Security Audit - stabble AMM

conducted by Neodyme AG

Lead Auditor:	Jasper Slusallek
Second Auditor:	Mathias Scherer
Administrative Lead:	Thomas Lambertz

Finalized: October 10, 2024
Last Updated: November 14, 2024

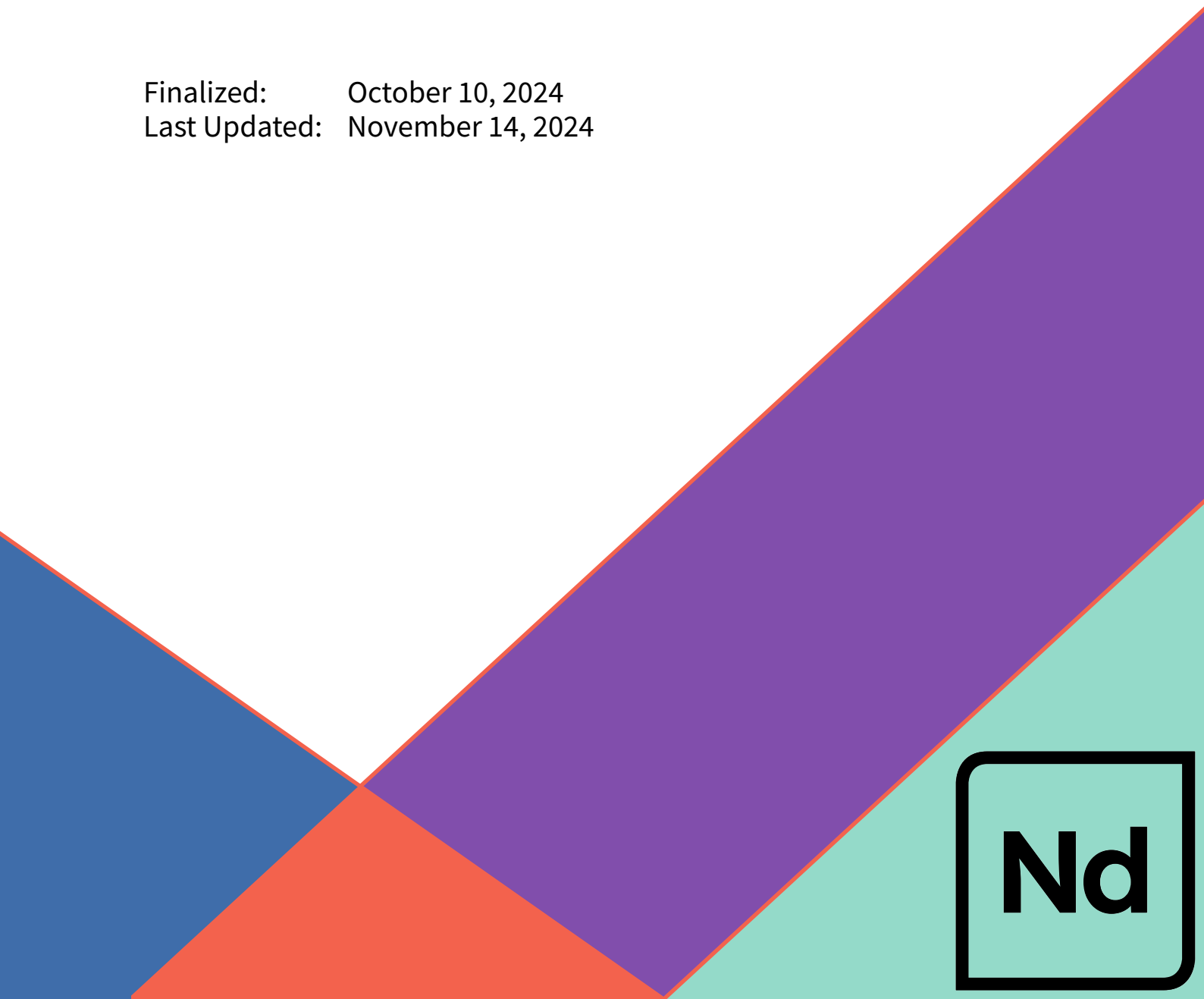

Table of Contents

Appendices

1 | Executive Summary

Neodyme audited **stabble's** on-chain stable-swap, weighted-swap, and vault programs in September of 2024.

Two senior researchers from Neodyme, Jasper Slusallek and Mathias Scherer, conducted independent full audits of the contract. The scope of this audit included the implementation security of these programs as well as the math libraries written by stabble to implement their business logic.

The auditors found that stabble's programs comprised a clean design and excellent code quality. According to Neodyme's Rating Classification, **1 issue above low severity** was found.

The number of findings identified throughout the audit, grouped by severity, can be seen in Figure 1.

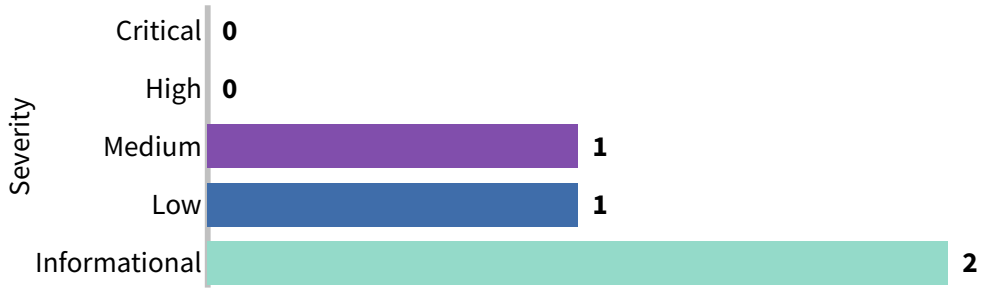

Figure 1: Overview of Findings

The auditors reported all findings to the stabble developers, who addressed them promptly. The security fixes were verified for completeness by Neodyme.

In addition to these findings, Neodyme delivered the stabble team a short list of nit-picks and additional notes that are not part of this report.

2 | Introduction

During September of 2024, stabble engaged Neodyme to do a detailed security analysis of their on-chain AMM.

The audit focused on the contract's technical security, as well as any economical risks. In the following sections, we discuss the protocol's functionality and security framework, as well as presenting our findings.

Neodyme would like to emphasize the high quality of stabble's code. Though some code duplication is apparent, the codebase is generally lean and well-structured, with the logic relying on well-tested routines. Naming schemes are clear, and the overall architecture of the program is clean and coherent.

The contract's source code has no unnecessary dependencies, relying mainly on the well-established Anchor framework.

Summary of Findings

During the audit **2 security-relevant** findings were identified. All found issues were **quickly remediated**. In total, the audit revealed:

0 critical • **0** high-severity • **1** medium-severity • **1** low-severity • **2** informational

issues.

3 | Scope

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

- Primarily, the **Implementation** security of the contract's source code
- Additionally, security of the **overall design**
- Additionally, resilience against **economical attacks**

Neodyme considered the source code of the on-chain programs in-scope for this audit. Specifically, this comprises:

- The code of the stable-swap, weighted-swap and vault contracts, located in the `programs` folder of the repository at https://github.com/stabbleorg/amm-v1, and
- the core mathematical functionality used by the contracts, located in the `libraries` folder of the repository at https://github.com/stabbleorg/amm-sdk.

Third-party dependencies are not in scope. It should be noted that stabble only relies on the Anchor library, the solana-program and the bn and math libraries, all of which are well-established.

During the audit, minor changes and fixes were made by stabble, which the auditors also reviewed in-depth.

Relevant source code revisions are as follows. For the `amm-v1` repository:

- `12c18ed3f2d8873cf3e1a89e1cfad2334a3ca367` · Start of the audit
- `fc465ade3c0e342be404876723cc7662660b462d` · Last reviewed revision

For the `amm-sdk` repository:

- `be385d25652325d365de7e0b75a9d7d32b7bae59` · Start of the audit
- `f15d379afe801dec49e214e88ee14e01493c5a06` · Last reviewed revision

4 | Project Overview

This section briefly outlines stabble's functionality, design, and architecture, followed by a detailed discussion of all related authorities.

Functionality

stabble's contracts implement AMMs for swaps with advanced liquidity invariants. They have two types of pools:

Firstly, **Weighted pools**, which hold a multitude of assets, each with a different weight. Unlike the traditional `x*y=k` formula for two-asset AMMs, they ensure that when discounting swap fees, the product of all the pool's token balances, each taken to the power of its weight, should not change during swaps. The weights must sum to 1. We will call this product the "weighted invariant" for the rest of this document.

Their input and output calculations closely follow Balancer's description of their Weighted Math formulas.

Secondly, **Stable pools**, which hold any number of tokens that have the same peg – for example, tokens that are all pegged to USD, or a collection of liquid SOL staking tokens. These pools concentrate their liquidity around a 1:1 price point, improving swap prices under the assumption that none of the tokens depeg significantly.

The stable-math equation is $A \cdot n^n \cdot \sum x_i + D = A \cdot D \cdot n^n + \frac{D^{n+1}}{n^n \cdot \prod x_i}$, where D is the invariant, n is the number of tokens, x_i is the balance of token i and A is a constant amplification coefficient chosen by the pool. In this formula, D must stay constant across swaps (discounting fees). Note that the equation is a polynomial in D. In order to calculate D, stabble uses a Newton-Raphson approximation with 255 iterations, much like Balancer's implementation.

For providing liquidity, in both types of pools, users can deposit any number of tokens in any distribution they like, and receive LP tokens proportional to how much they have increased the pool's invariant – that is, the amount of output tokens received is determined by `LP_token_supply * (invariant_after_deposit/invariant_before_deposit - 1)`. Some fees are applied for unbalanced deposits, e.g. for deposits that only deposit one token.

When withdrawing, users burn LP tokens and can choose between receiving a proportional share of each token held by the pool, or receiving a single token back. In the latter case, the new invariant is calculated via `invariant * (LP_token_supply - LP_to_burn) / LP_token_supply`, and subsequently the output amount is calculated via the invariant equation where all other token balances stay constant.

Stable's contracts are split into a contract for stable swaps, a contract for weighted swaps, and a simple vault contract that holds all assets.

The implementation is split into the amm-v1 repository, which holds the anchor implementation of the instructions and general business logic, and the amm-sdk repository, which holds in its `libraries` folder the core mathematical functions that are called by the instructions for things like calculating the invariant or calculating output amounts.

On-Chain Data and Accounts

The program stores fairly little data on-chain, mostly comprised of owner information and pool parameters.

Vault Contract

The vault contract is minimal in function. It has a single account for each vault that stores:

- The vault's admin, who can change the vault's parameters, assign a new admin, or pause/unpause the vault. In the stable swap contract, any of its pools that use this vault also allow the vault's admin to change the pool's amplification factor parameters.
- An optional pending admin, which is set to an address only if the current admin is transferring their role
- The authority who is able to withdraw from the vault
- The vault's beneficiary, who gets a portion of the pool's fees
- The beneficiary fee, which defines said share
- A boolean indicating whether the vault is currently active or paused
- Relevant bump seeds

The beneficiary fee is enforced via the swap contracts, not the vault contract itself.

The vault contract has a single relevant PDA for each vault, the vault authority. It is derived via the seeds `[b"vault_authority", vault.key()]` and is the authority for the funds that the contract holds.

Said funds are held in ATAs of the vault authority. This is enforced only in the pool's deposit functions – the vault itself does not verify the accounts, and withdrawals from the pool do not verify where the funds come from.

Stable Swap Contract

The stable swap contract has a single account for each pool that stores:

- The pool's owner, who can assign a new owner, change configuration parameters (save for the amplification factor parameters, which is controlled by the vault's admin) and pause the pool
- An optional pending owner, which is set to an address only if the current owner is transferring their role
- The pool's vault, where its assets are stored. This is the address of the account storing the vault configuration, as described in the section on the vault contract above.
- The mint of the LP token

- Parameters of the amplification factor. The contract includes functionality for gradually changing the factor over time. The parameters include a start and end timestamp of the change, as well as the starting and ending point (“target”) of the change.
- The fee charged on swaps
- A list of supported tokens in the pool. For each token, the contract stores:
  - The token’s mint and decimals
  - Scaling parameters that specify a scaling factor that is applied to any balances
  - The pool’s total balance of that token. This is an internal tracker which is never synced with the funds held in the vault but updated for swaps and LP deposits or withdrawals
- A boolean indicating whether the pool is currently active or paused
- Relevant bump seeds

Each pool has a pool authority associated with it. This is a PDA derived via `[b"pool_authority", pool.key()]` and is the mint authority of the LP token.

Additionally, each pool has a withdraw authority PDA seeded via `[b"withdraw_authority", vault.key()]`

Weighted Swap Contract

Similarly, the weighted swap contract has a single account for each pool that stores:

- the pool’s owner, a pending owner, the pool’s vault, the mint of the LP token, the swap fee, a boolean indicating pause status and bump seeds, all just as in the stable swap contract
- A list of supported tokens in the pool like in the stable swap contract. However, in addition to the metadata stored by the stable swap contract, a field for the token’s `weight` is added.
- The pool’s invariant, as recorded during the first liquidity provision

Again, each pool has a PDA associated with it which is the pool authority. It works the same and is seeded the same as in the stable swap contract.

Relationship of Pools and Vaults

Unintuitively, vaults are not specific to a pool. A vault can act as the funds storage for more than one pool, and indeed any new (permissionless created) pool can use any existing vault as its `pool.vault`. The funds will be mixed with those of the other pools using the vault.

We could not find a way to withdraw more funds out of a pool than were deposited in even as admin, hence there does not appear to be a way to use this unusual pool-vault relationship to extract funds from other pools. However, we would advise clarifying this in user documentation to avoid confusion.

Instructions

For completeness, we briefly summarize the instructions of the contracts here.

Vault Contract

Instruction	Category	Summary
Initialize	Permissionless	Initializes a vault by initializing the vault account and populating it with user-provided data, verifying only the beneficiary fee
Config Instructions	Admin Only	Changes the pool's configuration data, including the active bit, the beneficiary and the beneficiary fee. Can also change the admin, in which case the new admin has to accept in a separate config instruction.
Withdraw	Withdraw Authority Only	Withdraws funds from a token account where the vault authority is the token authority. Takes two amounts, one intended for an instruction-provided user account and one intended for the vault's beneficiary account.
Withdraw V2	Withdraw Authority Only	Same as Withdraw, but supports token22.

Stable Swap Contract

Instruction	Category	Summary
Initialize	Permissionless	Initializes a new pool with the given token configuration, LP mint, user-provided pool configuration. It checks that the LP mint has no other authorities than the pool itself.
Shutdown	Permissionless	Closes the pool if all token balances are 0.
Deposit	Permissionless	Deposits tokens in exchange for LP tokens.
Withdraw	Liquidity Provider	Withdraws liquidity from the pool for the given amount of LP Tokens.
Swap	Permissionless	Swaps one token for the other. Supports only Tokenkeg tokens
SwapV2	Permissionless	Swaps one token for the other. Supports Tokenkeg and Token2022 tokens.
Change AMP Factor	Vault Admin	Activate the amplification factor change to the user-defined value and time range.
Change Swap Fee	Owner	Changes the swap fee.
Pause	Owner	Pauses the pool.
Unpause	Owner	Unpauses the pool.

Instruction	Category	Summary
Transfer Owner	Owner	Sets a new `pending_owner` for the pool.
Accept Owner	Pending Owner	Accepts the owner transfer. Resets the `pending_owner` and sets the new owner.
Reject Owner	Pending Owner	Rejects the owner transfer and resets the `pending_owner` value.

Weighted Swap Contract

Instruction	Category	Summary
Initialize	Permissionless	Initializes a pool by initializing the pool account and populating it with user-provided data.
Shutdown	Permissionless	Closes the pool if all token balances are 0.
Deposit	Permissionless	Deposits tokens in exchange for LP tokens.
Withdraw	Liquidity Provider	Withdraws liquidity from the pool for the given amount of LP Tokens.
Swap	Permissionless	Swaps one token for the other. Supports only Tokenkeg tokens
SwapV2	Permissionless	Swaps one token for the other. Supports Tokenkeg and Token2022 tokens.
Change Swap Fee	Owner	Changes the swap fee.
Pause	Owner	Pauses the pool.
Unpause	Owner	Unpauses the pool.
Transfer Owner	Owner	Sets a new `pending_owner` for the pool.
Accept Owner	Pending Owner	Accepts the owner transfer. Resets the `pending_owner` and sets the new owner.
Reject Owner	Pending Owner	Rejects the owner transfer and resets the `pending_owner` value.

Authority Structure

Upgrade Authority

The upgrade authority has complete control over the contract's funds, as they can arbitrarily change the behaviour of the contract.

The upgrade authority is secured by a 2/3 multisig which is managed by the team.

Pool Owner

The pool owner has limited control over the parameters of the pool they control. They can change swap fees (though it is constrained to stay within a range of 0.0001% - 1% for stable swap and 0.01% - 2.5% for weighted swap), as well as being able to pause and unpause the pool.

Vault Admin

The vault admin has control over the vault parameters. This is limited to the identity of the vault beneficiary, the beneficiary fee percentage and to (un)pausing the protocol.

However, importantly, for stable swap pools that use the vault, they can also initiate an amplification factor change. This means they can extract funds from the pool, simply by letting the pool oscillate between being constant-product and constant-sum. By swapping almost all tokens in the constant-sum scenario, switching to constant-product and then swapping until the pool is balanced again, they can reduce the overall number of tokens in the pool.

As a (simplified) example, consider a two-token pool with 5 tokens of type A and B. In this example we ignore restrictions on the amplification factor set out by the contract, however the same principle applies. Initially, the admin sets the amplification factor as high as possible, such that the pool is (nearly) constant-sum. By swapping 4 type A tokens into type B, the pool will now have (approximately) 9 tokens of type A and 1 of type B. The admin now immediately sets A to 0 such that it is (nearly) a constant-product AMM. They do this with a 1 second ramp. They can now trade 2 tokens of type B into 6 tokens of type A, leaving the pool with 3 tokens of type A and 3 tokens of type B. The admin has profited by 2 tokens of type A and 2 tokens of type B.

The vault admin is secured by the same multisig as the upgrade authority. The team plans to transfer this authority to a governance system later on.

5 | Findings

This section outlines all of our findings. They are classified into one of five severity levels, detailed in Appendix C. In addition to these findings, Neodyme delivered the stabble team a list of nit-picks and additional notes which are not part of this report.

All findings are listed in Table 4 and further described in the following sections.

Identifier	Name	Severity	Status
ND-STB-MD-01	Token2022 Fees can lead to accounting error	MEDIUM	Resolved
ND-STB-LO-01	Transfer Hooks prevent token transfers	LOW	Resolved
ND-STB-IN-01	Beneficiary fees can end up in unexpected accounts	INFORMATIONAL	Resolved
ND-STB-IN-02	Invariant field is never updated but always printed in CLI	INFORMATIONAL	Acknowledged

Table 4: Findings

[ND-STB-MD-01] Token2022 Fees can lead to accounting error

Severity	Impact	Affected Component	Status
MEDIUM	Accounting Error	Stable Swap, Weighted Swap	Resolved

Description

We found that during deposits and swaps with token2022 mints, it is not checked if the mints have transfer fees enabled. This can lead to accounting problems because the number of tokens ending up in the pool does not correspond to the amounts the pool is thinking it holds. This leads to the theoretical amount (tracked by the pool account) being lower and lower than the actual amount (tracked by the vault token account) over time.

Relevant Code

deposit.rs, lines 160-204

```
impl<'info> Deposit<'info> {
    fn transfer_to_vault(
        &mut self,
        amount: u64,
        token_index: usize,
        user_account: &AccountInfo<'info>,
        vault_account: &AccountInfo<'info>,
        mint: &AccountInfo<'info>,
    ) -> Result<u64> {
                    let amount_in = self.pool.calc_rounded_amount(amount, token_index).unwrap();
                let balance_in = self.pool.calc_wrapped_amount(amount_in, token_index).unwrap();
        // add token balances
        self.pool.tokens[token_index].balance += balance_in;

        let token_program = if mint.owner.key() == Token::id() {
            self.token_program.to_account_info()
        } else {
            self.token_program_2022.to_account_info()
        };

        // check associated token account for vault
                                    let expected_vault_account_key = associated_token::get_associated_token_address_with_program_id(
            self.vault_authority.key,
            mint.key,
            token_program.key,
        );
        assert_eq!(expected_vault_account_key, vault_account.key());
```

```
28
29          transfer_checked(
30              CpiContext::new(
31                  token_program.to_account_info(),
32                  TransferChecked {
33                      from: user_account.to_account_info(),
34                      mint: mint.to_account_info(),
35                      to: vault_account.to_account_info(),
36                      authority: self.user.to_account_info(),
37                  },
38              ),
39              amount_in,
40              self.pool.tokens[token_index].decimals,
41          )?;
42
43          Ok(balance_in)
44      }
45  }
```

Resolution

In commit d1279b2a009a4dd817cde299977d9f3d70f64e7b, the stabble team added logic to deal with the case of token fees and correctly incorporate them in transfer amounts.

Neodyme flagged a newly introduced incorrect behaviour related to double-charging of fees, which was subsequently fixed in a4adb1414fa7bf350ee954f95bdea0e645683992. The final fix was again verified by Neodyme.

[ND-STB-LO-01] Transfer Hooks prevent token transfers

Severity	Impact	Affected Component	Status
LOW	DOS	Stable Swap, Weighted Swap	Resolved

Description

We found that the current implementation doesn't prevent mints with the TransferHook extension from being used in pools, while the program transfer doesn't add the additional account necessary for the hooks to work. Because transfer hooks can be enabled and disabled by the TransferHook authority at will, a pool can be used for some time and stop working for the token that just enabled the TransferHook. This impacts all locations where tokens are transferred.

Relevant Code

deposit.rs, lines 161-203

```
impl<'info> Deposit<'info> {
    fn transfer_to_vault(
        &mut self,
        amount: u64,
        token_index: usize,
        user_account: &AccountInfo<'info>,
        vault_account: &AccountInfo<'info>,
        mint: &AccountInfo<'info>,
    ) -> Result<u64> {
                    let amount_in = self.pool.calc_rounded_amount(amount, token_index).unwrap();
                let balance_in = self.pool.calc_wrapped_amount(amount_in, token_index).unwrap();
        // add token balances
        self.pool.tokens[token_index].balance += balance_in;

        let token_program = if mint.owner.key() == Token::id() {
            self.token_program.to_account_info()
        } else {
            self.token_program_2022.to_account_info()
        };

        // check associated token account for vault
                                    let expected_vault_account_key = associated_token::get_associated_token_address_with_program_id(
            self.vault_authority.key,
            mint.key,
            token_program.key,
        );
        assert_eq!(expected_vault_account_key, vault_account.key());
```

```
28
29          transfer_checked(
30              CpiContext::new(
31                  token_program.to_account_info(),
32                  TransferChecked {
33                      from: user_account.to_account_info(),
34                      mint: mint.to_account_info(),
35                      to: vault_account.to_account_info(),
36                      authority: self.user.to_account_info(),
37                  },
38              ),
39              amount_in,
40              self.pool.tokens[token_index].decimals,
41          )?;
42
43          Ok(balance_in)
44      }
45  }
```

Resolution

The stabble team added a whitelist of tokens which can become part of the pool in commit d1279b2a009a4dd817cde299977d9f3d70f64e7b. Neodyme verified the changes.

[ND-STB-IN-01] Beneficiary fees can end up in unexpected accounts

Severity	Impact	Affected Component	Status
INFORMATIONAL	Beneficiary trolling	Beneficiary Fee Disbursement	Resolved

Description

Swap instructions of both stable and weighted pools give a cut of the fees to the vault beneficiary. However, the token account to which the funds are sent is only checked to belong to the vault beneficiary, not to be the "expected" token account. Anyone can create a token account owned by the beneficiary and transact their swaps such that the fee cut is deposited into this wallet. The beneficiary will still have access to the funds, but they will have to go hunting for token accounts owned by them to withdraw them.

This behaviour exists in both the Withdraw and Withdraw2 instructions of the vault contract.

Relevant Code

withdraw.rs, lines 12-30

```
// ...
if beneficiary_amount > 0 {
    let beneficiary_token_account = ctx.accounts.beneficiary_token.as_ref().unwrap();
    assert_eq!(beneficiary_token_account.owner, ctx.accounts.vault.beneficiary);
    transfer_checked(
        CpiContext::new(
            ctx.accounts.token_program.to_account_info(),
            TransferChecked {
                from: ctx.accounts.vault_token.to_account_info(),
                mint: ctx.accounts.mint.to_account_info(),
                to: beneficiary_token_account.to_account_info(),
                authority: ctx.accounts.vault_authority.to_account_info(),
            },
        )
        .with_signer(&[signer_seed]),
        beneficiary_amount,
        ctx.accounts.mint.decimals,
    )?;
}
```

Resolution

The stabble team introduced a fix with commit fc8ade1f5aa245aa64ba42d5ad1600ff47de10cb, adding the necessary constraint.

[ND-STB-IN-02] Invariant field is never updated but always printed in CLI

Severity	Impact	Affected Component	Status
INFORMATIONAL	Faulty Tracking	Weighted Swap	Acknowledged

Description

In weighted swaps, `pool.invariant` is only set during the initial liquidity deposit. However, the pool's mathematical invariant changes as more deposits and withdrawals are done. These changes are not reflected in the `pool.invariant` field.

Note that this field is not used otherwise in the on-chain contract. However, it is printed via `console.log("Invariant:", pool.invariant);` in the CLI for every swap. This may confuse CLI users.

Relevant Code

deposit.rs, lines 25-112

```
// LP amount
let amount_out = if ctx.accounts.mint.supply == 0 {

    // ...

    // initial liquidity
    let invariant = weighted_math::calc_invariant(
        // ...
    )
    .unwrap();

    ctx.accounts.pool.invariant = invariant * num_tokens as u64;
    ctx.accounts.pool.invariant
} else {
    // do_join
    if num_tokens == 1 {
        // ...

        weighted_math::calc_pool_token_out_given_exact_token_in(
            // ...
        ).unwrap()

        // NO update of pool.invariant

    } else {
        weighted_math::calc_pool_token_out_given_exact_tokens_in(
            // ...
        ).unwrap()
```

```
29
30          // NO update of pool.invariant
31
32      }
33  };
```

Resolution

The stabble team acknowledged the finding but state that the CLI will be used internally only and that the invariant is stored to see the initial invariant value.

A | 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 that Neodyme hosts the most qualified auditors for Solana programs. 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.

All of our team members have a background in competitive hacking. During such hacking competitions, called CTFs, 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 in 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.

B | Methodology

We are not checklist auditors.

In fact, we pride ourselves on that. We adapt our approach to each audit, investing considerable time into understanding the program upfront and exploring its expected behavior, 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 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 here.

Select Common Vulnerabilities

Our most common findings are still specific to Solana itself. Among these are vulnerabilities such as the ones listed below:

- Insufficient validation, such as:
  - Missing ownership checks
  - Missing signer checks
  - Signed invocation of unverified programs
  - Account confusions
  - Missing freeze authority checks
  - Insufficient SPL account verification
  - Dangerous user-controlled bumps
  - Insufficient Anchor account linkage
- Account reinitialization vulnerabilities
- Account creation DoS
- Redeployment with cross-instance confusion
- Missing rent exemption assertion
- Casting truncation
- Arithmetic over- or underflows
- Numerical precision and rounding errors
- Anchor pitfalls, such as accounts not being reloaded
- Non-unique seeds
- Issues arising from CPI recursion
- Log truncation vulnerabilities
- Vulnerabilities specific to integration of Token Extensions, for example unexpected external token hook calls

Apart from such Solana-specific findings, some of the most common vulnerabilities relate to the general logical structure of the contract. Specifically, such findings may be:

- Errors in business logic
- Mismatches between contract logic and project specifications
- General denial-of-service attacks
- Sybil attacks
- Incorrect usage of on-chain randomness
- Contract-specific low-level vulnerabilities, such as incorrect account memory management
- Vulnerability to economic attacks
- Allowing front-running or sandwiching attacks

Miscellaneous other findings are also routinely checked for, among them:

- Unsafe design decisions that might lead to vulnerabilities being introduced in the future
    - Additionally, any findings related to code consistency and cleanliness
- Rug pull mechanisms or hidden backdoors

Often, we also examine the authority structure of a contract, investigating their security as well as the impact on contract operations should they be compromised.

Over the years, we have found hundreds of high and critical severity findings, many of which are highly nontrivial and do not fall into the strict categories above. This is why our approach has always gone way beyond simply checking for common vulnerabilities. We believe that the only way to truly secure a program is a deep and tailored exploration that covers all aspects of a program, from small low-level bugs to complex logical vulnerabilities.

C | Vulnerability Severity Rating

We use the following guideline to classify the severity of vulnerabilities. Note that we assess each vulnerability on an individual basis and may deviate from these guidelines in cases where it is well-founded. In such cases, we always provide an explanation.

Severity	Description
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.
INFORMATIONAL	Bugs or inconsistencies that have little to no security impact, but are still noteworthy.

Additionally, we often provide the client with a list of nit-picks, i.e. findings whose severity lies below Informational. In general, these findings are not part of the report.

Neodyme AG

Dirnismaning 55
Halle 13
85748 Garching
Germany

E-Mail: contact@neodyme.io

https://neodyme.io