Security Audit – Streamflow Distributor

Lead Auditor: Sebastian Fritsch

Second Auditor: Mathias Scherer

Administrative Lead: Thomas Lambertz

June 11th 2024

Table of Contents

Executive Summary 3

1 Introduction 4
Findings Summary 4

2 Scope 5

3 Project Overview 6
Functionality 6
On-Chain Data and Accounts 6
Fees and Rent 7
Instructions 7
Authority Structure and Off-Chain Components 8
Upgrade Authority 8
Admin Authority 9
Security Features 9

4 Findings 10
ND-STR1-I1 [Info; Resolved] Logging through emit! can get truncated 11
ND-STR1-I2 [Info; Acknowledged] Small amounts prevent continuous unlocking 13

Appendices

A Methodology 15

B Vulnerability Severity Rating 16

C About Neodyme 17

Executive Summary

Neodyme audited **Streamflow's** on-chain merkle distributor program during April 2024. In addition to the audit in April 2024 Neodyme reviewed a small change on June 10th 2024. Due to the specific threat model of token programs, the scope of this audit included implementation security, overall design and architecture. The auditors found that the Streamflow distributor program comprised a clean design and high code quality. According to Neodymes Rating Classification, **0 critical or high vulnerabilities** and **0 medium-severity issues** were found. The number of findings identified throughout the audit, grouped by severity, can be seen in Figure 1.

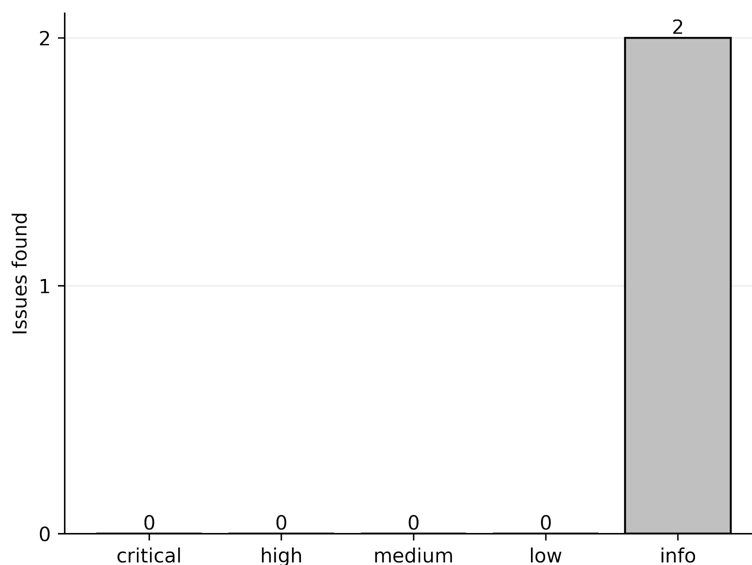


Figure 1: Overview of Findings

All findings were reported to the Streamflow developers and addressed promptly. Neodyme verified the security fixes as complete.

1 | Introduction

In April 2024, Streamflow commissioned Neodyme to conduct a detailed security analysis of Streamflow's on-chain Distributor program.

Two senior auditors performed the audit between April 2nd and April 9th. This report details all findings from this time span.

The audit mainly focused on the contract's technical security but also considered its design and architecture. After the introduction, this report details the audit's Scope, gives a brief Overview of the Contract's Design, then goes on to document our Findings.

Neodyme would like to emphasize the high quality of Streamflow's work. Streamflow's team always responded quickly and **competent** to findings of any kind. Streamflow invested significant effort and resources into their product's security. Their **code quality is above standard**, as the code is very well documented, naming schemes are clear, and the overall architecture of the program is **well thought out, clean and coherent**. The contract's source code has no unnecessary dependencies, relying mainly on its own custom framework.

Findings Summary

During the audit, **0 security-relevant** and **2 informational** findings were identified.

In total, the audit revealed:

0 critical • **0** high-severity • **0** medium-severity • **0** low-severity • **2** informational

issues.

All findings are detailed in section Findings.

2 | Scope

The contract audit's scope comprised of two major components:

- **Implementation** security of the source code
- Security of the **overall design**

All of the source code, located at https://github.com/streamflow-finance/distributor, is in scope of this audit. However, third-party dependencies are not. As Streamflow only relies on the well-established Anchor library, the Anchor SPL library, bytemuck, the solana-program library and solana-security-txt, this is not problematic. Relevant source code revisions are:

- `74b287ab4c1a4d9454b7ea78ee11e16638b03155` • Start of the audit
- `7b6514bd0e9697124897e527f13d599ae5f7b097` • End of the audit
- `e12c0a2c0c8f9dac9e8f7ad37597aa530de2e857` • Last reviewed revision

3 | Project Overview

This section briefly outlines the Streamflow distributor's functionality, design, and architecture, followed by a discussion of its authorities and security features.

Functionality

The Streamflow distributor is a fork of https://github.com/jito-foundation/distributor and extends it by adding an instruction for an admin to close claims, a mechanism for continuous unlocks as well as support for SPL Token-2022.

The distributor makes use of Merkle trees to distribute tokens to users efficiently. Users can claim their tokens by providing the claim data and the co-path of this data to the pre-saved Merkle root.
User claims can have unlocked amounts and locked amounts of tokens, where locked amounts are unlocked in stages according to the unlock period defined by the creator. Directly unlocked tokens are liquid as soon as the airdrop distribution starts. After each passed unlock period, the user can claim an amount proportional to the total amount locked and the number of unlock periods in the timeframe defined by `start_ts` and `end_ts`.

An additional feature built by Streamflow on top of the already existing functionality is to close user claims. The admin, defined during the distributor's creation, can close user claims to prevent them from claiming their tokens. This also works with claims that have already been partially claimed by the user.
To activate this feature, the creator must set the variable `claims_closable` to true. It cannot be activated after the distributor is created.
After the `clawback_start_ts` (defined during creation) timestamp is reached, the admin can withdraw the unclaimed tokens to the predefined clawback receiver.
This timestamp could also be set before the initial deadline (`end_ts`) for users to claim, which allows the admin to clawback all the tokens preliminary and cancel the further distribution.

On-Chain Data and Accounts

The contract uses multiple PDAs, which are seeded as follows:

- MerkleDistributor: `["MerkleDistributor", mint_key, version]`
- ClaimStatus: `["ClaimStatus", claimant_key , distributor_key]`

This data is represented on-chain as follows.

Each Distributor instance has a `MerkleDistributor` account, which stores the main parameters for the distribution:
- `root`: The Merkle Root to compare against
- `mint`: The mint of the token to be distributed
- `token_vault`: The TokenAccount that holds the tokens that are distributed
- `max_total_claim`: How many total claims there are
- `max_num_nodes`: How many nodes are present in the Merkle Tree
- `unlock_period`: The timeframe in which tokens are unlocked
- `total_amount_claimed`: How many tokens have been claimed so far
- `num_nodes_claimed`: How many Merkle leafs have been claimed so far
- `start_ts`: Timestamp of when the distribution starts
- `end_ts`: Timestamp when the last locked tokens get unlocked
- `clawback_start_ts`: Timestamp when the admin is allowed to clawback the remaining tokens (can be before `end_ts`)
- `clawback_receiver`: Which TokenAccount received the clawed back tokens
- `admin`: Public key of the admin account
- `claims_closable`: Defines if claims can be closed by the admin
- `clawed_back`: Boolean flag to indicate if the admin clawed back the remaining tokens

Each valid claim gets stored in a dedicated `ClaimStatus` account, which stores the parameters of the claim:
- `claimant`: The public key of the user who claimed
- `locked_amount`: Total locked amount (withdrawable in parts)
- `unlocked_amount`: Total unlocked amount (withdrawable immediately)
- `locked_amount_withdrawn`: How many tokens that were locked have been withdrawn
- `last_claim_ts`: Last time the user has claimed tokens
- `last_amount_per_unlock`: How many tokens were claimed per unlock period at the last claim

Fees and Rent

The Streamflow distributor functions without any fees.
To claim the tokens granted, a user must pay the rent for the `ClaimStatus` account and the transaction fees.
Neither of these accounts can be closed. Therefore, the rent paid cannot be reclaimed.

Instructions

The contract has 7 instructions, which we briefly summarize here for completeness.

Table 1: Instructions with Descriptions

Instruction	Category	Summary
newDistributor	Permissionless	Create a new distributor
newClaim	User	Claim unlocked tokens and until now unlocked tokens from the distributor
claimLocked	User	Claim locked tokens, if unlocked
clawback	Admin-Only	Clawback all remaining tokens from the vault
closeClaim	Admin-Only	Prevents a user from claiming tokens, even when an existing `ClaimStatus` is present
setClawbackReceiver	Admin-Only	Sets a new receiver for the clawed-back tokens
setAdmin	Admin-Only	Sets a new admin

Authority Structure and Off-Chain Components

A crucial part of the design overview is authorities and components running off-chain. These authorities and components are described in the following.

Upgrade Authority

A program's upgrade authority allows complete control over its behaviour, signatures, and funds. Therefore, it should be well protected.
The team plans to use a Squads multisig to govern the upgrade authority, which we consider a good approach.

Admin Authority

Each distributor has a defined admin authority, which allows to perform certain actions.
These actions include clawing back unclaimed tokens from the vault, closing claims if necessary, updating the clawback receiver address or setting a new admin on the distributor.
Clawing back tokens is only possible after the timestamp `clawback_start_ts` has passed. This variable cannot be updated by the admin after creation.
Closing user claims is only allowed when the flag `claims_closable` is set during distributor creation; it cannot be changed afterwards.

Security Features

Streamflow relies on Solana's main framework, Anchor, which handles basic security checks, such as owner checks.

4 | Findings

This section outlines all of our findings. They are classified into one of five severity levels, detailed in Appendix C.

All findings are listed in Table 2 and further described in the following sections.

Table 2: Findings

Name	Severity
[ND-STR1-I1] Logging through emit! can get truncated	Info
[ND-STR1-I2] Small amounts prevent continuous unlocking	Info

ND-STR1-I1 – Logging through emit! can get truncated

Severity	Impact	Affected Component	Status
Info	Offchain infrastructure could be unable to process onchain events correctly	User Claims	Resolved

Description

Streamflow uses Anchor's `emit!` macro to record events for logging and monitoring purposes. On a technical level, `emit!` calls `sol_log_data` with the serialized event data. `sol_log_data` is susceptible to truncating by RPC providers, and therefore events can be missed and shouldn't be used for important monitoring jobs. (See https://github.com/coral-xyz/anchor/issues/1613 for further discussion of this issue). The same applies to the `msg!` macro.

Location

https://github.com/streamflow-finance/distributor/blob/625861df349852fa32f4445ada73e2b98042c0e2/programs/merkle-distributor/src/instructions/new_claim.rs#L174-184

Relevant Code

```
1  msg!(
2      "Created new claim with locked {} and {} unlocked with lockup start
           :{} end:{}",
3      claim_status.locked_amount,
4      claim_status.unlocked_amount,
5      distributor.start_ts,
6      distributor.end_ts,
7  );
8  emit!(NewClaimEvent {
9      claimant: claimant_account.key(),
10     timestamp: curr_ts
11 });
```

Mitigation Suggestion

Anchor recently added the `emit_cpi!` feature, which logs events by self-reentrancy to a special logging function. That way, logs cannot be truncated.

Remediation

The Streamflow team switched to the new `event_cpi!` feature in commit `7b6514bd0e9697124897e527f13d599ae5f7b097`.

ND-STR1-I2 – Small amounts prevent continuous unlocking

Severity	Impact	Affected Component	Status
Info	Users with small locked amounts are unable to withdraw before the end date	User Claims	Acknowledged

Description

The implementation of `amount_per_unlock()` loses precision when `(end_ts - start_ts)/ unlock_period` is smaller than the locked amount the user is allowed to claim. In this case, `unlocked_amount()` would always return 0 because the division in `amount_per_unlock()` rounds down to the nearest integer, which is, in this case, 0. This will result in users being unable to unlock their tokens until the end timestamp is reached.

Location

https://github.com/streamflow-finance/distributor/blob/74b287ab4c1a4d9454b7ea78ee11e16638b03155/programs/merkle-distributor/src/state/claim_status.rs

Relevant Code

```
1  pub fn amount_per_unlock(&self, start_ts: u64, end_ts: u64,
       unlock_period: u64) -> Result<u64> {
2      if self.locked_amount == 0 {
3          return Ok(0);
4      };
5      let total_duration = end_ts.checked_sub(start_ts).ok_or(
           ArithmeticError)?;
6      if unlock_period >= total_duration {
7          return Ok(self.locked_amount);
8      }
9      let total_unlocks = total_duration
10         .checked_div(unlock_period)
11         .ok_or(ArithmeticError)?;
12     let amount = self
13         .locked_amount
14         .checked_div(total_unlocks)
15         .ok_or(ArithmeticError)?;
16
17     Ok(amount)
18 }
```

Mitigation Suggestion

Because a fix on-chain is difficult, we suggest adding a check to the Merkle Tree generation so that all amounts are larger than `(end_ts - start_ts)/ unlock_period`.

Remediation

The Streamflow team was aware of this issue and acknowledged this code behavior.

A | Methodology

Neodyme prides itself on not being a checklist auditor. We adapt our approach to each audit, investing considerable time into understanding the program upfront and exploring its expected behaviour, edge cases, invariants, and ways in which the latter could be violated. We use our uniquely deep knowledge of Solana internals and our years-long experience in auditing Solana programs to even find bugs that others miss. We often extend our audit to cover off-chain components in order to see how users could be tricked or the contract affected by bugs in those components.

Nonetheless, we also have a list of common vulnerability classes, which we always exhaustively look for. We provide a sample of this list below.

- Rule out common classes of Solana contract vulnerabilities, such as:
  - Missing ownership checks
  - Missing signer checks
  - Signed invocation of unverified programs
  - Solana account confusions
  - Redeployment with cross-instance confusion
  - Missing freeze authority checks
  - Insufficient SPL account verification
  - Missing rent exemption assertion
  - Casting truncation
  - Arithmetic over- or underflows
  - Numerical precision errors
- Check for unsafe designs which might lead to common vulnerabilities being introduced in the future
- Check for any other, as-of-yet unknown classes of vulnerabilities arising from the structure of the Solana blockchain
- Ensure that the contract logic correctly implements the project specifications
- Examine the code in detail for contract-specific low-level vulnerabilities
- Rule out denial of service attacks
- Rule out economic attacks
- Check for instructions that allow front-running or sandwiching attacks
- Check for rug pull mechanisms or hidden backdoors

B | Vulnerability Severity Rating

Critical Vulnerabilities that will likely cause loss of funds. An attacker can trigger them with little or no preparation, or they are expected to happen accidentally. Effects are difficult to undo after they are detected.

High Bugs that can be used to set up loss of funds in a more limited capacity, or to render the contract unusable.

Medium Bugs that do not cause direct loss of funds but that may lead to other exploitable mechanisms, or that could be exploited to render the contract partially unusable.

Low Bugs that do not have a significant immediate impact and could be fixed easily after detection.

Info Bugs or inconsistencies that have little to no security impact.

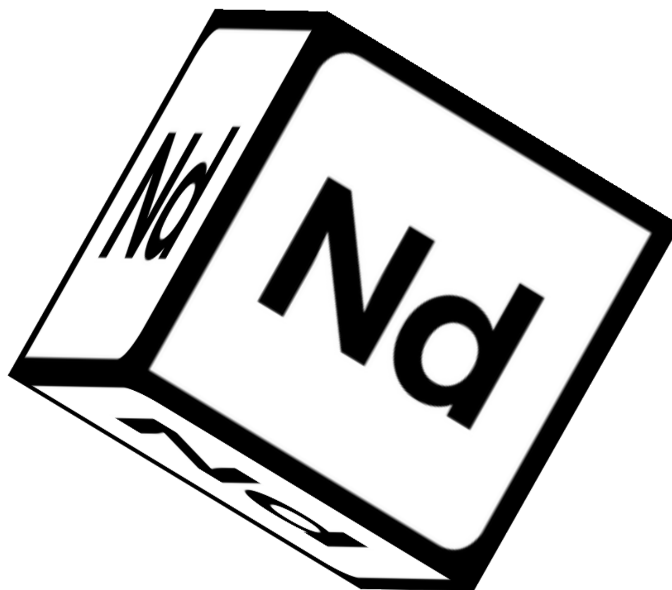
C | About Neodyme

Security is difficult.

To understand and break complex things, you need a certain type of people. People who thrive in complexity, who love to play around with code, and who don't stop exploring until they fully understand every aspect of it. That's us.

Our team never outsources audits. Having found over 80 High or Critical bugs in Solana's core code itself, we believe we have the most qualified auditors for Solana programs in our company. We've also found and disclosed critical vulnerabilities in many of Solana's top projects and have responsibly disclosed issues that could have resulted in the theft of over $10B in TVL on the Solana blockchain.

Our team met as participants in hacking competitions called CTFs. There, we competed and collaborated while finding vulnerabilities, breaking encryption, reverse engineering complicated algorithms, and much more. Through the years, many of our team members have won national and international hacking competitions and keep ranking highly among some of the hardest CTF events worldwide. In 2020, some of our members started experimenting with validators and became active members of the early Solana community. With the prospect of an interesting technical challenge and bug bounties, they quickly encouraged others from our CTF team to look for security issues in Solana. The result was so successful that after reporting several bugs, in 2021, the Solana Foundation contracted us for source code auditing. As a result, Neodyme was born.



Neodyme AG

Dirnismaning 55
Halle 13
85748 Garching
E-Mail: contact@neodyme.io

https://neodyme.io