M0 Solana Extensions

Security Assessment

July 12th, 2025 — Prepared by OtterSec

Ajay Shankar Kunapareddy d1r3wolf@osec.io

Xiang Yin soreatu@osec.io

Robert Chen r@osec.io

Table of Contents

01 — Executive Summary

Overview

M0-foundation engaged OtterSec to assess the `solana-extensions` program. This assessment was conducted between June 16th and June 20th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 4 findings throughout this audit engagement.

In particular, we identified a critical vulnerability where the whitelist instructions in the Ext swap program lack an admin signer check, allowing any user to add or remove whitelisted extensions or unwrappers (OS-MSE-ADV-00). Additionally, we highlighted an instance of incorrect token program validation, which may result in valid swaps to fail (OS-MSE-ADV-01).

We also made recommendations to ensure adherence to coding best practices (OS-MSE-SUG-01), and advised enforcing a whitelist check on the Token2022 extensions of the mint program to restrict unsupported extensions that may result in certain instructions behaving in unintended ways (OS-MSE-SUG-00).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/m0-foundation/solana-extensions. This audit was performed against PR#11 and PR#12.

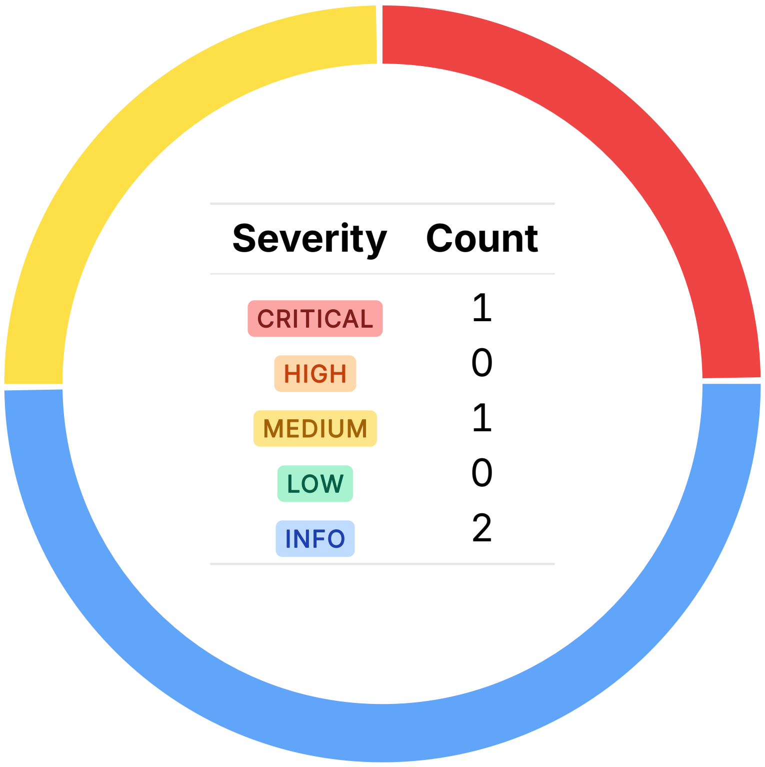
A brief description of the programs is as follows:

Name	Description
solana-extensions	Enables the creation of stablecoins backed by the M token with customizable yield behavior. The `m_ext` program defines extension tokens that either earn no yield or accrue yield via rebalancing. The `ext_swap` program allows seamless conversion between these tokens without passing through M.

03 — Findings

Overall, we reported 4 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	1
HIGH	0
MEDIUM	1
LOW	0
INFO	2

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-MSE-ADV-00	CRITICAL	RESOLVED ✓	The whitelist instructions in `ext_swap` lack an admin signer check, allowing any user to add or remove whitelisted extensions or unwrappers.
OS-MSE-ADV-01	MEDIUM	RESOLVED ✓	The `from_token_account` is incorrectly validated against `to_token_program` instead of `from_token_program`, which may result in valid swaps to fail.

Missing Admin Authorization Check CRITICAL OS-MSE-ADV-00

Description

The four whitelist-related instructions in `ext_swap` (`whitelist_extension`, `remove_whitelisted_extension`, `whitelist_unwrapper`, and `remove_whitelisted_unwrapper`) lack a critical authorization check. Specifically, they do not enforce that the signer is the admin specified in the `swap_global` account. Without a `has_one = admin` constraint, any user may call these instructions to arbitrarily modify whitelisted programs or unwrapper authorities. This allows malicious actors to whitelist rogue extensions, remove legitimate ones, or give themselves privileged unwrap permissions.

>_ *programs/ext_swap/src/lib.rs* RUST

```rust
pub fn whitelist_extension<'info>(
    ctx: Context<WhitelistExt>,
    ext_program: Pubkey,
) -> Result<()> {
    WhitelistExt::handler(ctx, ext_program)
}

pub fn remove_whitelisted_extension<'info>(
    ctx: Context<RemoveWhitelistedExt>,
    ext_program: Pubkey,
) -> Result<()> {
    RemoveWhitelistedExt::handler(ctx, ext_program)
}

pub fn whitelist_unwrapper<'info>(
    ctx: Context<WhitelistUnwrapper>,
    authority: Pubkey,
) -> Result<()> {
    WhitelistUnwrapper::handler(ctx, authority)
}

pub fn remove_whitelisted_unwrapper<'info>(
    ctx: Context<RemoveWhitelistedUnwrapper>,
    authority: Pubkey,
) -> Result<()> {
    RemoveWhitelistedUnwrapper::handler(ctx, authority)
}
```

Remediation

Add a `has_one = admin` constraint on the `swap_global` account within the 4 whitelist instructions to ensure only the designated admin may perform these operations.

Patch

Resolved in PR#20.

Incorrect Token Program Validation MEDIUM OS-MSE-ADV-01

Description

In the `WhitelistExt` instruction within `ext_swap`, the `from_token_account` is incorrectly constrained to utilize the `to_token_program` instead of the correct `from_token_program`. This creates a mismatch during validation, when the `from_token_program` and `to_token_program` differ. As a result, valid swap attempts will be rejected, resulting in a denial-of-service for legitimate users.

>_ *programs/ext_swap/src/instructions/swap.rs* RUST

```rust
#[derive(Accounts)]
pub struct Swap<'info> {
    [...]
    #[account(
        mut,
        token::mint = from_mint,
        token::token_program = to_token_program,
    )]
    pub from_token_account: Box<InterfaceAccount<'info, TokenAccount>>,
    [...]
}
```

Remediation

Update the constraint to utilize `token::token_program = from_token_program`. This change ensures that the source token account is properly validated.

Patch

Resolved in PR#21.

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-MSE-SUG-00	`ext_mint` may include unsupported extensions that may result in certain instructions behaving in unintended ways.
OS-MSE-SUG-01	Suggestions regarding inconsistencies in the codebase and ensuring adherence to coding best practices.

Unsupported Mint Extensions OS-MSE-SUG-00

Description

In `m_ext::initialize` within `validate`, the `ext_mint` may contain unsupported extensions such as `TransferHook`, which require special handling. Without this handling, some instructions may fail or behave unexpectedly.

>_ *programs/m_ext/src/instructions/initialize.rs* RUST

```rust
fn validate(&self, _fee_bps: u64) -> Result<()> {
    [...]
    cfg_if! {
        if #[cfg(feature = "scaled-ui")] {
            // Validate that the ext mint has the ScaledUiAmount extension and
            // that the ext mint authority is the extension authority
            let extensions = get_mint_extensions(&self.ext_mint)?;
            if !extensions.contains(&ExtensionType::ScaledUiAmount) {
                return err!(ExtError::InvalidMint);
            }
            [...]
        }
    }
    Ok(())
}
```

Remediation

Enforce a whitelist check on the `Token2022` extensions of `ext_mint`.

Patch

The developers acknowledged the issue by mentioning they intend to ensure that any `ext_mint` that is deployed does not utilize a transfer hook with excess accounts.

Code Maturity OS-MSE-SUG-01

Description

1. In `m_ext`, the code utilizes `trunc` and `floor` to convert `multiplier * INDEX_SCALE_F64` into an integer index, which may result in discarding edge cases with small fractional parts such as `.9995`, creating an off-by-one error. Utilizing `round` instead will preserve accuracy by correctly rounding values to the nearest whole number.

>_ *programs/m_ext/src/utils/conversion.rs* RUST

```rust
pub fn sync_multiplier<'info>([...]) -> Result<f64> {
    cfg_if! {
        if #[cfg(feature = "scaled-ui")] {
            [...]
            ext_global_account.yield_config.last_ext_index = (multiplier *
                ↪  INDEX_SCALE_F64).floor() as u64;
            [...]
        }[...]
    }
}

pub fn principal_to_amount_up(principal: u64, multiplier: f64) -> Result<u64> {
    [...]
    let index = (multiplier * INDEX_SCALE_F64).trunc() as u128;
    Ok(amount)
}
```

2. In the `RemoveWhitelistedExt` and `RemoveWhitelistedUnwrapper` accounts of the `ext_swap` program, the `system_program` account is not utilized and should be removed.

>_ *programs/ext_swap/src/instructions/whitelist.rs* RUST

```rust
#[derive(Accounts)]
pub struct RemoveWhitelistedExt<'info> {
    [...]
    pub system_program: Program<'info, System>,
}

#[derive(Accounts)]
pub struct RemoveWhitelistedUnwrapper<'info> {
    [...]
    pub system_program: Program<'info, System>,
}
```

3. In the `m_ext::remove_wrap_authority` instruction, the `ExtGlobal` account is shrunk utilizing `realloc`, but the excess lamports allocated for rent exemption are not refunded to the admin. Either ensure these lamports are refunded, or avoid marking the admin account as mutable if no refund occurs.

   >_ *programs/m_ext/src/instructions/manage_wrap_authority.rs* RUST

   ```rust
   pub fn handler(ctx: Context<Self>, wrap_authority: Pubkey) -> Result<()> {
       [...]
       ctx.accounts
           .global_account
           .to_account_info()
           .realloc(new_size, false)?;
       Ok(())
   }
   ```

4. In `m_ext::set_fee`, if `fee_bps` is updated without first syncing the multiplier, the new fee may be incorrectly applied to outdated yield data. This occurs because `sync_multiplier` utilizes both the fee and the last claimed yield index to compute the new multiplier. Without syncing first, the new `fee_bps` may affect past earnings instead of future ones. To prevent this, the multiplier should be updated utilizing the latest `m_earner_account` state (`m_earner_account.last_claim_timestamp` should be the current timestamp) before calling `set_fee`.

Remediation

Implement the above-mentioned suggestions.

Patch

1. Issue #1 was acknowledged by the developers.
2. Issue #2 was resolved in PR#22.
3. Issue #2 was resolved in PR#23.
4. Issue #2 was resolved in PR#24.

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.