Security Report – Streamflow Protocol

Lead Auditor: Sebastian Fritsch

Second Auditor: Mathias Scherer

Administrative Lead: Thomas Lambertz

May 16th 2024

Table of Contents

Executive Summary

Neodyme audited **Streamflow's** on-chain token distribution protocol program during April and Mai 2024.

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.

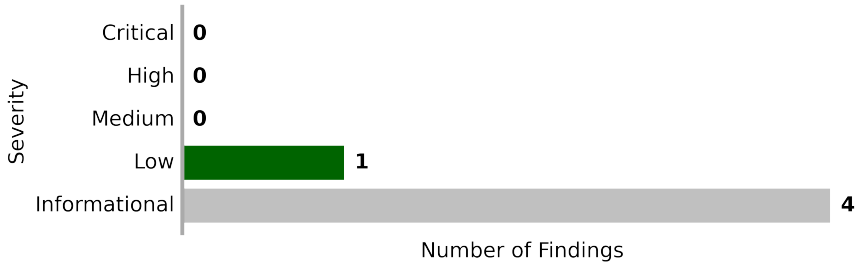

Figure 1: Overview of Findings

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 token distribution protocol. Neodyme did an initial peer-review of the program in February 2022.

The first pass by an auditor was between April 9th and April 17th and the second between Mai 7th and Mai 15th.

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**.

Additionally, Neodyme delivered the Streamflow team a list of nitpicks and additional notes that are not part of this report.

Findings Summary

During the audit, **1 security-relevant** and **4 informational** findings were identified.

In total, the audit revealed:

0 critical • **0** high-severity • **0** medium-severity • **1** low-severity • **4** 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/protocol, is in scope of this audit. However, third-party dependencies are not. Relevant source code revisions are:

- `db63597baa4e994dff2c67db55e44d1decec18c5` • Start of the first audit pass
- `c2dee4f65aa2a712aedb2decfc182f192c785d23` • Start of the second audit pass
- `4248294ef9dac378632ac32dbe2786828db2d91b` • Last reviewed revision

3 Project Overview

This section briefly outlines the Streamflow protocol's functionality, design, and architecture, then discusses its authorities and security features.

Functionality

The Streamflow protocol provides a way to distribute tokens in a time-vested manner.

A user can vest tokens that will be distributed over time to a pre-defined spender. Each such arrangement is called a 'Contract'. The vesting contract consists of a cliff that is unlocked at a specified timestamp and an amount of money that linearly unlocks over a specified time period. All the tokens that will be distributed must be deposited upfront, ensuring their distribution later on.

The contract creator can also decide to top up the current amount of money and thereby extend the running time and capital of the vesting contract. Furthermore, Streamflow allows the creator to enable additional settings. For example pausing, and ability to cancel the contract. On cancel, all currently unlocked funds will be transferred to the recipient, and the remaining funds will be returned to the sender. A creator can also define the contract as updatable, which allows to update the withdraw frequency and amount, as well as enabling automatic withdrawals. When enabled, Streamflow also offers the possibility to change a contract's recipient address, thereby transferring the remaining claims to a new user. A contract can specify whether the recipient, creator, or both can transfer or cancel the contract.

Streamflow offers automatic withdrawals, a service in which an off-chain bot calls the withdraw function on behalf of the contract recipient, relieving him from manually withdrawing his unlocked funds. The contract creator pre-transfers the transaction fees for this.

On-Chain Data and Accounts

The contract's metadata, a crucial component, is stored in a single program-owned account type. This metadata includes the locked amount, timestamps for start, end and cliff, the recipient of the tokens, the token mint, and more. Streamflow uses one single account per vesting contract.

Additionally, every contract has an associated escrow account, where the tokens are stored for further distribution. This PDA is derived by the following seeds: `["strm", metadata_pubkey]`

Fees and Rent

Streamflow charges a fee for utilizing the protocol. The fees consist of a fee directly to Streamflow and a partner/referral fee. The fees are queried via the Streamflow partner oracle program. If no valid partner is submitted, Streamflow defaults to a 0% partner fee and a 1% fee to Streamflow.

The total fee amount is added to the amount that will be distributed to the recipient and deposited by the contract creator at the time the contract is created. The fees are not directly transferred to Streamflow and the partner. Instead, they are distributed proportionally with every unlock, ensuring a fair and transparent distribution process.

The contract creator has to pay the Solana account rent when creating a new vesting contract for the metadata PDA, escrow PDA and if not already existing the recipients and fee recipients ATA accounts. If the contract has finished, the escrow account will be closed, and the remaining rent will be transferred to the Streamflow treasury. The metadata account will not be closed and no rent will be reclaimed.

Instructions

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

Table 1: Instructions with Descriptions

Instruction	Category	Summary
Create	Permissionless	Create a new vesting contract
CreateUnchecked	Permissionless	Create a new vesting contract with relaxed checks on existing accounts
Topup	Creator-Only	Increases the amount of tokens to be distributed
Pause	Creator-Only	Pause the vesting contract (if allowed)
Unpause	Creator-Only	Unpause the vesting contract
Withdraw	Withdraw-Authority	Withdraw unlocked tokens to the recipient and transfer fees
Transfer	Transfer-Authority	Change the contract recipient
Update	Update-Authority	Update the unlocked amount per period (Creator-Only), the automatic withdraw frequency and enable automatic withdrawals

Instruction	Category	Summary
Cancel	Cancel-Authority	Cancel the vesting contract, transferring unlocked tokens to the recipient and locked tokens to the creator

Authority Structure

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 Streamflow team currently uses a three-out-of-five Squads multisig to govern the upgrade authority, which means no single key can do an upgrade by itself.

Contract Creator

The contract creator can turn on and off different functionalities of the vesting contract during the contract creation. In the following, we will outline those properties:

- `cancelable_by_sender`: If set to true, the contract creator can cancel this stream and reclaim locked tokens. This allows the creator to prevent payout of future tokens, though already unlocked tokens cannot be held-back.
- `transferable_by_sender`: The contract creator can transfer this stream to a different recipient if set to true. All future payouts will go to the new recipient.
- `can_topup`: If set to true, the contract creator can increase the vested amount in the contract. This cannot have any bad consequences for the recipient.
- `pausable`: If set to true, the contract creator can pause the vesting contract and later unpause it. This can prevent the recipient from receiving any tokens, but the contract creator cannot receive them themselves either.
- `can_update_rate`: If set to true, the contract creator can increase or decrease the unlocked amount per period.

Contract Recipient

Depending on the contract specification, the contract receiver also has the ability to perform certain actions. Those include analogues to the above `cancelable_by_recipient` and `transferable_by_recipient`. Furthermore, the recipient can also `Update` the automatic withdrawal frequency or enable automatic withdrawals.

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-STR2-L1] Pausing logic can result in unexpected behaviour	Low
[ND-STR2-I1] create_account allows for DoS attack	Info
[ND-STR2-I2] Floating point math can lead to minor rounding errors	Info
[ND-STR2-I3] Presence of unused but broken utility function	Info
[ND-STR2-I4] Missing SPL Token 2022 extension support	Info

ND-STR2-L1 – Pausing logic can result in unexpected behaviour

Severity	Impact	Affected Component	Status
Low	Pausing logic can result in unexpected behaviour	Pause mechanism	Resolved

Description

Streamflow allows the contract creator to pause the stream if the pausing functionality is enabled during stream creation. Streamflow treats this as a no-op if the stream gets paused and unpaused before any unlocks happen. This is intended behaviour, but an edge case was present in the unpause implementation, which could lead to a delayed start and a cumulated payout in the end.

Assume the contract got paused after the `start_time` timestamp but before the `cliff` timestamp and gets unpaused before the `cliff` timestamp. As there were no unlocks yet, this pause should be treated as a no-op. This was not the case in the previous implementation, as the check for actually adding the pause time only checked if `now > start_time` and not for the `cliff` timestamp.

Location

https://github.com/streamflow-finance/protocol/blob/31d0c6cd543e9f5f417124f2852441718cd02901/programs/protocol/src/state.rs#L514-542

Relevant Code

```
pub fn unpause(&mut self, now: u64) -> Result<(), ProgramError> {
    let cliff = self.ix.cliff;
    let current_pause_length = self.current_pause_len(now)?;

    // if stream was paused after start or if pause shifts start_time, we should also shift end time
    if self.current_pause_start >= self.ix.unlock_start() || now >= self.ix.unlock_start() {
        self.end_time.try_add_assign(current_pause_length)?;
    }

    //increment cumulative
    if now > self.ix.start_time {
        self.pause_cumulative.try_add_assign(current_pause_length)?;
    }

```

```
15      ...
16
17      self.current_pause_start = 0;
18
19      Ok(())
20  }
```

Mitigation Suggestion

We recommend checking that `now > self.ix.unlock_start()` to incorporate the check for a cliff.

Remediation

The Streamflow quickly responded to the issue and implemented the recommended mitigation in commit `c2dee4f65aa2a712aedb2decfc182f192c785d23`.

ND-STR2-I1 – create_account allows for DoS attack

Severity	Impact	Affected Component	Status
Info	The contract creation through CreateUnchecked could get interrupted	Contract Creation	Resolved

Description

In the `CreateUnchecked` instruction, Streamflow uses the `create_account` method to initialize the escrow PDA. In contrast to the `Create` instruction, the metadata account already has to be created and initiated here. Due to an intrinsic of `create_account`, it will fail if the to-be-created account already has more than zero lamports of balance. As anyone can transfer lamports to any account, this allows for a Denial of Service attack if the initial metadata account creation and the stream creation are in separate transactions.

An attacker would scan for the creation of empty metadata accounts and transfer lamports to the to-be-created escrow PDA, thereby triggering a failure in `CreateUnchecked`.

This would prevent new streams from being created with the `CreateUnchecked` method.

In the `Create` instruction, this problem does not arise, although `create_account` is used because the to-be-created account is derived from the newly created metadata account.

Location

https://github.com/streamflow-finance/protocol/blob/31d0c6cd543e9f5f417124f2852441718cd02901/programs/protocol/src/create_unchecked.rs#L245-257

Relevant Code

```
let seeds = [ESCROW_SEED_PREFIX, acc.metadata.key.as_ref(), &[
    stream_escrow_bump]];

invoke_signed(
    &system_instruction::create_account(
        acc.payer.key,
        acc.escrow_tokens.key,
        tokens_rent,
        tokens_struct_size as u64,
        acc.token_program.key,
    ),
```

```
11      &[acc.payer.clone(), acc.escrow_tokens.clone(), acc.system_program.
            clone()],
12      &[&seeds],
13  )?;
```

Mitigation Suggestion

A pattern has evolved to mitigate this issue. It checks whether the current account balance is > 0 and, if so, manually transfers, allocates, and assigns the new owner. An example of this can be found in the Neon source code.

Remediation

The Streamflow quickly responded to the issue and implemented the modified creation pattern in commit `0fb65d2e4450e343ad63bc90cd3c091074d01c61`.

ND-STR2-I2 – Floating point math can lead to minor rounding errors

Severity	Impact	Affected Component	Status
Info	An unlock might contain slightly more or less tokens than mathematically exact	Unlock calculation	Acknowledged

Description

To calculate the amount that is already unlocked for the user to withdraw and the corresponding fees, Streamflow uses floating point math with a precision factor. As rounding floating numbers in Rust defaults to the nearest representable value, this might result in a number slightly off from the mathematically exact result. This could lead to a slightly skewed unlocked amount. As the last unlock period will withdraw all remaining funds, the user can be assured of receiving the total deposited amount, and possible rounding errors will be mitigated in the end.

Location

https://github.com/streamflow-finance/protocol/blob/db63597baa4e994dff2c67db55e44d1dececc18c5/programs/protocol/src/utils.rs#L100-108

Relevant Code

```
/// Given amount and percentage, return the u64 of that percentage.
pub fn calculate_fee_from_amount(amount: u64, percentage: f32) -> u64 {
    if percentage <= 0.0 {
        return 0
    }
    let precision_factor: f32 = 1000000.0;
    let factor = (percentage / 100.0 * precision_factor) as u128; // largest it can get is 10^4
    (amount as u128 * factor / precision_factor as u128) as u64 // this does not fit if amount
                                                                 //
        itself cannot fit into u64
}
```

Mitigation Suggestion

Use fixed point math to calculate the percentages.

Remediation

The Streamflow acknowledged this issue and decided to keep the current implementation.

ND-STR2-I3 – Presence of unused but broken utility function

Severity	Impact	Affected Component	Status
Info	None, but future refactors might cause issues	Deposit calculation	Resolved

Description

There is an unused utility function `deposit_gross`, which does erroneous calculations. During the calculation of the net amount from the gross amount, the code calls `calculate_fee_from_amount` twice to calculate the fees included in the gross amount and deduct these fees to get the deposited net amount. `calculate_fee_from_amount` assumes that the passed amount is net and therefore calculates the fees incorrectly which results in an incorrect net amount added to the metadata.

Locations

https://github.com/streamflow-finance/protocol/blob/db63597baa4e994dff2c67db55e44d1dcec18c5/programs/protocol/src/state.rs#L616-619
https://github.com/streamflow-finance/protocol/blob/db63597baa4e994dff2c67db55e44d1dcec18c5/programs/protocol/src/utils.rs#L107

Relevant Code

state.rs

```
pub fn deposit_gross(&mut self, gross_amount: u64) -> Result<(),
    ProgramError> {
    let partner_fee_addition =
        calculate_fee_from_amount(gross_amount, self.
            partner_fee_percent);
    let strm_fee_addition =
        calculate_fee_from_amount(gross_amount, self.
            streamflow_fee_percent);
    let net_amount = gross_amount.try_sub(partner_fee_addition)?.
        try_sub(strm_fee_addition)?;
    self.ix.net_amount_deposited.try_add_assign(net_amount)?;
    self.partner_fee_total.try_add_assign(partner_fee_addition)?;
    self.streamflow_fee_total.try_add_assign(strm_fee_addition)?;
    self.end_time = self.effective_end_time()?;
    Ok(())
}
```

utils.rs

```rust
pub fn calculate_fee_from_amount(amount: u64, percentage: f32) -> u64 {
    if percentage <= 0.0 {
        return 0
    }
    let precision_factor: f32 = 1000000.0;
    let factor = (percentage / 100.0 * precision_factor) as u128;
    (amount as u128 * factor / precision_factor as u128) as u64
}
```

Mitigation Suggestion

Because `deposit_gross` and `try_sync` are currently not in use we suggest removing them.

Remediation

Streamflow removed the functions `deposit_gross` and `try_sync_balance` from their codebase.

ND-STR2-I4 – Missing SPL Token 2022 extension support

Severity	Impact	Affected Component	Status
Info	Missing SPL Token 2022 extension support	Dependency support	Resolved

Description

Even though the protocol supports basic SPL Token 2022 mints it doesn't work with mints that require extensions for token accounts. The Token2022 program requires the token account size to be big enough for the extensions to fit, but the Streamflow protocol program only creates PDAs with the size of a default account (165 bytes) without any room for extensions. Because of this, the call to the account initialization instruction in Token2022 fails with an `InvalidAccountData` error, if the mint requires account extensions.

Location

https://github.com/streamflow-finance/protocol/blob/db63597baa4e994dff2c67db55e44d1dcec18c5/programs/protocol/src/create.rs#L232
https://github.com/streamflow-finance/protocol/blob/db63597baa4e994dff2c67db55e44d1dcec18c5/programs/protocol/src/create_unchecked.rs#L224

Relevant Code

state.rs

```
pub fn create(pid: &Pubkey, acc: CreateAccounts, mut ix: CreateParams)
    -> ProgramResult {
    msg!("Initializing SPL token stream");

    ...

    let tokens_struct_size = spl_token_2022::state::Account::LEN;
    let cluster_rent = Rent::get()?;
    let metadata_rent = cluster_rent.minimum_balance(
        metadata_struct_size);
    let mut tokens_rent = cluster_rent.minimum_balance(
        tokens_struct_size);
    if acc.recipient_tokens.data_is_empty() {
        tokens_rent.try_add_assign(cluster_rent.minimum_balance(
            tokens_struct_size))?;
    }
```

```
    let withdraw_fees = metadata.withdraw_fees()?;
    if acc.sender.lamports() < metadata_rent.try_add(tokens_rent)?.
       try_add(withdraw_fees)? {
        msg!("Error: Insufficient funds in {}", acc.sender.key);
        return Err(ProgramError::InsufficientFunds)
    }

    ...

    msg!("Creating stream escrow account");
    let seeds = [ESCROW_SEED_PREFIX, acc.metadata.key.as_ref(), &[
       stream_escrow_bump]];

    create_pda_account_safe(
        &acc.system_program,
        acc.token_program.key,
        &acc.sender,
        &acc.escrow_tokens,
        &seeds,
        tokens_struct_size,
    )?;

    msg!("Initializing stream escrow SPL token account");
    invoke(
        &spl_token_2022::instruction::initialize_account(
            acc.token_program.key,
            acc.escrow_tokens.key,
            acc.mint.key,
            acc.escrow_tokens.key,
        )?,
        &[
            acc.token_program.clone(),
            acc.escrow_tokens.clone(),
            acc.mint.clone(),
            acc.escrow_tokens.clone(), // owner
            acc.rent.clone(),
        ],
    )?;

    ...

    Ok(())
}
```

Mitigation Suggestion

We suggest using the same logic as the AssociatedTokenAccount program by calling the Token2022 programs `GetAccountDataSize` instruction to retrieve the required size for the token account. https://github.com/solana-labs/solana-program-library/blob/0ab6ed7869679c0f5e2a72068e7a4e0

591076d1f/associated-token-account/program/src/tools/account.rs#L77-L100

Because certain extensions like the transfer fee open up other vulnerabilities and logic issues for the streamflow protocol we suggest implementing an allowlist which extensions the program supports and blocks mints that require extensions that aren't explicitly allowed.
Example code to unpack a mint and retrieve the enabled extensions:

```
let mint = spl_token_2022::extension::PodStateWithExtensions::<
   spl_token_2022::pod::PodMint>::unpack(&account_info.data.borrow())?;
let extensions = mint.get_extension_types()?;
```

Remediation

The streamflow team added support for extensions in commit `4248294ef9dac378632ac32dbe2786828db2d91b`.
As suggested they implemented an allowlist and allowed the following extensions:

- TransferFeeConfig
- MintCloseAuthority
- ConfidentialTransferMint
- DefaultAccountState
- NonTransferable
- InterestBearingConfig
- PermanentDelegate
- ConfidentialTransferFeeConfig
- MetadataPointer
- TokenMetadata
- GroupPointer
- TokenGroup
- GroupMemberPointer
- TokenGroupMember

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