Solana M

Security Assessment

April 18th, 2025 — Prepared by OtterSec

Akash Gurugunti sud0u53r.ak@osec.io

Robert Chen r@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

M0 Foundation engaged OtterSec to assess the `solana-m` program. This assessment was conducted between March 10th and March 17th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 5 findings throughout this audit engagement.

In particular, we identified a vulnerability where the mint account is not reloaded after minting tokens, potentially updating the max supply based on outdated data, resulting in incorrect tracking of the total minted supply (OS-SMZ-ADV-01). Additionally, unclaimed rewards may accumulate as excess M tokens in the vault, resulting in inaccurate collateral accounting (OS-SMZ-ADV-00).

We made recommendations regarding modifications to the codebase to ensure adherence to coding best practices (OS-SMZ-SUG-01), and suggested removing redundant and unutilized code instances (OS-SMZ-SUG-02. Moreover, we advised including additional safety checks within the codebase to improve security (OS-SMZ-SUG-00).

Scope

The source code was delivered to us in a Git repository at https://github.com/m0-foundation/solana-m. This audit was performed against commits c299921 and d44cbd8.

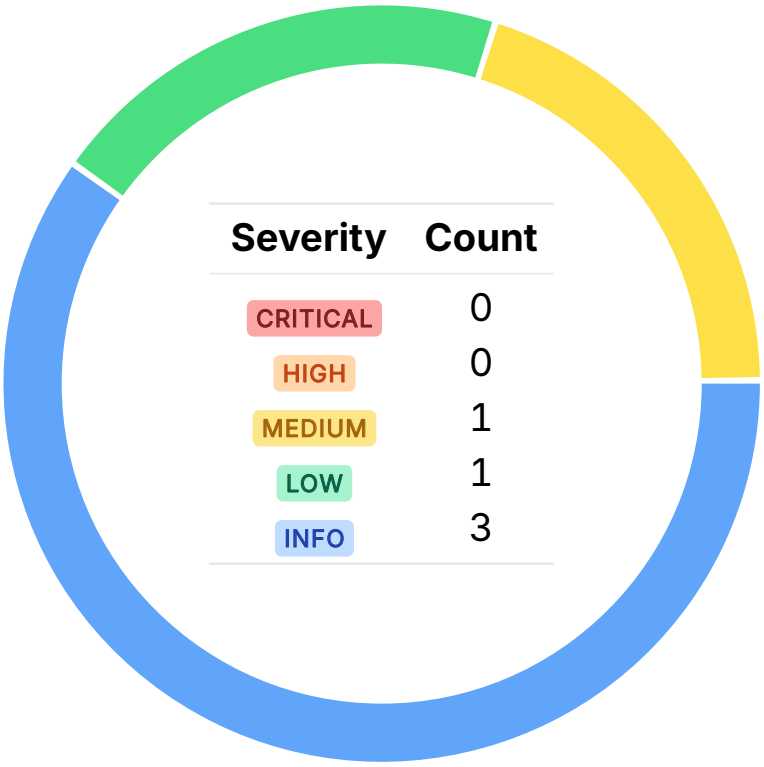
A brief description of the program is as follows:

Name	Description
earn	A program for yield distribution logic and earner management for the M token.
ext-earn	A program to handle wrapping/unwrapping M to wM, as well as yield distribution and earner manager for the wM token.

02 — 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 aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	0
MEDIUM	1
LOW	1
INFO	3

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-SMZ-ADV-00	MEDIUM	RESOLVED	Unclaimed rewards may be accumulated as excess `M` tokens in the vault, resulting in inaccurate collateral accounting.
OS-SMZ-ADV-01	LOW	RESOLVED	The `mint` account is not reloaded after minting tokens, potentially updating the `max_supply` based on outdated data, resulting in incorrect tracking of the max minted supply.

Improper Handling of Excess M Tokens MEDIUM OS-SMZ-ADV-00

Description

`vault_m_token_account` will accumulate `M` tokens from rewards. If some wrapped M tokens did not utilize `ClaimFor` to earn rewards, then the rewards for those wrapped M may remain in the vault. Since the protocol only mints `ext_mint` tokens for locked M tokens during wrapping, this surplus `M` stays idle and inaccessible. The system appears overcollateralized because there is more `M` in the vault than needed to back existing `ext_mint` supply.

Remediation

Implement logic to properly handle excess rewards in `vault_m_token_account`.

Patch

Acknowledged by the developers.

Inaccurate Mint Supply Tracking LOW OS-SMZ-ADV-01

Description

In the `ClaimFor` instruction, the function performs a CPI to mint tokens via `mint_tokens(...)`, and immediately afterward, it reads from `ctx.accounts.mint.supply` to update `global_account.max_supply`. However, after tokens are minted, the in-memory representation of `ctx.accounts.mint` does not reflect the increased supply. `ctx.accounts.mint` still holds the old state of the mint account. Consequently, `max_supply` may not reflect the most recent mint, resulting in inaccurate values.

>_ *earn/src/instructions/earn_authority/claim_for.rs* RUST

```rust
pub fn handler(ctx: Context<ClaimFor>, snapshot_balance: u64) -> Result<()> {
    [...]
    // Mint the tokens to the user's token aaccount
    // The result of the CPI is the result of the handler
    mint_tokens(
        &ctx.accounts.user_token_account,      // to
        &rewards,                              // amount
        &ctx.accounts.mint,                    // mint
        &ctx.accounts.mint_multisig,           // multisig mint authority
        &ctx.accounts.token_authority_account, // signer
        token_authority_seeds,                 // signer seeds
        &ctx.accounts.token_program,           // token program
    )?;

    // Check the current supply of M against the max supply in the global account
    // If it is greater, update the max supply
    // This check is also done when an index is propagated for a bridge
    // These are the only two actions that can mint M on Solana
    // Therefore, we always have an accurate max supply for calculating max yield
    if ctx.accounts.mint.supply > ctx.accounts.global_account.max_supply {
        ctx.accounts.global_account.max_supply = ctx.accounts.mint.supply;
    }
    [...]
}
```

Remediation

Ensure the mint account is reloaded after the `mint_tokens` call and before accessing its supply to update `max_supply`.

Patch

Fixed in PR#62.

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 result in security issues in the future.

ID	Description
OS-SMZ-SUG-00	There are several instances where proper validation is not done, resulting in potential security issues.
OS-SMZ-SUG-01	Suggestions regarding ensuring adherence to coding best practices.
OS-SMZ-SUG-02	The codebase contains multiple cases of redundant and unutilized code that should be removed for better maintainability and clarity.

Missing Validation Logic

OS-SMZ-SUG-00

Description

1. The `ext_earn::Initialize` instruction sets up the core state (`ExtGlobal`) for the extended earn program. This state links the `m_mint` (original reward token mint) and `m_earn_global_account`. However, `Initialize`, should verify that the `m_mint` account matches `m_earn_global_account.mint`, as currently there is no explicit validation to ensure the correct mint is utilized.
2. The `ext_earn::Initialize` instruction currently accepts an external `ext_mint` without validating its configuration. This poses a security risk, as a malicious or misconfigured mint may be passed. To prevent this, either initialize `ext_mint` within the instruction or verify that it has zero supply, the correct PDA as mint authority, no freeze authority.

Remediation

Incorporate the above validations into the codebase.

Code Maturity OS-SMZ-SUG-01

Description

1. Add an admin functionality to update `Global.claim_cooldown` to allow flexibility in adjusting the claim cooldown period, ensuring adaptability to changing yield distribution needs.
2. In `AddRegistrarEarner::handler` (shown below), the `last_claim_timestamp` for a newly initialized `Earner` account is set to the current Unix timestamp at the time of instruction execution. However, it will be more appropriate to utilize `global_account.timestamp` to align with the index recording time. The same adjustment should be made in `ext_earn::AddEarner`, setting `last_claim_timestamp` to `global_account.timestamp` for consistency.

>_ earn/src/instructions/open/add_registrar_earner.rs RUST

```rust
pub fn handler(
    ctx: Context<AddRegistrarEarner>,
    user: Pubkey,
    proof: Vec<ProofElement>,
) -> Result<()> {
    [...]
    ctx.accounts.earner_account.set_inner(Earner {
        last_claim_index: ctx.accounts.global_account.index,
        last_claim_timestamp: Clock::get()?.unix_timestamp.try_into().unwrap(),
        bump: ctx.bumps.earner_account,
        user,
        user_token_account: ctx.accounts.user_token_account.key(),
    });

    Ok(())
}
```

3. Currently, the removal check utilizes `earner_account.user` to verify if the user is excluded from the Merkle root. However, this prevents the removal of earners as long as the original user remains in the Merkle root, regardless of changes in token account ownership. This behavior contradicts the intended design, which is to remove earners whose current token account owners are no longer included in the Merkle root. Utilize `earner_account.user_token_account.owner` aligning removal logic with the current token account ownership and ensuring only approved owners remain earners.

Remediation

Implement the above-mentioned suggestions.

Unutilized/Redundant Code

OS-SMZ-SUG-02

Description

1. It will be more efficient to assign `ctx.accounts.global_account` to a variable and re-utilize it to reduce code redundancy by eliminating repeated calls to `ctx.accounts.global_account` in the `PropagateIndex` handler within `earn`.

>_ *earn/src/instructions/portal/propagate_index.rs* RUST

```rust
pub fn handler(
    ctx: Context<PropagateIndex>,
    new_index: u64,
    earner_merkle_root: [u8; 32],
    earn_manager_merkle_root: [u8; 32],
) -> Result<()> {
    [...]
    if new_index >= ctx.accounts.global_account.index {
        if earner_merkle_root != [0u8; 32] {
            ctx.accounts.global_account.earner_merkle_root = earner_merkle_root;
        }
        if earn_manager_merkle_root != [0u8; 32] {
            ctx.accounts.global_account.earn_manager_merkle_root =
                ↪  earn_manager_merkle_root;
        }
    }
    [...]
    let cooldown_target =
        ctx.accounts.global_account.timestamp +
            ↪  ctx.accounts.global_account.claim_cooldown;

    if !ctx.accounts.global_account.claim_complete
        || current_timestamp < cooldown_target
        || new_index <= ctx.accounts.global_account.index
    {
        if current_supply > ctx.accounts.global_account.max_supply {
            ctx.accounts.global_account.max_supply = current_supply;
        }

        return Ok(());
    }
    [...]
}
```

2. Storing `portal_authority` is unnecessary, as it may be derived when required, such as in `extPropagateIndex`.

3. `SetClaimCooldown` and `SetEarnAuthority` instructions duplicate identical account validation logic for admin access and global account mutation. Extracting this shared logic into a reusable context will improve code clarity and maintainability.

Remediation

Remove the redundant and unutilized code instances highlighted above.

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 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.