Account Compression

Audit

Presented by:

OtterSec contact@osec.io

Michał Bochnak embe221ed@osec.io

Robert Chen notdeghost@osec.io

Contents

Appendices

01 | Executive Summary

Overview

Solana engaged OtterSec to perform an assessment of the `account-compression` program. This assessment was conducted between October 3rd and October 21st, 2022.

Critical vulnerabilities were communicated to the team prior to the delivery of the report to speed up remediation. After delivering our audit report, we worked closely with the team over to streamline patches and confirm remediation. We delivered the final confirmation of the patches on October 24th, 2022.

Key Findings

Over the course of this audit engagement, we have produced 5 total findings.

In particular, we identified an off-by-one issue in the concurrent merkle tree implementation (OS-CMT-ADV-00). If this library was used separately, it could lead to security implications. Luckily, the particular usage in `account-comprssion` was mitigated by additional checks.

We also identified minor errors in the whitepaper (OS-CMT-SUG-03) and misleading error messages (OS-CMT-SUG-01).

Overall, we commend the Solana team for their attention to detail and responsiveness throughout the audit.

02 | Scope

The source code was delivered to us in a git repositories at github.com/solana-labs/solana-program-library/tree/master/account-compression. This audit was performed against commit `6e81794`.

Alongside with source code, we were provided with whitepaper, `Compressing Digital Assets with Concurrent Merkle Trees`

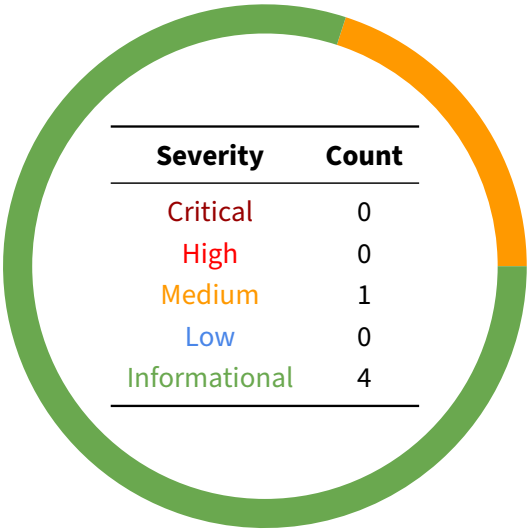
A brief description of the programs is as follows.

Name	Description
account-compression	Interface for composing smart-contracts to create and use SPL Concurrent Merkle Trees
concurrent-merkle-tree	Concurrent Merkle Tree implementation

03 | Findings

Overall, we report 5 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 help mitigate future vulnerabilities.



Severity	Count
Critical	0
High	0
Medium	1
Low	0
Informational	4

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-CMT-ADV-00	Medium	Resolved	Off-by-one in concurrent merkle tree indexing

OS-CMT-ADV-00 [med] [resolved] | Off-By-One in Leaf Index

Description

The `rightmost_proof.index` value represents the index of first empty leaf. In case the tree is full, it's value is equal to `1<<MAX_DEPTH` which doesn't correspond to any valid leaf index.

In functions `set_leaf()` and `prove_leaf()` the leaf index is compared to value of `rightmost_proof.index` to determine whether it is valid.

libraries/concurrent-merkle-tree/src/concurrent_merkle_tree.rs RUST

```rust
pub fn prove_leaf(
    &self,
    current_root: Node,
    leaf: Node,
    proof_vec: &[Node],
    leaf_index: u32,
) -> Result<(), ConcurrentMerkleTreeError> {
    check_bounds(MAX_DEPTH, MAX_BUFFER_SIZE);
    if leaf_index > self.rightmost_proof.index {
        solana_logging!(
            "Received an index larger than the rightmost index {} > {}",
            leaf_index,
            self.rightmost_proof.index
        );
        Err(ConcurrentMerkleTreeError::LeafIndexOutOfBounds)
    } else {
        let mut proof: [Node; MAX_DEPTH] = [Node::default(); MAX_DEPTH];
        fill_in_proof::<MAX_DEPTH>(proof_vec, &mut proof);
        let valid_root =
            self.check_valid_leaf(current_root, leaf, &mut proof,
    ↪   leaf_index, true)?;
        if !valid_root {
            solana_logging!("Proof failed to verify");
            return Err(ConcurrentMerkleTreeError::InvalidProof);
        }
        Ok(())
    }
}
```

libraries/concurrent-merkle-tree/src/concurrent_merkle_tree.rs RUST

```rust
pub fn set_leaf(
    &mut self,
    current_root: Node,
    previous_leaf: Node,
    new_leaf: Node,
    proof_vec: &[Node],
    index: u32,
) -> Result<Node, ConcurrentMerkleTreeError> {
    check_bounds(MAX_DEPTH, MAX_BUFFER_SIZE);
    if index > self.rightmost_proof.index {
        Err(ConcurrentMerkleTreeError::LeafIndexOutOfBounds)
    } else {
        let mut proof: [Node; MAX_DEPTH] = [Node::default(); MAX_DEPTH];
        fill_in_proof::<MAX_DEPTH>(proof_vec, &mut proof);

        log_compute!();
        self.try_apply_proof(
            current_root,
            previous_leaf,
            new_leaf,
            &mut proof,
            index,
            true,
        )
    }
}
```

Note that a leaf of `index == 1<<MAX_DEPTH` will be treated as valid. This is incorrect, as leafs are indexed from `0` to `(1<<MAX_DEPTH)-1`

Proof of Concept

1. Initialize concurrent merkle tree with `MAX_DEPTH=3`
2. Append 8 leafs in order to fill the tree and set `rightmost_proof.index` to `8==1<<MAX_DEPTH`
3. Run `set_leaf()` with following parameters:
    - `current_root` set to current root value
    - `previous_leaf` set to value of leaf with index `0`
    - `new_leaf` with the desired new value
    - `proof_vec` with the proof for leaf with index `0`
    - `index` set to `8`

4. Run `set_leaf()` for leaf with index `0`

Executing the last step will output the following result

```text
thread 'test_replaces' panicked at 'attempt to subtract with overflow',
    ↪  ./libraries/concurrent-merkle-tree/src/changelog.rs:73:33
note: run with `RUST_BACKTRACE=1` environment variable to display a
    ↪  backtrace
test test_replaces ... FAILED
```

Remediation

Any function that takes an index as a parameter should confirm that it is

- less than or equal to `rightmost_proof.index`
- **less than** `1<<MAX_DEPTH`

Patch

libraries/concurrent-merkle-tree/src/concurrent_merkle_tree.rs DIFF

```
@@ -186,7 +186,7 @@ impl<const MAX_DEPTH: usize, const MAX_BUFFER_SIZE:
    ↪  usize>
         leaf_index: u32,
     ) -> Result<(), ConcurrentMerkleTreeError> {
         check_bounds(MAX_DEPTH, MAX_BUFFER_SIZE);
-        if leaf_index > self.rightmost_proof.index {
+        if leaf_index > self.rightmost_proof.index || leaf_index >= 1 <<
    ↪  MAX_DEPTH {
             solana_logging!(
                 "Received an index larger than the rightmost index {} >
    ↪  {}",
                 leaf_index,
@@ -314,7 +314,7 @@ impl<const MAX_DEPTH: usize, const MAX_BUFFER_SIZE:
    ↪  usize>
         index: u32,
     ) -> Result<Node, ConcurrentMerkleTreeError> {
         check_bounds(MAX_DEPTH, MAX_BUFFER_SIZE);
-        if index > self.rightmost_proof.index {
+        if index > self.rightmost_proof.index || index >= 1 << MAX_DEPTH
    ↪  {
             Err(ConcurrentMerkleTreeError::LeafIndexOutOfBounds)
```

```
        } else {
            let mut proof: [Node; MAX_DEPTH] = [Node::default();
    ↪  MAX_DEPTH];
```

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 could lead to security issues in the future.

ID	Description
OS-CMT-SUG-00	Not fully specified list of allowed permutations
OS-CMT-SUG-01	Potentially misleading proof state claims in error messages
OS-CMT-SUG-02	Inconsistent default values
OS-CMT-SUG-03	Typo in algorithm 1 and incorrect definition of rightmost leaf

OS-CMT-SUG-00 | Inaccurate Depth Docstring

Description

The docstring above `ConcurrentMerkleTreeHeader` definition is presented below

concurrent_merkle_tree_header.rs RUST

```rust
/// Only the following permutations are valid:
///
/// | max_depth | max_buffer_size       |
/// | --------- | --------------------- |
/// | 14        | (64, 256, 1024, 2048) |
/// | 20        | (64, 256, 1024, 2048) |
/// | 24        | (64, 256, 512, 1024, 2048) |
/// | 26        | (64, 256, 512, 1024, 2048) |
/// | 30        | (512, 1024, 2048) |
///
#[repr(C)]
#[derive(AnchorDeserialize, AnchorSerialize)]
pub struct ConcurrentMerkleTreeHeader {
    /// Account type
    pub account_type: CompressionAccountType,
    /// Versioned header
    pub header: ConcurrentMerkleTreeHeaderData,
}
```

However, it is possible to initialize and operate on Concurrent Merkle Tree with depth equal to 3 and 5

Remediation

One of two things should happen:

1. disable possibility of tree with depth 3 and 5
2. update docstring with all possible permutations

OS-CMT-SUG-01 [resolved] | Misleading Proof State Errors

Description

There is an error for the specific case whenever the leaf's value changed since the state indicated by the current root value. Its definition is presented below

RUST

```rust
#[error(
    "Valid proof was passed to a leaf, but its value has changed since
    ↪ the proof was issued"
)]
LeafContentsModified,
```

However, the assumption that the proof was valid is incorrect here. At this moment, we cannot say if the proof was valid or not. Even if the proof will be valid, it can still be a result of proof autocompletion.

Remediation

Change the error message to not assume that the valid proof was passed.

OS-CMT-SUG-02 [resolved] | Inconsistent Defaults

Description

The `trait Default` for `ConcurrentMerkleTree` has following implementation

concurrent_merkle_tree.rs RUST

```rust
impl<const MAX_DEPTH: usize, const MAX_BUFFER_SIZE: usize> Default
    for ConcurrentMerkleTree<MAX_DEPTH, MAX_BUFFER_SIZE>
{
    fn default() -> Self {
        Self {
            sequence_number: 0,
            active_index: 0,
            buffer_size: 0,
            change_logs: [ChangeLog::<MAX_DEPTH>::default();
    ↪   MAX_BUFFER_SIZE],
            rightmost_proof: Path::<MAX_DEPTH>::default(),
        }
    }
}
```

Meaning that the current state can be described as:

- `active_index` is pointing at the first element of `change_logs` buffer
- `buffer_size` which describes the current size of a `change_logs` buffer is equal to `0`

Due to fact that functions `append` and `set_leaf` are not checking if the tree was initialized, it can result in following errors being thrown.

RUST

```
---- test_replaces stdout ----
thread 'test_replaces' panicked at 'attempt to subtract with overflow',
  libraries/concurrent-merkle-tree/src/concurrent_merkle_tree.rs:409:56
note: run with `RUST_BACKTRACE=1` environment variable to display a
    ↪  backtrace

---- test_append stdout ----
thread 'test_append' panicked at 'called `Result::unwrap()` on an `Err`
    ↪  value: TreeAlreadyInitialized',
  libraries/concurrent-merkle-tree/tests/tests.rs:51:26
note: run with `RUST_BACKTRACE=1` environment variable to display a
    ↪  backtrace
```

Remediation

Both functions should check if the tree was already initialized before performing further actions.

OS-CMT-SUG-03 [resolved] | Whitepaper Errors

Description

There is a typo in `Algorithm 1` in the procedure `UpdatePathToLeaf`

```
procedure UpdatePathToLeaf(tree, leaf, i, proof)
  changeLog ← NewChangeLog(i)
  node ← leaf
  for j ← 0 to Length(proof) do
    PushBack(changeLog.path, node)
    node ← ComputeParentNode(node, proof[j], i, j)
  end for
  PushFront(tree.changeLogs, changeLog)
  PushFront(tree.rootBuffer, node)
end procedure
```

the `PushFront` calls at the end of the procedure should in fact be `PushBack` calls.

The definition of rightmost leaf as:
`the leaf node at the first empty leaf index`
is also incorrect.

Remediation

Replace `PushFront` calls with `PushBack`

Change definition of rightmost leaf to
`the leaf node at the last non-empty leaf index`
and add definition of rightmost index being
`the leaf index of first empty leaf node`

A | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the General Findings section.

Critical	Vulnerabilities that immediately lead to 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 could lead to 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 could lead to denial of service scenarios or degraded usability. Examples: • Malicious input that causes computational limit exhaustion • Forced exceptions in normal user flow
Low	Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions
Informational	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants • Improved input validation