Raydium Staking

Audit

Presented by:

OtterSec contact@osec.io

Robert Chen r@osec.io

Rajvardhan Agarwal raj@osec.io

Aleksandre Khokhiashvili

khoko@osec.io

Contents

Appendices

01 | Executive Summary

Overview

Raydium engaged OtterSec to perform an assessment of the `raydium-staking` program. This assessment was conducted between May 30th and June 20th, 2022. For more information on our auditing methodology, see Appendix C.

Key Findings

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

In particular, we reported an issue involving missing vault account checks that may lead to a loss of funds (OS-RDM-ADV-00).

Additionally, we provided recommendations around redundant code (OS-RDM-SUG-00) and unused instructions (OS-RDM-SUG-02) to increase code readability and reduce the attack surface of the program.

02 | Scope

The source code was delivered to us in a git repository at github.com/raydium-io/raydium-staking. This audit was performed against commit `76a2744`.

A brief description of the programs is as follows.

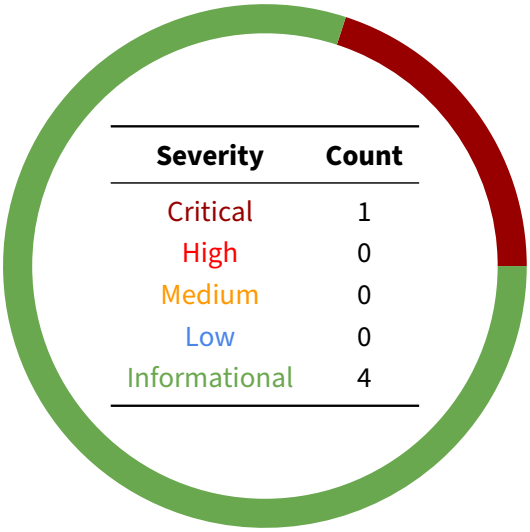
Name	Description
`staking`	The `staking` program allows users to stake their tokens in order to earn additional reward tokens and has the following functionalities: • Depositing tokens to the staker account. • Withdraw earned reward tokens. • Withdrawing staked tokens.
`farm`	The `farm` program allows users to stake their liquidity pool tokens to possibly earn two different reward tokens. It has the following functionalities: • Depositing liquidity pool tokens to the farm pool. • Withdrawing two types of earned reward tokens. • Withdrawing staked liquidity pool tokens.

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 don't have an immediate impact but will help mitigate future vulnerabilities.

The below chart displays the findings by severity.



Severity	Count
Critical	1
High	0
Medium	0
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 D.

ID	Severity	Status	Description
OS-RDM-ADV-00	Critical	Resolved	Missing validation on the reward vault account may allow an attacker to steal liquidity provider tokens.

OS-RDM-ADV-00 [crit] | Missing TokenAccount Checks

Description

In the `WithdrawV2` instruction, the program does not validate the `vault_reward_token_b_info` account. This may allow an attacker to withdraw tokens from any `TokenAccount` owned by the pool authority. In this case, the attacker would be able to withdraw from either `reward_vault_a` or the liquidity provider vault.

This is extremely dangerous as tokens from the liquidity provider vault could potentially be worth much greater than the tokens from `reward_vault_b`. The attacker could exploit this vulnerability to withdraw all liquidity provider tokens from the pool.

The affected code can be found in the code snippet below, wherein it can be seen that no checks are performed on `vault_reward_token_b_info`.

doubleReward/src/processor.rs RUST

```
let dest_reward_token_b_info = next_account_info(account_info_iter)?;
let vault_reward_token_b_info = next_account_info(account_info_iter)?;

[...]

if pending_b > 0 {
    Self::token_transfer_with_authority(
        stake_pool_info.key,
        token_program_info.clone(),
        vault_reward_token_b_info.clone(),
        dest_reward_token_b_info.clone(),
        authority_info.clone(),
        stake_pool.nonce as u8,
        pending_b
    )?;
}
```

Real world impact

The numbers below are borrowed from https://raydium.io/farms/ (accessed on June 21, 2022). Using a small off-chain program, OtterSec iterated through every listed `StakePool` and calculated the possible profits, in LP tokens, for an attacker. Listed below are the most profitable and fastest target farms for an attacker.

An attacker can multiply their deposited amount by 20 times in a single day using the BTC - stSOL farm. This farm has an estimated total value of $671,327. It is believed that a majority of these LP tokens were vulnerable. In this scenario, the attacker's actions are indistinguishable from a normal user right up until the deposit/withdraw instruction is invoked using the wrong (crafted) vault accounts.

In some cases, the attacker would be able to approximately double their deposited LP tokens by waiting for a day's worth of rewards to accumulate:

- mSOL - USDT estimated total value: $315,131.
- ETH - stSOL estimated total value: $707,470.

The same strategy could be used to steal LP tokens from another farm; stealing from other farms would likely take longer. Additionally, there's a risk of the attacker abusing this vulnerability to withdraw reward tokens from vault A instead of vault B.

Proof of Concept

1. Initialize the farm with two reward tokens. We'll call them A and B tokens.
    - `reward_per_slot_a` is set to 10,000.
    - `reward_per_slot_b` is set to 10,000.
2. Victim deposits 50,000 liquidity pool(LP) tokens to the farm.
3. Attacker deposits 10 LP tokens to the farm.
4. Attacker waits for 9000 slots to pass, which is roughly equivalent to one hour.
5. Attacker calls the `WithdrawV2` instruction:
    - Amount of LPs to withdraw is set to 10.
    - `vault_reward_token_b` is set to the farm's LP token account.
    - `dest_reward_token_b` is set to the attacker's LP token account.
6. The program calculates the size of the B token reward to be 18,000.
7. Attacker receives 18,000 LP tokens instead of 18,000 B tokens.

Remediation

The provided in `vault_reward_token_b_info` account must be checked against `stake_pool.reward_vault_b`. Moreover, it is recommended to validate the underlying mint for all `TokenAccounts`.

Patch

Fixed in a2a2b52 as the account is now checked against `stake_pool.reward_vault_b`.

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 do represent antipatterns and could introduce a vulnerability in the future.

ID	Description
OS-RDM-SUG-00	Redundant code is used in multiple instructions.
OS-RDM-SUG-01	The integer states used for accounts are ambiguous.
OS-RDM-SUG-02	There are several instructions present which serve no purpose.
OS-RDM-SUG-03	The usage of the reward B token account is inconsistent between different instructions.

OS-RDM-SUG-00 | Code Redundancy

Description

It is observed that both staking and farm programs contain a lot of redundant code. For example, the `process_deposit_v2` and `process_withdraw_v2` instructions use the same block of code for merging `StakerInfo` accounts.

However, the block present in `process_deposit_v2` is missing a line of code which could potentially lead to a bad bounds bug. This can be seen in the following code snippets.

doubleReward/src/processor.rs RUST

```
577             while remain_accounts > 0 {
578                 let staker_info_v1 = next_account_info(account_info_iter)?;
579 
580                 if *staker_info_v1.owner == Pubkey::default() {
581                     continue;
582                 }
```

doubleReward/src/processor.rs RUST

```
886             while remain_accounts > 0 {
887                 let staker_info_v1 = next_account_info(account_info_iter)?;
888 
889                 if *staker_info_v1.owner == Pubkey::default() {
890                     remain_accounts = remain_accounts - 1;
891                     continue;
892                 }
```

It is not recommended to use redundant code as it makes the program harder to review for developers and could cause the code to be error-prone during updates. This could also increase the size of the program executable.

Moreover, improving code decomposition for functions in `process_deposit_v2` and `process_withdraw_v2` could have possibly prevented OS-RDM-ADV-00, since `vault_reward_token_b_info` was properly checked inside one function but not the other.

Remediation

Replace redundant code with utility functions.

OS-RDM-SUG-01 | Ambiguous States

Description

It is observed that many of the accounts used integers to track their current state. We recommend following idiomatic rust rules and using enums for any kind of state machine.

This makes all the possible states explicit and, in combination with `#![deny(missing_docs)]`, also forces each state to be documented. All of this makes it much easier for developers to reason about the program's behavior.

We also observed following behaviors in different instructions when `stake_pool.state == 1`:

- `process_deposit` returns an error if `stake_pool.state != 1`.
- `process_deposit_v2` returns an error if `stake_pool.state != 1`.
- `process_withdraw` withdraws entire user's deposit if `stake_pool.state != 1`. This happens even if the user requests to withdraw only part of it.
- `process_withdraw_v2` only checks whether `stake_pool.state == 0`.

Without documentation for different possible states, it is unclear why v1 and v2 deposits behave the same but withdraw instructions do not. It is recommended that the Raydium team reviews the instructions to ensure that the difference is intended.

Remediation

Replace all integer `state` fields using custom Rust `enum` types. It is recommended that `#[repr(u64)]` be used to ensure that all existing accounts remain compatible with the updated code.

The code snippet below contains an example `enum` type for tracking `StakePool`'s state.

program/src/state.rs RUST

```rust
/// Tracks current state for StakerPool
#[repr(u64)]
#[derive(Clone, Copy, Debug, PartialEq)]
pub enum StakePoolState {
    /// Indicates that account has not been initialized
    Uninitialized = 0,
    /// Indicates that account is open to deposits
    Reward = 1,
    /// Indicates that account is initialized
    Initialized = 2,
}

impl Default for StakerPoolState {
    fn default() -> Self {
```

```
            return Self::Uninitialized;
        }
    }
```

OS-RDM-SUG-02 | Purposeless Instructions

Description

It was observed that several fields and instructions were effectively unused inside the programs. `process_set_fee`, `_owner`, and `process_set_fee` inside `/program` are not needed since the staking program has no fees for any kind of interaction.

program/src/processor.rs RUST

```
    pub fn process_set_fee(
        program_id: &Pubkey,
        accounts: &[AccountInfo],
        fee: Fee,
    ) -> ProgramResult {
        [...]

        if stake_pool.owner != *owner_info.key {
            return Err(StakePoolError::InvalidSignAccount.into());
        }

        stake_pool.fee = fee;
        Ok(())
    }
```

Removing these instructions would reduce the attack surface of the program, making it easier to review and extend.

Remediation

Remove instructions that serve no purpose inside the program. Ideally, these fields should not be removed from the relevant structures, since this could break compatibility with existing accounts.

The `staking` program could remove these instructions in a similar manner to the `farm` program:

doubleReward/src/processor.rs RUST

```
            StakePoolInstruction::SetFeeOwner => {
                return Err(StakePoolError::UnImplement.into());
            }
            StakePoolInstruction::SetFee => {
                return Err(StakePoolError::UnImplement.into());
            }
```

OS-RDM-SUG-03 | Optional Reward Vault Inconsistency

Description

It is observed that in the `farm` program, `vault_reward_token_b_info` is required inconsistently inside different instructions.

- `process_initialize` requires `reward_vault_b_info` and always sets `stake_poll.reward_vault_b`.
- `vault_reward_token_b_info` is optional in the `process_deposit` and `process_withdraw` instructions.
- `vault_reward_token_b_info` is required in the `process_deposit_v2` and `process_withdraw_v2` instructions.

This is evident in the code snippets shown below.

doubleRewards/src/processor.rs RUST

```
    pub fn process_withdraw(
        program_id: &Pubkey,
        accounts: &[AccountInfo],
        amount: u64,
    ) -> ProgramResult {
        [...]
        let mut dest_reward_token_b_info = None;
        let mut vault_reward_token_b_info = None;
        if accounts.len() == 12 {
            dest_reward_token_b_info = Some(next_account_info(account_info_iter)?);
            vault_reward_token_b_info =
    ↪   Some(next_account_info(account_info_iter)?);
        }
```

doubleRewards/src/processor.rs RUST

```
    pub fn process_withdraw_v2(
        program_id: &Pubkey,
        accounts: &[AccountInfo],
        amount: u64,
    ) -> ProgramResult {
        [...]
        let dest_reward_token_b_info = next_account_info(account_info_iter)?;
        let vault_reward_token_b_info = next_account_info(account_info_iter)?;
```

Remediation

Document when `stake_poll.reward_vault_b` can be optional if this behaviour is intended.

A | Program Files

Below are the files in scope for this audit and their corresponding SHA256 hashes.

```
doubleReward
  Cargo.lock                                    8f9ad4c88f3596655304c93d171a6086
  Cargo.toml                                    d2e5ae6f8e69e4a309a320bae86d9d60
  Xargo.toml                                    815f2dfb6197712a703a8e1f75b03c69
  src
    entrypoint.rs                               a50450d44cb783c1dfefb93bbb5c73b2
    error.rs                                    d0b6a5b31e49b248e3b99e72046f735b
    instruction.rs                              6ee3305e2d39885f83bf0ec6310ad35a
    lib.rs                                      fae970960014a6973718cfad159600c8
    processor.rs                                3ed78c38b3c0bb413c727089b0256c94
    state.rs                                    5e97242dc8ad9db28091cf05e0e00988
program
  Cargo.lock                                    4a8451d39567c0cfec38f537a88322fb
  Cargo.toml                                    9e4a448f820e438a5cf55d85d966d87d
  Xargo.toml                                    815f2dfb6197712a703a8e1f75b03c69
  src
    entrypoint.rs                               a50450d44cb783c1dfefb93bbb5c73b2
    error.rs                                    edd04212b01af36956e3cd01bb697f11
    instruction.rs                              69c031e2081107e1ec4ef4aff6332ac3
    lib.rs                                      fae970960014a6973718cfad159600c8
    processor.rs                                2d667fb5e1c2574c0fd37c4c7ceb2c1e
    state.rs                                    9ac9273e44e834fad63b8a10547270cb
```

B | Implementation Security Checklist

Unsafe arithmetic

Integer underflows or overflows	Unconstrained input sizes could lead to integer over or underflows, causing potentially unexpected behavior. Ensure that for unchecked arithmetic, all integers are properly bounded.
Rounding	Rounding should always be done against the user to avoid potentially exploitable off-by-one vulnerabilities.
Conversions	Rust as conversions can cause truncation if the source value does not fit into the destination type. While this is not undefined behavior, such truncation could still lead to unexpected behavior by the program.

Account security

Account Ownership	Account ownership should be properly checked to avoid type confusion attacks. For Anchor, the safety of unchecked accounts should be clearly justified and immediately obvious.
Accounts	For non-Anchor programs, the type of the account should be explicitly validated to avoid type confusion attacks.
Signer Checks	Privileged operations should ensure that the operation is signed by the correct accounts.
PDA Seeds	PDA seeds are uniquely chosen to differentiate between different object classes, avoiding collision.

Input validation

Timestamps	Timestamp inputs should be properly validated against the current clock time. Timestamps which are meant to be in the future should be explicitly validated so.
Numbers	Sane limits should be put on numerical input data to mitigate the risk of unexpected over and underflows. Input data should be constrained to the smallest size type possible, and upcasted for unchecked arithmetic.
Strings	Strings should have sane size restrictions to prevent denial of service conditions
Internal State	If there is internal state, ensure that there is explicit validation on the input account's state before engaging in any state transitions. For example, only open accounts should be eligible for closing.

Miscellaneous

Libraries	Out of date libraries should not include any publicly disclosed vulnerabilities
Clippy	cargo clippy is an effective linter to detect potential anti-patterns.

C | 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 onchain program. In other words, there is no way to steal tokens or deny service, ignoring any Solana specific quirks such as account ownership issues. An example of a design vulnerability would be an onchain oracle which could be manipulated by flash loans or large deposits.

On the other hand, auditing the implementation of the program requires a deep understanding of Solana's execution model. Some common implementation vulnerabilities include account ownership issues, arithmetic overflows, and rounding bugs. For a non-exhaustive list of security issues we check for, see Appendix B.

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 any target in a team of two. 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.

D | 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