Security Assessment Report

Solana M Earn and Earn Extension

May 02, 2025

Summary

The Sec3 team (formerly Soteria) was engaged to conduct a thorough security analysis of the Solana M Earn and Earn Extension smart contracts.

The artifact of the audit was the source code of the following programs, excluding tests, in https://github.com/m0-foundation/solana-m/tree/2d8e2af.

The initial audit focused on the following versions and revealed 4 issues or questions.

#	program	type	commit
P1	earn	Solana	2d8e2af72d8beae9c995c9630cef2d8952280be9
P2	ext_earn	Solana	2d8e2af72d8beae9c995c9630cef2d8952280be9

This report provides a detailed description of the findings and their respective resolutions.

Table of Contents

Result Overview

Issue	Impact	Status
EARN		
[P1-M-01] Imprecise off-chain `snapshot_balance` calculation	Medium	Resolved
[P1-M-02] Mint account is not reloaded after CPI	Medium	Resolved
EXT_EARN		
[P2-L-01] Manually check `earn_manager_token_account`	Low	Resolved
[P2-Q-01] Is `earn_manager` trustworthy?	Question	Resolved

Findings in Detail

EARN

[P1-M-01] Imprecise off-chain `snapshot_balance` calculation

> *Identified in commit* `2d8e2af`.

How does the off-chain program calculate the `snapshot_balance`, which is a parameter of `claim_for`? Is it determined by taking a snapshot of the token balances of all relevant Earner accounts before calling `claim_for`?

Is it possible that the following situation could occur?

1. The off-chain program fetches an account's balance in time and uses that value as the `snapshot_balance` parameter when calling `claim_for`.
2. In period `i`, the rewards to be distributed are `10%`. The off-chain program retrieves account A's balance as `100`, but the `claim_for` call fails (possibly due to the token account being closed or another issue). So the associated earner account's `last_claim_index` is not updated.
3. In period `i + 1`, another `10%` reward will be distributed. The off-chain program retrieves account's balance again, which is now `10,000`. Since the `last_claim_index` wasn't updated in the previous period, the rewards calculated this time will be: `10,000 * (1.1 * 1.1 - 1)`, which is much higher than what the user is entitled to.

Similarly, it's also possible that the balance is `100` in period `i` and `10,000` in period `i + 1`. Therefore, it's not sufficient to simply use the balance at a single point in time.

If the off-chain program uses the token balance at a single point in time (e.g., always using the most recent balance - similar logic applies if the oldest balance is used), it could potentially be exploited by a malicious attacker:

1. The attacker can analyze on-chain transactions and infer the snapshot selection strategy

used by the off-chain program.

2. The attacker controls two Earner accounts, A and B. They ensure that A holds a large balance while B has a zero balance. The attacker deliberately closes B's `TokenAccount`, keeping its `last_claim_index` unchanged for an extended period.
3. Once the difference between `global_account.index` and `last_claim_index` becomes significant, the attacker reopens B's `TokenAccount` and transfers A's balance into B.
4. At this point, calling `claim_for` would calculate rewards based on B's high balance, even though the tokens were already used to claim rewards under A previously. This effectively allows the same funds to earn double rewards over the same period.

An offchain implementation

The current `getTimeWeightedBalance` function calculates the time-weighted average balance and uses it as the `snapshot_balance` when calling the `claim_for` instruction.

```
// sdk/src/graph.ts#L132-L165, commit 7dd06a7
132 |   private static calculateTimeWeightedBalance(
137 |   ): BN {
142 |     let weightedBalance = new BN(0);
143 |     let prevTS = upperTS;
145 |     // use transfers to calculate the weighted balance
146 |     for (const transfer of transfers) {
147 |       if (upperTS.lt(new BN(transfer.ts))) {
148 |         continue;
149 |       }
150 |       if (lowerTS.gt(new BN(transfer.ts))) {
151 |         break;
152 |       }
154 |       weightedBalance = weightedBalance.add(balance.mul(prevTS.sub(new BN(transfer.ts))));
155 |       balance = balance.sub(new BN(transfer.amount));
156 |       prevTS = new BN(transfer.ts);
157 |     }
159 |     // calculate up to sinceTS
160 |     weightedBalance = weightedBalance.add(balance.mul(prevTS.sub(lowerTS)));
162 |     // return the time-weighted balance
163 |     return weightedBalance.div(upperTS.sub(lowerTS));
164 |   }
```

However, this approach is inaccurate because the global `index` does not necessarily grow at a constant rate over time. It may increase rapidly during some periods and slowly during others.

As a result, applying time-based weighting is imprecise, and this could lead to discrepancies in

rewards between multiple small claims and a single large claim over the same period.

Consider the following example, which shows a scenario with a high initial balance followed by a dramatic withdrawal

Scenario Setup:

- Initial base index at last claim: 1.0
- **Cycle 1** ($T_0 \rightarrow T_1$): Global index increases from `1.0` to `1.1`, and the user balance remains high at `10,000`.
- Cycle 2 ($T_1 \rightarrow T_2$): Global index increases from `1.1` to `2.0`. In Cycle 2, after the user claimed in Cycle 1, the user withdraws most funds so that the balance becomes `100`.
- **Note:** For simplicity, we ignore the compounding of interest in this calculation. This approximation does not affect our conclusion.

Scenario A: Claiming at Every Cycle

cycle	index	user balance	calculation	reward
1 ($T_0 \rightarrow T_1$)	$1.0 \rightarrow 1.1$	$10,000$	$10,000 \times ((1.1/1.0) - 1)$	$1,000$
2 ($T_1 \rightarrow T_2$)	$1.1 \rightarrow 2.0$	100	$100 \times ((2.0/1.1) - 1$	~ 81.8
total				$\sim 1,081.8$

After the claim at Cycle 1, the base index resets to `1.1` for Cycle 2, and the low balance in Cycle 2 leads to a low reward.

Scenario B: Skipping Cycle 1 and Claiming at Cycle 2 Only

Assume the user does not claim at T_1. So the base index remains `1.0` across both cycles. However, the balance during the combined period is given by a weighted average.

For simplicity, we assume that cycle 1 and cycle 2 have the same length, so the average balance over the period is `(10,000 + 100) / 2 = 5,050`

cycle	index	user balance	calculation	reward
1 ($T_0 \rightarrow T_2$)	$1.0 \rightarrow 2.0$	$5,050$	$5,050 \times ((2.0/1.0) - 1)$	$5,050.00$

In this case, the reward is significantly higher (`5,050`) because the large index jump from `1.0` directly to `2.0` applies to an average balance that includes the period when the balance was high.

An attacker can prevent claiming rewards for a specific cycle by closing their own `TokenAccount`, potentially allowing them to gain additional rewards. Moreover, due to the presence of a `max_yield` limit, excessive rewards claimed by the attacker could prevent some legitimate users from receiving their fair share, resulting in a DoS scenario.

Resolution

This issue has been resolved by PR#60 and PR#68.

EARN

[P1-M-02] Mint account is not reloaded after CPI

Identified in commit `2d8e2af`.

In the `ClaimFor` instruction of the `earn` program, after performing a CPI call to the `token` program to mint tokens, the account is not reloaded. As a result, the `mint.supply` value read is the outdated value before the minting operation.

This causes `global_account.max_supply` to be underestimated, which in turn leads to a lower `max_yield` when `PropagateIndex` is called next. Consequently, some users may be unable to claim their rewards properly.

```
/* programs/earn/src/instructions/earn_authority/claim_for.rs */
061 | pub fn handler(ctx: Context<ClaimFor>, snapshot_balance: u64) -> Result<()> {
110 |     mint_tokens(
111 |         &ctx.accounts.user_token_account,      // to
112 |         &rewards,                              // amount
113 |         &ctx.accounts.mint,                    // mint
114 |         &ctx.accounts.mint_multisig,           // multisig mint authority
115 |         &ctx.accounts.token_authority_account, // signer
116 |         token_authority_seeds,                 // signer seeds
117 |         &ctx.accounts.token_program,           // token program
118 |     )?;
          // @audit: should reload "mint" account before reading valus from it after CPI
125 |     if ctx.accounts.mint.supply > ctx.accounts.global_account.max_supply {
126 |         ctx.accounts.global_account.max_supply = ctx.accounts.mint.supply;
127 |     }
```

Resolution

This issue has been fixed by commit [b02649e](b02649e).

EXT_EARN

[P2-L-01] Manually check `earn_manager_token_account`

Identified in commit `2d8e2af`.

In the `claim_for` instruction, if the `earn_manager` is not active (i.e., it has been removed), no fees are charged to the `earn_manager`. However, the program still requires the `earn_manager_token_account` to exist.

```
/* programs/ext_earn/src/instructions/earn_authority/claim_for.rs */
022 | pub struct ClaimFor<'info> {
023 |     pub earn_authority: Signer<'info>,
074 |     #[account(
075 |         seeds = [EARN_MANAGER_SEED, earner_account.earn_manager.as_ref()],
076 |         bump = earn_manager_account.bump,
077 |     )]
078 |     pub earn_manager_account: Account<'info, EarnManager>,
079 |
080 |     #[account(mut, address = earn_manager_account.fee_token_account @ ExtError::InvalidAccount)]
081 |     pub earn_manager_token_account: InterfaceAccount<'info, TokenAccount>,
084 | }

122 |     // Calculate the earn manager fee if applicable and subtract from the earner's rewards
123 |     // If the earn manager is not active, then no fee is taken
124 |     let fee = if ctx.accounts.earn_manager_account.fee_bps > 0 &&
↪  ctx.accounts.earn_manager_account.is_active {
125 |         // Fees are rounded down in favor of the user
126 |         let fee = (rewards * ctx.accounts.earn_manager_account.fee_bps) / ONE_HUNDRED_PERCENT;
127 |
128 |         if fee > 0 {
129 |             mint_tokens(
130 |                 &ctx.accounts.earn_manager_token_account,  // to
131 |                 fee,                                       // amount
132 |                 &ctx.accounts.ext_mint,                    // mint
133 |                 &ctx.accounts.ext_mint_authority,          // mint authority
134 |                 mint_authority_seeds,                      // mint authority seeds
135 |                 &ctx.accounts.token_2022,                  // token program
136 |             )?;
137 |
138 |             // Return the fee to reduce the rewards by
139 |             fee
140 |         } else {
141 |             0u64
142 |         }
143 |     } else {
144 |         0u64
145 |     };
```

Besides, the instruction will also fail if the `earn_manager` is active but the token account is closed. Fortunately, the `earn_manager` can only be added by the admin.

This introduces a potential risk to users' earnings. If the `earn_manager_account.fee_token_account` has already been closed, the `earn_authority` will be unable to invoke the instruction to claim rewards for the user.

It is recommended to make this account `UncheckedAccount` and manually check its existence in the instruction. If it has been closed, the fee collection phase should be skipped.

Resolution

This issue has been fixed by commit a93fec4.

EXT_EARN

[P2-Q-01] Is `earn_manager` trustworthy?

Identified in commit `2d8e2af`.

Is the `earn_manager` is trustworthy? If they act maliciously, it will lead to the following two consequences that potentially put users' earnings at risk:

First, for all the `earner_account` accounts owned by an `earn_manager`, the manager can set the `recipient_token_account` to any account within the `set_recipient` instruction.

An evil `earn_manager` can set the recipient to his own account and collect all user rewards.

```
/* programs/ext_earn/src/instructions/earner/set_recipient.rs */
016 | pub struct SetRecipient<'info> {
017 |     #[account(
018 |         constraint =
019 |             signer.key() == earner_account.user ||
020 |             signer.key() == earner_account.earn_manager
021 |             @ ExtError::NotAuthorized,
022 |     )]
023 |     pub signer: Signer<'info>,

038 |     #[account(
039 |         token::mint = global_account.ext_mint,
040 |         constraint = has_immutable_owner(&recipient_token_account) @ ExtError::MutableOwner,
041 |     )]
042 |     pub recipient_token_account: Option<InterfaceAccount<'info, TokenAccount>>,
043 | }
```

Second, an evil `earn_manager` can set the `fee_bps` field of his `earn_manager_account` to `100%`. As a result, all rewards will be collected as fees.

```
/* programs/ext_earn/src/instructions/earn_manager/configure.rs */
028 | #[account(
029 |     mut,
030 |     seeds = [EARN_MANAGER_SEED, signer.key().as_ref()],
031 |     bump = earn_manager_account.bump
032 | )]
033 | pub earn_manager_account: Account<'info, EarnManager>,

039 | pub fn handler(
040 |     ctx: Context<ConfigureEarnManager>,
041 |     fee_bps: Option<u64>,
```

```
042 | ) -> Result<()> {
049 |         ctx.accounts.earn_manager_account.fee_bps = fee_bps;
```

Resolution

The team clarified that the Earn managers generally need to be trusted by their earners. That being said, the admin owns the token extension and can/should deal with misbehaving earners to prevent issues with their reputation.

At a deeper level, earn managers need to be able to set recipient accounts because the design is to have an earn manager which adds DeFi protocols holding M as earners. The yield cannot be sent directly and has to be redirected for incentives.

Appendix: Methodology and Scope of Work

Assisted by the Sec3 Scanner developed in-house, the manual audit particularly focused on the following work items:

- Check common security issues.
- Check program logic implementation against available design specifications.
- Check poor coding practices and unsafe behavior.
- The soundness of the economics design and algorithm is out of scope of this work

DISCLAIMER

The instance report ("Report") was prepared pursuant to an agreement between Coderrect Inc. d/b/a Sec3 (the "Company") and The Thing GmbH dba M^0 (the "Client''). This Report solely includes the results of a technical assessment of a specific build and/or version of the Client's code specified in the Report ("Assessed Code") by the Company. The sole purpose of the Report is to provide the Client with the results of the technical assessment of the Assessed Code. The Report does not apply to any other version and/or build of the Assessed Code. Regardless of the contents of the Report, the Report does not (and should not be interpreted to) provide any warranty, representation or covenant that the Assessed Code: (i) is error and/or bug free, (ii) has no security vulnerabilities, and/or (iii) does not infringe any third-party rights. Moreover, the Report is not, and should not be considered, an endorsement by the Company of the Assessed Code and/or of the Client. Finally, the Report should not be considered investment advice or a recommendation to invest in the Assessed Code and/or the Client.

This Report is considered null and void if the Report (or any portion thereof) is altered in any manner.

ABOUT

The Sec3 audit team comprises a group of computer science professors, researchers, and industry veterans with extensive experience in smart contract security, program analysis, testing, and formal verification. We are also building automated security tools that incorporate static analysis, penetration testing, and formal verification.

At Sec3, we identify and eliminate security vulnerabilities through the most rigorous process and aided by the most advanced analysis tools.

For more information, check out our website and follow us on twitter.