is feasible because a Merkle tree can accommodate multiple identical hashes, and adding a hash to the tree does not inherently prevent duplication. Although the same compressed output account hash may be added multiple times to the Merkle tree, the nullifier queue only stores each unique hash once. This implies that while multiple entries with the same hash may be recorded in the Merkle tree, the nullifier queue will recognize and prevent duplicate processing of these hashes.

Remediation

Prevent the addition of duplicate compressed output account hashes into the Merkle tree.

Patch

Resolved in bb222d4.

Utilization of Index Zero LOW

OS-LPL-ADV-20

Description

The vulnerability concerns the behavior of `dequeue_at` in `IndexedArray`, when handling index zero. Throughout the code, index zero is reserved for zero-value elements and is treated as a special case. This implies that operations on index zero need to be carefully managed to maintain the consistency and integrity of the array. `dequeue_at` does not currently enforce this special handling for index zero. As a result, it allows the removal of the element at index zero without considering its unique status.

>_ *merkle-tree/concurrent/src/lib.rs* rust

```rust
pub fn dequeue_at(
    &mut self,
    index: I,
) -> Result<Option<IndexedElement<I>>, IndexedMerkleTreeError> {
    match self.elements.get(usize::from(index)) {
        Some(node) => {
            let low_element_index = self.find_low_element_index_for_existent(&node.value)?;
            self.dequeue_at_with_low_element_index(low_element_index, index)
        }
        None => Ok(None),
    }
}
```

Remediation

Implement a check at the start of `dequeue_at` to ensure that it does not operate on index zero.

Patch

Resolved in 92930cd by removing the dequeue method.

Incorrect Index Copy LOW OS-LPL-ADV-21

Description

There is a potential memory access issue in `HashSet::from_bytes_copy`. `next_value_index` allocates memory for a single `usize` value. It is utilized to store the index of the next element to be inserted into the hash set. The function attempts to copy the `next_value_index` from `bytes[24..32]`. However, this offset range corresponds to the memory location where `HashSetCell` elements are expected to begin. Thus, the copied `next_value_index` will point to `HashSetCell` elements.

>_ *merkle-tree/hash-set/src/lib.rs* rust

```rust
pub unsafe fn from_bytes_copy(bytes: &mut [u8]) -> Result<Self, HashSetError> {
    [...]
    unsafe {
            *next_value_index = usize::from_ne_bytes(bytes[24..32].try_into().unwrap());
        }
    [...]
}
```

Remediation

Update `HashSet::from_bytes_copy` such that `next_value_index` is copied from `bytes[16..24]`.

Patch

Resolved in 7e755b2

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-LPL-SUG-00	Circuit public inputs, such as `input_compressed_account_hashes` and fetched roots, are not verified to be less than the `bn254` field order (`p`).
OS-LPL-SUG-01	`get_input_compressed_accounts_with_merkle_context_and_check_signer` does not ensure consistency between `input_token_data.delegated_amount` and `input_token_data.delegate_index` .
OS-LPL-SUG-02	There are several instances where proper validation is not done, resulting in potential security issues.
OS-LPL-SUG-03	Additional safety checks may be incorporated within the codebase to make it more robust and secure.
OS-LPL-SUG-04	The overall code may be streamlined further to reduce complexity, eliminate redundancy, and enhance readability.
OS-LPL-SUG-05	Recommendation for modifying the codebase for improved efficiency.
OS-LPL-SUG-06	Suggestions regarding inconsistencies in the codebase and ensuring adherence to coding best practices.
OS-LPL-SUG-07	The codebase contains multiple cases of redundancy that should be removed for better maintainability and clarity.

Missing Field Order Constraint

OS-LPL-SUG-00

Description

There is a flaw that potentially compromises the integrity of zero-knowledge (zk) proofs and Merkle tree operations. In zk-SNARK, it is crucial that all public inputs (such as hashes of compressed accounts and Merkle tree roots) are within certain mathematical constraints. Specifically, these values should be within the field order of the elliptic curve or the zk-SNARK's underlying group. For example, in the `bn254` curve, this order is denoted as `p`.

Currently, the values of `input_compressed_account_hashes` and fetched roots are not checked to ensure they are less than `p`. Thus, if a proof is valid for a given hash value `x`, it may also be valid for `x + p` due to the cyclic nature of the field. Therefore, if `x` is a valid hash value, `x + p` is also considered valid as the `hash_with_hashed_values` of the compressed account is not checked to see if it is less than `p`, enabling an attacker to insert both `x` and `x + p` as valid hashes into the state tree.

Remediation

Ensure that all public inputs, including hash values and Merkle tree roots, are checked to be within the field order `p`.

Patch

Resolved in 8892dae.

Inconsistency in Delegate Validation OS-LPL-SUG-01

Description

In the current implementation, there is a check to ensure that if `delegated_amount` is specified (`Some)`, `delegate_index` should also be specified (`Some`). However, it is not explicitly verified that the opposite is true. If `delegate_index` is `Some`, `delegated_amount` should also be `Some`. This may result in inconsistent states where a delegate is referenced without a corresponding delegated amount.

>_ *compressed-token/src/process_transfer.rs* rust

```rust
pub fn get_input_compressed_accounts_with_merkle_context_and_check_signer(
    &self,
    signer: &Pubkey,
    remaining_accounts: &[AccountInfo<'_>],
) -> Result<( [...] )>
{
    [...]
    for input_token_data in self.input_token_data_with_context.iter() {
        [...]
        if input_token_data.delegated_amount.is_some()
            && input_token_data.delegate_index.is_none()
        {
            return err!(crate::ErrorCode::DelegateUndefined);
        }
        let compressed_account = CompressedAccount {
            owner: crate::ID,
            lamports: input_token_data.is_native.unwrap_or_default(),
            data: None,
            address: None,
        };
        [...]
    [...]
}
```

If the `delegate_index` is `Some` but `delegated_amount` is `None`, it implies that the authorization check is bypassed. This will allow a delegate to be included in the hash calculation or other critical operations without a valid delegated amount. If the delegate's involvement is included in the hash calculation for `TokenData`, but there is no corresponding delegated amount, the integrity of the hash may be compromised.

Remediation

Enforce consistency between `delegated_amount` and `delegate_index`. Specifically, add a check to ensure that both are either present or absent together.

Patch

This check was removed in 03e374e.

Missing Validation Logic OS-LPL-SUG-02

Description

1. In the current implementation of `initialize_group_authority`, the `authority` is passed as a function argument. Passing the `authority` as a function argument implies that the caller has to provide the correct public key for the new authority. If this value is not properly verified, it may result in unauthorized access or misconfiguration.

   >_ *account-compression/src/lib.rs* rust

   ```rust
   pub fn initialize_group_authority<'info>(
       ctx: Context<'_, '_, '_, 'info, InitializeGroupAuthority<'info>>,
       authority: Pubkey,
   ) -> Result<()> {
       let seed_pubkey = ctx.accounts.seed.key();
       set_group_authority(
           &mut ctx.accounts.group_authority,
           authority,
           Some(seed_pubkey),
       )?;
       Ok(())
   }
   ```

2. `nullify_state::insert_nullifier` checks the length of only the first element in the proofs array. It assumes that all subsequent elements in the `proofs` array have the same length as the first element. However, the vector may be heterogeneous (`proofs[0].len() != proofs[1].len()`).

3. In `nullify_state::insert_nullifiers`, the nullify operation involves inserting nullifiers into the `nullifier` queue. If `leaf_cell.value_bytes()` is equal to `ZERO_BYTES[0]`, it indicates that the leaf cell holds a null value. Nullifying an empty or null value is redundant because it does not add meaningful data to the `nullifier` queue.

4. `utils::is_prime` utilizes a floating-point number (`f64`) as its input type. This is not ideal for prime-checking since the concept of a prime number does not apply to non-integer values, rendering the need for a floating-point type (`f64`) unnecessary.

Remediation

1. Ensure to take the `authority` from the accounts instead of from the function arguments.
2. Verify the length of each proof in the proofs array. Throw a custom error if any proof's length does not match the expected length, ensuring graceful error handling.
3. Skip the nullify operation if `leaf_cell.value_bytes() == ZERO_BYTES[0]` to optimize the function by avoiding unnecessary operations.
4. Rewrite the function to utilize `u64` (an unsigned 64-bit integer type) instead of `f64`.

Patch

1. Issue #2 was resolved in 51d076e.
2. Issue #3 will be implemented in a subsequent release.
3. Issue #4 was resolved in f0a7b5b.

Additional Safety Checks OS-LPL-SUG-03

Description

1. Modify the visibility of `fill_vectors_mut` from `public` to `private` in both `ConcurrentMerkleTree` and `IndexedMerkleTree` implementations. The parameters are sensitive concerning the alignment of the byte slices, and the function is also not called from outside the module.
2. Utilize `Layout::array` for allocation in `BoundedVec::with_capacity` for more robust handling of memory layout calculations and allocations. `Layout::array` leverages Rust's standard library functions to handle memory layout calculations, ensuring that the size and alignment calculations are handled correctly based on the type `T`.
3. In `ConcurrentMerkleTree::new` and `ConcurrentMerkleTree::from_bytes_init`, check that `height == HEIGHT` and `canopy_depth < height` to ensure that the provided height parameter matches the constant `HEIGHT` and that the `canopy_depth` is less than the overall height of the Merkle tree, respectively.
4. `invoke_cpi::process_cpi_context` should first check if `cpi_context_account.context` is empty before attempting to access its elements. This prevents the function from trying to access an element that may not exist, thus avoiding potential runtime panics.

>_ *invoke_cpi/process_cpi_context.rs* rust

```rust
pub fn process_cpi_context<'info>(
   [...]
) -> Result<Option<InstructionDataInvokeCpi>> {
    [...]
    if let Some(cpi_context) = cpi_context {
        [...]else {
            if cpi_context_account.context[0].proof != inputs.proof {
                return err!(SystemProgramError::CpiContextProofMismatch);
            } else if cpi_context_account.context.is_empty() {
                return err!(SystemProgramError::CpiContextEmpty);
            }
            [...]
        }
    }
    Ok(Some(inputs))
}
```

5. In `IndexedArray::new_element_with_low_element_index`, perform a length check before incrementing `current_node_index` to prevent potential panics that may occur if the array is full or exceeds its predefined capacity (`ELEMENTS`).

Remediation

Add the missing validations mentioned above.

Patch

1. Issue #1 was resolved by removing the function.
2. Issue #2 was resolved in 534d90f.
3. Issue #3 was resolved in 6accb20.
4. Issue #4 was resolved in 47c0616.

Code Clarity OS-LPL-SUG-04

Description

1. `ConcurrentMerkleTree::append_batch` utilizes an additional variable (`first_leaf_index`) to store the initial value of `self.next_index` and then computes the current leaf index within the loop as `first_leaf_index + leaf_i`. This process may be streamlined by directly utilizing `self.next_index` and incrementing it at the appropriate points in the code.
2. Simplify `IndexedArray::hash_element` by directly calling `element.hash(next_element.value)` instead of performing the hashing process within the function. Similarly, in `IndexedArray::hash_element`, replace `indexed_array.append(&init_value)` with `indexed_array.init()` to make the code cleaner and reduce redundancy.
3. `HashSet::find_element_iter` iterates over the range `(start_iter..num_iterations)`, which may be misleading given the parameter name `num_iterations`. To align the iteration with the parameter names and the intention, the loop should iterate from `start_iter` to `(start_iter + num_iterations)`. This ensures that the function performs exactly `num_iterations` starting from `start_iter`.

   >_ *merkle-tree/hash-set/src/lib.rs* rust

   ```rust
   pub fn find_element_iter(
       &mut self,
       value: &BigUint,
       current_sequence_number: usize,
       start_iter: usize,
       num_iterations: usize,
   ) -> Result<Option<(usize, bool)>, HashSetError> {
       let mut first_free_element: Option<(usize, bool)> = None;
       for i in start_iter..num_iterations {
           let probe_index = self.probe_index(value, i);
           let bucket = self.get_bucket(probe_index).unwrap();
       [...]
   }
   ```

4. In `invoke::process`, utilize `num_new_addresses` and `num_input_compressed_accounts` to set the lengths of the `new_address_roots` and `roots` vectors, respectively, to improve clarity and optimize memory allocation for the ectors.

Remediation

1. Utilize `self.next_index` directly instead of `first_leaf_index + leaf_i` to simplify the code and avoid potential confusion.
2. Replace the above-mentioned code snippets with their respective functions to ensure better encapsulation and clarity.
3. Iterate from `start_iter` to `(start_iter + num_iterations)` in `HashSet::find_element_iter`.
4. Implement the above modification.

Patch

1. Issue #1 resolved in 0ccb5b0.
2. Issue #2 will be implemented in a subsequent release.
3. Issue #3 was resolved in 0ccb5b0.
4. Issue #4 was resolved in 4736fca.

Code Refactoring OS-LPL-SUG-05

Description

1. Implement the `Deref` trait for the `IndexedMerkleTreeZeroCopy` and `IndexedMerkleTreeZeroCopyMut` structures to allow them to behave like references to their inner `IndexedMerkleTree` structure. This will help access methods or fields of `IndexedMerkleTree` directly through instances of `IndexedMerkleTreeZeroCopy` or `IndexedMerkleTreeZeroCopyMut`.
2. Fix the `CPI` authority seed in `verify_signer::cpi_signer_check` instead of accepting arbitrary seeds. This approach minimizes the risk of unauthorized access and ensures that only programs with the correct, fixed seed can perform actions on behalf of the CPI authority, providing a more robust and secure implementation.

>_ invoke_cpi/verify_signer.rs rust

```rust
pub fn cpi_signer_check(
    signer_seeds: &[Vec<u8>],
    invoking_program: &Pubkey,
    authority: &Pubkey,
) -> Result<()> {
    let seeds = signer_seeds
        .iter()
        .map(|x| x.as_slice())
        .collect::<Vec<&[u8]>>();
   [...]
    Ok(())
}
```

3. In `ConcurrentMerkleTree::get_changelog_event`, the `path` vector constructs a sequence of `PathNode` structures representing nodes in the Merkle tree path affected by a `changelog` entry. Utilize `Vec::with_capacity(self.height + 1)` to ensure that the `path` vector has enough initial capacity to store all `PathNode` structures without reallocation during construction, including the root node. This approach avoids unnecessary reallocations and improves efficiency, especially when constructing the `path` vector for multiple `changelog` entries.
4. In `HashSet::find_element_iter`, the loop checks if `first_free_element` is `None` before updating it, in both the `Some(bucket)` and `None` cases. In the `None` case, directly update `first_free_element` and break out of the loop. This guarantees that the function stops iterating as soon as it finds an empty slot. Consequently, there is no necessity to check if `first_free_element.is_none()` before updating it in the `None` case, as the loop is terminated immediately.

5. Move the `GroupAccess` implementation of `StateMerkleTreeAccount` to `state/public_state_merkle_tree.rs`, and in `concurrent::compute_parent_node`, rename `sibling_index` to `level` for better clarity.
6. In `process_transfer::add_token_data_to_input_compressed_accounts`, the `hashed_delegate` is computed inside the loop for each `TokenData` entry. However, if all token accounts utilize the same delegate, and this delegate is verified against the authority, then computing the `hashed_delegate` inside the loop is redundant. Instead, it may be computed once before the loop and reutilized.

Remediation

Update the codebase with the above modifications.

Patch

1. Issue #1 was resolved in 534d90f.
2. Issue #2 was resolved in 1f8ad6d.
3. The method was removed in issue #3.
4. Issue #4 was resolved in 1f8ad6d.
5. Issue #5 was resolved in 534d90f.
6. Issue #6 was skipped because an owner may want to spend multiple compressed accounts that are delegated to different delegates in the same transaction.

Code Maturity

OS-LPL-SUG-06

Description

1. Within `queue`, the comments for `queue_from_bytes_copy`, `queue_from_bytes_zero_copy_mut`, and `queue_from_bytes_zero_copy_init` mention `IndexedArray` instead of `HashSet`.
2. The current implementations of `RegisterProgramToGroup` (shown below) and `UpdateGroupAuthority` ensure that the `authority` account is correctly linked to the `group_authority_pda` by explicitly checking the address within the `#[account]` attribute. The account constraint for these two instructions may be simplified.

   >_ *account-compression/src/instructions/register_program.rs* rust

   ```rust
   pub struct RegisterProgramToGroup<'info> {
       /// CHECK: Signer is checked according to authority pda in instruction.
       #[account(mut, address=group_authority_pda.authority
           ↪  @AccountCompressionErrorCode::InvalidAuthority)]
       pub authority: Signer<'info>,
       [...]
   }
   ```

3. The size allocated for the `merkle_tree_canopy` and `state_merkle_tree_canopy` fields in the `AddressMerkleTreeAccount` and `StateMerkleTreeAccount` structures, respectively, is incorrect given the specified canopy height. The canopy height is 10, and thus, the bytes required should be $2^{10} * 32 = 32768$.
4. In the current implementation, `hash-set::find_element_index` iterates through all possible bucket indices (`0..self.capacity`) to find the element. This may be inefficient, especially for large capacities.

Remediation

1. Replace `IndexedArray` with `HashSet` in the comments for the above functions.
2. Utilize the `has_one = authority` attribute within the `#[account]` attribute for `group_authority_pda` in the two instructions to simplify the constraint.
3. Ensure the size allocated for the `merkle_tree_canopy` and `state_merkle_tree_canopy` fields is updated to 32768.
4. Limit the number of probes to a smaller, constant number instead of utilizing the hard-coded value of 20.

Patch

1. Issue #1 was resolved in 036b50a.
2. Issue #4 was resolved in 0ccb5b0.

Code Redundancy OS-LPL-SUG-07

Description

1. In `process_initialize_nullifier_queue`, instead of manually initializing `AccessMetadata` and `RolloverMetadata`, invoke the constructors `AccessMetadata::new` and `RolloverMetadata::new` to encapsulate the logic of metadata initialization, thereby reducing redundancy.
2. In `HasherError::from`, both variants (`UnknownSolanaSyscall` and `PoseidonSyscall`) are associated with the same error code (`7002`). This will result in ambiguity when handling errors, as the same error code will be utilized for different underlying reasons. Ensure `UnknownSolanaSyscall` utilizes a different error code unique from other codes.

>_ *merkle-tree/hasher/src/errors.rs* rust

```rust
impl From<HasherError> for u32 {
    fn from(e: HasherError) -> u32 {
        match e {
            [...]
            HasherError::PoseidonSyscall(e) => (u64::from(e)).try_into().unwrap_or(7002),
            HasherError::UnknownSolanaSyscall(e) => e.try_into().unwrap_or(7002),
        }
    }
}
```

3. In `CyclicBoundedVec::push`, the `else if !self.is_empty()` condition is redundant and may be simplified to `else` because if the vector is not empty, it will always fall into the last `else` block.
4. In `process_initialize_address_merkle_tree`, the call to `load_merkle_tree_mut` is redundant because `load_merkle_tree_init` will have already updated the mutable state of `address_merkle_tree` to include the initialized Merkle tree. Thus, the return value of `load_merkle_tree_init` may be utilized for `address_merkle_tree_inited` instead of reloading it through `load_merkle_tree_mut`.
5. Remove the following unutilized variables: `program_owner` in `AccessMetadata`, `network_fee` in `RolloverMetadata`, `CPI_SEED` in `invoke_cpi`, and `system_program` in `InitializeStateMerkleTreeAndNullifierQueue`.
6. In `TokenData::get_hash_inputs_with_hashed_values`, the `native_amount` matching may be avoided as it is already checked to be `is_some`. The same applies to `self.is_native` in `TokenData::hash`.

Remediation

Remove the redundant and unutilized code instances highlighted above.

Patch

1. Issue #2 was resolved in ddb97c8.
2. `self.is_native` was removed in issue #6.

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.