Code Inc.

Audit

Presented by:

OtterSec contact@osec.io

Nicola Vella nick0ve@osec.io

Robert Chen r@osec.io

Contents

01 Executive Summary **2**
Overview 2
Scope 2
02 Findings **3**
03 Vulnerabilities **4**
OS-CDE-ADV-00 [low] | Misleading Instruction Implementation 5
04 General Findings **6**
OS-CDE-SUG-00 [info] | Restrictive TimeLockAccount PDA 7

Appendices

A Vulnerability Rating Scale **8**
B Procedure **9**

01 | Executive Summary

Overview

Code Inc. engaged OtterSec to perform an assessment of the `timelock` program. This assessment was conducted between March 27th and March 30th, 2023. For more information on our auditing methodology, see Appendix B.

Over the course of this audit engagement, we produced 2 findings total.

Scope

The source code for this audit was provided to us in a Git repository hosted at github.com/code-wallet/code-program-library. Our analysis was conducted against commit b9194e6.

During the course of our analysis, we conducted a thorough examination of the `timelock` program's implementation.

The present program is designed specifically to enable simple state-channel mechanics. In particular, it enables two parties to stipulate an acknowledgment:

1. `vault_owner`
2. `timelock_authority`

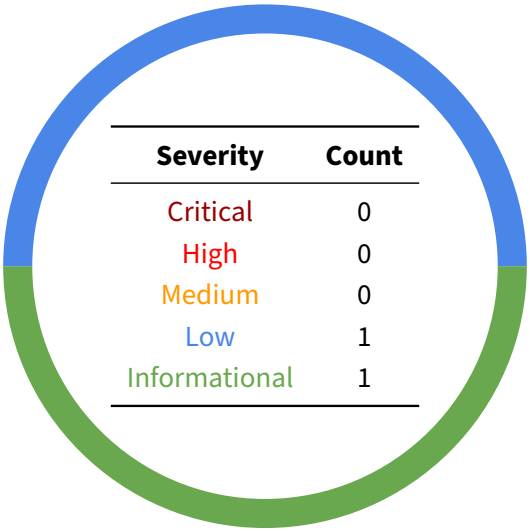
The `vault_owner` maintains the sole custodian of funds within a `Timelock` account. Specifically, the `vault_owner`'s signature is always required to move or destroy tokens. The `time_authority` gets to decide whether the funds can move immediately, otherwise the `vault_owner` will need to wait the number of days specified in the PDA of that `Timelock` account.

Given that this program was developed with an integration with the Code Wallet application in mind, a critical aspect of our security assessment involved comprehending the interactions between off-chain and on-chain programs. We would like to thank the developers for being responsive and helpful throughout the audit process.

02 | Findings

Overall, we reported 2 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 don't have an immediate impact but will help mitigate future vulnerabilities.



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

03 | 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-CDE-ADV-00	Low	Resolved	The implementation of `BurnDustWithAuthority` may be potentially misleading due to a discrepancy between the documentation and its behaviour.

OS-CDE-ADV-00 [low] | Misleading Instruction Implementation

Description

The `BurnDustWithAuthority` instruction is documented to only allow the burning of tokens in a vault if its amount is less than a specified `max_amount`. However, the implementation of the instruction may allow some tokens to be burned, even when the vault contains more tokens than the limit.

An excerpt from the implementation code for reference is shown below:

```rust
pub fn burn_dust_with_authority(ctx: Context<BurnDustWithAuthority>,
↪ _timelock_bump : u8, max_amount : u64) -> Result<()> {
    ...
    // Only allow dust transfers if the amount is less than the dust
    ↪ threshold.
    let amount = ctx.accounts.vault.amount % max_amount;
    if amount > max_amount {
        return Err(ErrorCode::InvalidDustBurn.into());
    }
    ...
}
```

This discrepancy may lead to confusion and errors among developers using the instruction.

Remediation

Review the implementation of the `BurnDustWithAuthority` instruction to clarify its intended behaviour. This may involve updating the documentation to match the actual implementation or modifying the implementation to align with the documented behaviour.

Patch

The `if` statement was removed from the implementation to align with the intended behaviour in the off-chain code that interacts with the program. Additionally, the `vault_owner` co-signs for only one of these instructions to prevent any accidental or malicious burning of excess tokens.

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 antipatterns and could lead to security issues in the future.

ID	Severity	Status	Description
OS-CDE-SUG-00	Informational	Resolved	`TimeLockAccount`'s current implementation imposes a constraint on a `vault_owner`, limiting to having up to one `timelock` with a specific `time_authority`.

OS-CDE-SUG-00 [info] | Restrictive TimeLockAccount PDA

Description

The current implementation calculates the address of a `TimeLockAccount` as follows: `PDA("timelock_state", mint, time_authority, vault_owner, num_days_locked)`.

```rust
    #[account(
        ...
        seeds=[
            TIMELOCK_STATE.as_bytes(),
            timelock.mint.as_ref(),
            timelock.time_authority.as_ref(),
            timelock.vault_owner.as_ref(),
            timelock.num_days_locked.to_le_bytes().as_ref(),
        ],
        bump = timelock_bump,
    )]
    pub timelock: Box<Account<'info, TimeLockAccount>>,
```

This design decision restricts a `vault_owner` to creating only one `timelock` with a particular `time_authority`.

Remediation

Incorporate a nonce as a seed in the address calculation process to increase the flexibility of the `TimeLockAccount` protocol. This modification enables a single `vault_owner` to create multiple `timelocks` with the same `time_authority`, without sacrificing the security or efficiency of the protocol.

By adopting this remediation, the `TimeLockAccount` protocol may become more adaptable to the diverse requirements of Solana developers and users, while maintaining its robustness and dependability.

Patch

Code Inc. implemented changes in the off-chain code that interacts with the program. Specifically, users are now able to obtain multiple `vault_owner` accounts, which effectively renders the `vault_owner` field as a nonce of the PDA. This patch improves the flexibility and usability of the `TimeLockAccount` protocol without compromising its security or efficiency.

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.