Code Inc Splitter

Audit

Presented by:

OtterSec contact@osec.io

Nicola Vella nick0ve@osec.io

Robert Chen r@osec.io

Contents

01 | Executive Summary

Overview

Code Inc. engaged OtterSec to perform an assessment of the `splitter` program. This assessment was conducted between May 22nd and May 26th, 2023. For more information on our auditing methodology, see Appendix B.

Key Findings

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

In particular, we made recommendations regarding the enhancement of input validation for an instruction (OS-SPT-SUG-00), the inclusion of a mechanism to manage compute budget limits (OS-SPT-SUG-01), and the unnecessary requirement for a payer account in instructions that do not demand a rent payer (OS-SPT-SUG-02).

02 | Scope

The source code was delivered to us in a git repository at github.com/code-wallet/code-program-library/tree/main/splitter. This audit was performed against commit 3fa2ce1.

The Splitter program enables users to split token transfers into two instructions while preserving the non-custodial nature of the transfer.

The Splitter program facilitates the splitting process by creating a treasury account to maintain the non-custodial nature of the original transfer. Controlled by a reliable entity, this account is an intermediary that approves the divided transfer. The program ensures the division of a single transfer into two components: one from the program to the destination and another from the source to the program. The total transfer amount remains unaffected.

Despite its role in enabling the split transfer, the treasury account does not possess control over the tokens beyond its mandate to approve the transfer. An additional safeguard exists as a precondition: if the destination account does not receive the tokens prior to debiting the source's account, the source account remains untouched.

Since the program's development progressed with an integration with the Code Wallet application in mind, our security assessment included an in-depth study of the interactions between on-chain and off-chain programs. We thank the developers for being responsive and helpful throughout the audit process.

03 | Findings

Overall, we reported 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	0
Low	0
Informational	5

04 | General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may lead to security issues in the future.

ID	Description
OS-SPT-SUG-00	`InitializePool` lacks sanity checks over `levels` and `name`.
OS-SPT-SUG-01	The loop in `MerkleTree::verify` may fail due to the compute budget limit.
OS-SPT-SUG-02	Requiring payer accounts in instructions not requiring a rent payer increases transaction costs.
OS-SPT-SUG-03	The code handles the `authority` and `payer` accounts as the same entity.
OS-SPT-SUG-04	Double-spending due to the commitment account not being verified in `TransferWithCommitment`.

OS-SPT-SUG-00 | Insufficient Input Validation

Description

The current implementation of `MerkleTree` in `InitializePool` does not handle `levels` equaling to zero. Initializing a `MerkleTree` with `levels` set to zero results in a crash since `MerkleTree::calc_zeros` returns an empty vector.

src/state.rs RUST

```rust
    pub fn new(seeds: &[&[u8]], levels: u8) -> Self {
        let zeros = MerkleTree::calc_zeros(levels, seeds);

        MerkleTree {
            levels: levels,
            next_index: 0,
            root: zeros.last().unwrap().clone(),
            filled_subtrees: zeros.clone(),
            zero_values: zeros,
        }
    }
```

In addition, there are no checks on the length of the name string, allowing it to exceed the allocated size defined by: `NAME_DEFAULT_SIZE = 32`.

Remediation

Add additional input validations to `InitializePool`.

DIFF

```diff
pub fn initialize_pool(
        ctx: Context<InitializePool>,
        name: String,       // The name of the treasury pool
        levels: u8,         // Depth of the merkle tree
    ) -> Result<()> {
-       if levels as usize > MAXIMUM_MERKLE_DEPTH {
+       if levels == 0 || levels as usize > MAXIMUM_MERKLE_DEPTH {
            return err!(ErrorCode::InvalidMerkleTreeDepth);
        }
+       if name.len() == 0 || name.len() > NAME_DEFAULT_SIZE  {
+           return err!(ErrorCode::InvalidPoolName);
+       }
```

OS-SPT-SUG-01 | Compute Budget Limit

Description

The current implementation of `MerkleTree::verify` fails to account for the compute budget limit during loop execution.

src/state.rs RUST

```rust
pub fn verify(proof: &Vec<[u8; 32]>, root: [u8; 32], leaf: [u8; 32]) -> bool {
    let mut computed_hash =
↪   solana_program::hash::hash(leaf.clone().as_ref()).to_bytes();
    for proof_element in proof.into_iter() {
        computed_hash = MerkleTree::hash_left_right(computed_hash,
↪   proof_element.clone());
    }
    // Check if the computed hash (root) is equal to the provided root
    computed_hash == root
}
```

Remediation

Divide the proof verification into multiple calls, similar to the approach already used for uploads.

OS-SPT-SUG-02 | Unnecessary Use Of Payer Account

Description

All instructions in the program require a mutable payer account, even when it is not necessary for the execution of the instructions. This results in an increase in transaction costs.

One such instance is in the `SaveRecentRoot` instruction:

src/context.rs RUST

```rust
pub struct SaveRecentRoot<'info> {
    #[account(
        mut,
        has_one = authority @ ErrorCode::InvalidAuthority,
        seeds=[
            PREFIX_POOL.as_bytes(),
            pool.mint.as_ref(),
            pool.authority.as_ref(),
            pool.name.as_ref(),
        ],
        bump = pool_bump,
    )]
    pub pool: Box<Account<'info, Pool>>,

    pub authority: Signer<'info>,

    #[account(mut)]
    pub payer: Signer<'info>,
}
```

The example above requires the payer account, even though it is not involved in the execution of the `SaveRecentRoot` instruction.

Remediation

Remove the requirement for the `payer` account when not required for the execution of an instruction.

OS-SPT-SUG-03 | Inconsistency Between Accounts

Description

Despite the clear distinction between the two roles, several parts of the code handle the `authority` and `payer` accounts as the same entity. For instance, in the `InitializeProof` instruction, the rent payer for the `Proof` account is the `payer` account. However, later this rent returns to the `authority` account in `CloseProof`.

src/context.rs RUST

```rust
pub struct InitializeProof<'info> {
    ...
    #[account(
        init,
        ...
        payer = payer,
        ...
    )]
    pub proof: Box<Account<'info, Proof>>,
    pub authority: Signer<'info>,
    #[account(mut)]
    pub payer: Signer<'info>,
    ...
}
pub struct CloseProof<'info> {
    ...
    #[account(
        close = authority,
    )]
    pub proof: Box<Account<'info, Proof>>,
    pub authority: Signer<'info>,
    #[account(mut)]
    pub payer: Signer<'info>,
    ...
}
```

Remediation

Distinguish the `authority` and `payer` accounts as two different entities in the code.

OS-SPT-SUG-04 | Double Spending

Description

The on-chain program is unable to ascertain whether or not the sending of funds to the destination token account has already occurred. This inability is due to the absence of verification for the `commitment` account. Currently, the program relies on the fact that the off-chain program calls it only once.

Remediation

Initialize the `commitment` account to assert that the instruction may only be called once with the same set of inputs.

src/context.rs DIFF

```diff
    #[account(
+       init,
        seeds=[
            PREFIX_COMMITMENT.as_bytes(),
            pool.to_account_info().key.as_ref(),
            recent_root.as_ref(),
            transcript.as_ref(),

            // Here we're checking that the provided PDA matches what we would
            // get when making the transfer on chain.
            destination.to_account_info().key.as_ref(),
            amount.to_le_bytes().as_ref(),
        ],
        bump
    )]
-   pub commitment: UncheckedAccount<'info>,
+   pub commitment: Box<Account<'info, Commitment>>,
```

Patch

The developers are fully aware of the potential risks and will approach the instructions with the utmost caution, ensuring they proactively prevent any issues from arising.

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

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 implementation of the program 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 sum, 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 get a comprehensive understanding of 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 the other 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.