OtterSec

ORE v2

Security Assessment

July 9th, 2024 — Prepared by OtterSec

Robert Chen r@osec.io

Table of Contents

Executive Summary **2**

Overview 2

Key Findings 2

Scope **3**

Findings **4**

Vulnerabilities **5**

OS-ORE-ADV-00 | Missing Associated Token Account Check 6

OS-ORE-ADV-01 | Enhancing Mining Activity Tracking 7

General Findings **8**

OS-ORE-SUG-00 | Unsafe Account Deserialization 9

OS-ORE-SUG-01 | Code Redundancy 10

OS-ORE-SUG-02 | Excessive Utilization Of Saturated Math 11

OS-ORE-SUG-03 | Code Maturity 12

Appendices

Vulnerability Rating Scale **13**

Procedure **14**

01 — Executive Summary

Overview

Regolith Labs engaged OtterSec to assess the `ore` program. This assessment was conducted between June 27th and July 2nd, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 6 findings throughout this audit engagement.

In particular, we identified a high-risk vulnerability where the associated token account of the treasury is not validated to ensure it is the expected one, which may allow tokens to be misdirected (OS-ORE-ADV-00). Additionally, the tolerance setting may be ineffective due to its current configuration (OS-ORE-ADV-01).

We also made recommendations to limit the utilization of saturating math to check for overflows (OS-ORE-SUG-02) and suggested the need to ensure adherence to coding best practices for better maintainability (OS-ORE-SUG-03). Furthermore, while loading the mint and token account, we advised performing checks during the deserialization of the account data (OS-ORE-SUG-00).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/regolith-labs/ore-private. This audit was performed against commit 77e1a7b.

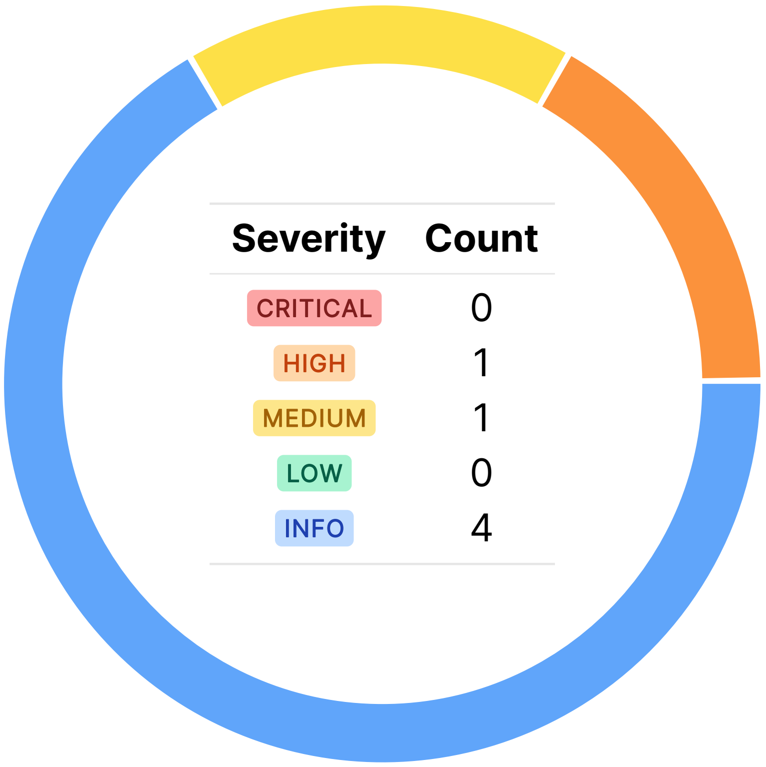
A brief description of the programs is as follows:

Name	Description
ore	A digital currency that is easy to mine, utilizing a novel proof-of-work algorithm to ensure that no miner is ever starved of earning rewards.

03 — Findings

Overall, we reported 6 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	0
HIGH	1
MEDIUM	1
LOW	0
INFO	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-ORE-ADV-00	HIGH	RESOLVED ✓	`process_stake` does not verify whether the `treasury_tokens_info` is the expected associated token account (ATA), which may allow tokens to be misdirected.
OS-ORE-ADV-01	MEDIUM	RESOLVED ✓	`TOLERANCE` setting may be ineffective due to its current configuration. Instead, the `last_hash_at` field may be adjusted to ensure that it reflects a meaningful timestamp.

Missing Associated Token Account Check HIGH OS-ORE-ADV-00

Description

`stake::process_stake` loads various accounts, including the `treasury_tokens_info` account, which is intended to be the associated token account of the treasury. However, without a strict equality check, there is no assurance that the provided `treasury_tokens_info` account is indeed the correct associated token account. Thus, an attacker may provide a different token account for `treasury_tokens_info`, thereby redirecting the staked tokens to an account they control instead of the intended treasury account.

>_ *src/processor/stake.rs* rust

```rust
pub fn process_stake<'a, 'info>(
    _program_id: &Pubkey,
    accounts: &'a [AccountInfo<'info>],
    data: &[u8],
) -> ProgramResult {
    // Parse args
    let args = StakeArgs::try_from_bytes(data)?;
    let amount = u64::from_le_bytes(args.amount);

    // Load accounts
    let [signer, proof_info, sender_info, treasury_tokens_info, token_program] = accounts else {
        return Err(ProgramError::NotEnoughAccountKeys);
    };
    [...]
}
```

Remediation

Implement a strict equality check to ensure that `treasury_tokens_info` matches the expected associated token account for the treasury. Similarly, do this in the `claim` and `rest` instructions.

Patch

Resolved in 3b1039b.

Enhancing Mining Activity Tracking MEDIUM OS-ORE-ADV-01

Description

`TOLERANCE` is the threshold for what the system considers acceptable levels of spam or delays. Adjusting this tolerance may inadvertently decrease the effective spam duration. This reduction could potentially lower the overall reliability of the system against spam attacks. Instead of directly manipulating `TOLERANCE` , consider updating the `last_hash_at` with logic that ensures a minimum interval between hash submissions or other critical actions. This will mitigate spamming or rapid submissions by ensuring that `last_hash_at` only reflects significant changes in miner activity rather than every single hash submission.

Remediation

Modify the logic that updates `last_hash_at`. Instead of setting it to the current Unix timestamp (`clock.unix_timestamp`), update it to the maximum value between `prev_last_hash_at + ONE_MINUTE` and `clock.unix_timestamp` . This introduces a minimum interval (`ONE_MINUTE`) between consecutive hash submissions that can update `last_hash_at`.

Patch

Resolved in 3e91505.

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-ORE-SUG-00	Utilizing `unpack_unchecked` for data deserialization in `load_mint` and `load_token_account` may result in potential security risks because it does not validate the account data length.
OS-ORE-SUG-01	The `Treasury` account is currently unutilized and serves only as a signing authority, rendering its initialization unnecessary.
OS-ORE-SUG-02	Recommendations to limit the utilization of saturating math to check for overflows.
OS-ORE-SUG-03	Suggestion concerning an inconsistency in the codebase and ensuring adherence to coding best practices.

Unsafe Account Deserialization

OS-ORE-SUG-00

Description

In `loader`, `load_mint` and `load_token_account` (as shown below) currently utilize `unpack` to deserialize the account data. `unpack_unchecked` does not perform any additional checks beyond the deserialization process itself. Thus, if the data structure is corrupted or malformed, utilizing `unpack_unchecked` may result in undefined behavior or runtime errors because it assumes that the data being deserialized is valid.

>_ src/loaders.rs rust

```rust
pub fn load_token_account<'a, 'info>(
    info: &'a AccountInfo<'info>,
    owner: Option<&Pubkey>,
    mint: &Pubkey,
    is_writable: bool,
) -> Result<(), ProgramError> {
    [...]
    let account_data = info.data.borrow();
    let account = spl_token::state::Account::unpack_unchecked(&account_data)
        .or(Err(ProgramError::InvalidAccountData))?;
    [...]
}
```

Remediation

Utilize `unpack` instead of `unpack_unchecked` within `load_mint` and `load_token_account`, as it deserializes the account data and performs additional integrity checks to ensure that the data conforms to the expected format and constraints.

Patch

Resolved in 2cbcaab.

Code Redundancy

OS-ORE-SUG-01

Description

The `Treasury` account is currently empty and serves only as a signing authority. Since it does not have any other function beyond authorization, it does not need to be initialized and can be removed as it is unnecessary.

>_ *src/consts.rs* rust

```rust
/// The address of the treasury account.
pub const TREASURY_ADDRESS: Pubkey =
    Pubkey::new_from_array(ed25519::derive_program_address(&[TREASURY], &PROGRAM_ID).0);
```

Remediation

Instead of managing an initialized `Treasury` account, directly compare against `TREASURY_ADDRESS`.

Excessive Utilization Of Saturated Math OS-ORE-SUG-02

Description

Currently, within the codebase, saturating math is utilized in multiple places to prevent potential overflows, specifically through the use of `saturating_mul`. When declaring constants, `saturating_mul` is employed to set their values. Since constants are computed statically, the compiler verifies at compile time whether the computations will result in overflow. In scenarios where values are known and fixed, there is typically no risk of overflow because these values are designed to fit within safe bounds. Even if an overflow occurs, it is preferable to abort rather than end up with unexpected saturation.

>_ *src/consts.rs* rust

```rust
/// The duration of an Ore epoch, in seconds.
pub const EPOCH_DURATION: i64 = ONE_MINUTE.saturating_mul(EPOCH_MINUTES);

/// The maximum token supply (21 million).
pub const MAX_SUPPLY: u64 = ONE_ORE.saturating_mul(21_000_000);

/// The target quantity of ORE to be mined per epoch.
pub const TARGET_EPOCH_REWARDS: u64 = ONE_ORE.saturating_mul(EPOCH_MINUTES as u64);

/// The maximum quantity of ORE that can be mined per epoch.
/// Inflation rate ≈ 1 ORE / min (min 0, max 5)
pub const MAX_EPOCH_REWARDS: u64 = TARGET_EPOCH_REWARDS.saturating_mul(5);
```

Furthermore, it would be appropriate to change the type from `u64` to `u128` to prevent potential overflows, instead of relying on `saturating_mul`. For instance, `total_rewards`, currently defined as `u64`, could benefit from this adjustment.

Remediation

Apply the above optimizations for increased efficiency in the code base.

Patch

Resolved in 3e91505.

Code Maturity OS-ORE-SUG-03

Description

In `loader::load_system_account`, it may be appropriate to relax the constraints on the `miner` account ownership, changing from being strictly a system account to potentially being owned by the protocol. However, it should be noted that it should not be a Program Derived Addresses (PDAs) due to checks in mine.

Remediation

Implement the above suggestion.

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 the 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.