Security Peer Review – MetaDAO AMM

Lead Auditor: Robert Reith

Second Auditor: Jasper Slusallek

Administrative Lead: Thomas Lambertz

July 10th 2024

Table of Contents

Executive Summary

Neodyme reviewed **Metadao’** on-chain MetaDAO AMM program during April and May of 2024. The scope of this review included implementation security, overall design and architecture. The reviewers found that Metadao’s MetaDAO AMM program comprised a clean modular design and high code quality. According to Neodymes Rating Classification, **0 critical or high vulnerabilities** and only **4 medium or low severity issues** were found. The number of findings identified throughout the review, grouped by severity, can be seen in Figure 1.

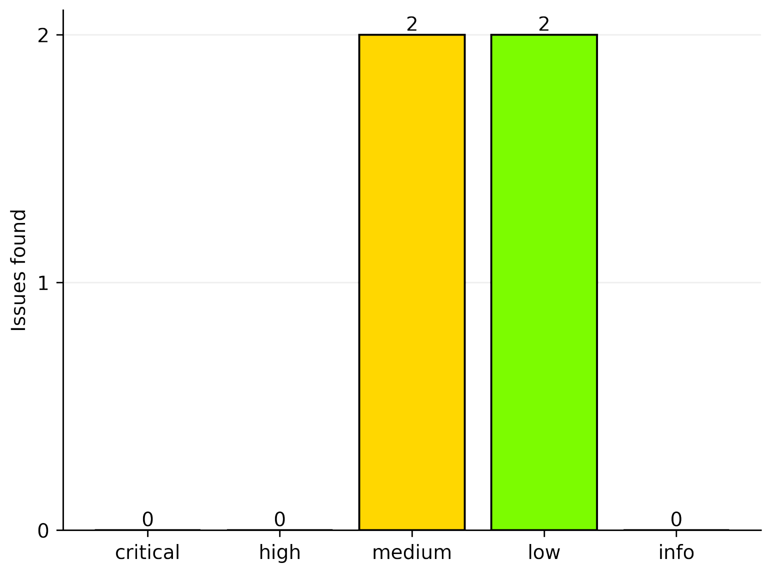

Figure 1: Overview of Findings

All findings were reported to the Metadao developers and addressed promptly. The security fixes were verified for completeness by Neodyme. In addition to these findings, Neodyme delivered the Metadao team a list of nit-picks and additional notes that are not part of this report.

1 | Introduction

During April and May 2024, Metadao commissioned Neodyme to conduct a detailed security analysis of Metadao's on-chain MetaDAO AMM program.

Two senior auditors performed the review between the 24th of April and the 1st of June. This report details all findings from this time span.

The review mainly focused on the contract's technical security, but also considered its design and architecture. After the introduction, this report details the review'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 Metadao's work. The Metadao team always responded quickly and **competent** to findings of any kind. Their **deep knowledge and passion** about innovation in governance programs was apparent during all stages of the cooperation. Evidently, Metadao 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 mostly on the well-established Anchor framework.

Findings Summary

During the review, **4 security-relevant** and **0 informational** findings were identified. Metadao remediated all of those findings before the protocol's launch.

In total, the review revealed:

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

issues.

The highest severity findings addresses two vulnerabilities where metadata could be manipulated, and proposals could have settled other proposals.

All findings are detailed in section Findings.

2 | Scope

The contract review'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/metaDAOproject/futarchy, is in scope of this review. However, third-party dependencies are not. Metadao only relies on the well-established Anchor library, the Anchor SPL library and solana-security-txt. Relevant source code revisions are:

- `61a6f2bc600b72c07af2892e2c74509755d132d4` • Start of the review
- `feda41360d4c934307616d854391bc81d3e8bdcf` • Last reviewed revision

3 | Project Overview

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

Functionality

The Metadao is a project that implements the concept of a DAO controlled by Futarchy[1]. Like a regular DAO, it implements proposals of on-chain actions that have to be decided on. However, instead of voting on proposals, participants *trade them*.

This works by creating two markets for each proposal: A **pass** market, and a **fail** market. On each market, derivative tokens are traded. For instance, a proposal would have a **pMETA**/**pUSDC** market, and a **fMETA**/**fUSDC** market. The market that closes after a certain time with a higher time-weighted average price (including a margin) wins. If the pass market wins, the proposed action will be executed. If the fail market wins, the proposal is discarded. At the end, only the winning derivative token can be converted back to regular Meta or USDC. The losing token will become unredeemable and thus worthless.

A previous version of the Metadao was implemented using Openbook, an on-chain CLOB. This led to liquidity problems for the on-chain markets. This new version uses a simple constant-product automated market maker instead.

A typical user flow would look like this:

1. Someone creates a new proposal. They propose to pay a developer for creating a new feature for the Metadao. They initialize the proposal, two AMMs, one for each market, and two conditional vaults, one for USDC and one for Meta. They also provide the initial liquidity for the markets, which is locked until the proposal ends.
2. Another user comes along and converts some USDC and META to proposal-specific pUSDC, fUSDC, pMETA, and fMETA using the conditional vaults.
3. That user thinks the proposal is good. They decide to buy pMETA for pUSDC on the pass market, and sell fMETA for fUSDC on the fail market.
4. After some time, the proposal succeeds, and is executed. The developer receives their payment.
5. The trader can now convert their pMETA to META and pUSDC to USDC. fMETA and fUSDC is burned.
6. The proposer can withdraw their liquidity.

[1] https://en.wikipedia.org/wiki/Futarchy

On-Chain Data and Accounts

The on-chain MetaDAO contract is split into three main programs, and a migration program:

- **AMM**: Generic CPMM program
- **Autocrat**: Manage DAOs and proposals
- **Conditional Vault**: Convert tokens into pass- and fail- derivatives and back
- **Autocrat Migrator**: Migrate legacy MetaDAO to the new program

Each of these programs maintains multiple accounts to keep track of its state and data. In particular, the following data need to be tracked:

- Configuration parameters of DAOs
- Proposal details and state
- State of each AMM
- State of Conditional Vaults, and details about their tokens

This data is represented on-chain as follows.

The **Autocrat program** saves the state of each DAO, including their treasury, the DAO token and quote token details, and parameters such as liquidity minimums, twap parameters, and proposal duration. It also saves a proposal counter. Proposals are their own Accounts, saving data such as description, proposer information, the proposed instruction, and references to the AMMs and conditional vaults.

The **AMM program** simply tracks the TWAP, pool sizes, creation time and mint information.

The **Conditional Vault** tracks the state of the vault, some references to the underlying and derivative token mints, and information on which account may settile the vault.

PDAs

The programs use multiple PDAs, which are seeded as follows:

Derived from Autocrat:

- Proposal: `"proposal"+ proposer.key + nonce`
- DAO Treasury: `dao.key`

Derived from AMM:

- AMM: `"amm__"+ base_mint.key + quote_mint.key`
- AMM LP Token Mint: `"amm_lp_mint"+ amm.key`

Derived from Conditional Vault:

- Conditional Vault: `"conditional_vault"+ settlement_authority.key + underlying_mint .key`
- Pass Mint: `"conditional_on_finalize_mint"+ conditional_vault.key`
- Fail Mint: `"conditional_on_revert_mint"+ conditional_vault.key`

Funds

Finally, the contract's funds are stored in the following accounts:

- the Treasury PDA, containing any funds the DAO holds
- the Conditional Vault PDA, escrowing any user deposited funds while they are converted to conditional tokens

Fees and Rent

Currently, only the AMM charges a trading fee which goes to Liquidity Providers. All other functions do not charge fees. In the current version, rents are not reclaimable.

Instructions

The contract has a total of 4 programs with a combined 17 instructions, which we briefly summarize here for completeness.

Table 1: Instructions with Descriptions

Instruction	Category	Summary
AMM		
add_liquidity	User	Add Liquidity and receive LP Tokens
crank_that_twap	Crank	Update the TWAP
create_amm	User	Create a new AMM
remove_liquidity	User	Turn LP tokens into Tokens
swap	User	Swap one token for another in a pool
Autocrat		
execute_proposal	User	Execute a proposal

Instruction	Category	Summary
finalize_proposal	User	Settle a proposal
initialize_dao	User	Create a new DAO
update_dao	Admin	Update DAO parameters
autocrat_migrator		
multi_transfer2	User	Migrate 2 DAO Assets
multi_transfer4	User	Migrate 4 DAO Assets
conditional_vault		
add_metadata_to_conditional_tokens	User	Add metadata to conditional tokens
initialize_conditional_vault	User	Create a new conditional vault
mint_conditional_tokens	User	Swap tokens for conditional tokens
merge_conditional_tokens	User	Swap a pair of cond.- tokens to underlying tokens
redeem_conditional_tokens _for_underlying_tokens	User	Swap settled cond.- tokens for underlying tokens
settle_conditional_vault	Admin	Settle vault as pass or fail

Authority Structure and Off-Chain Components

Upgrade Authority

The upgrage authority of the Metadao program will be delegated to a multisig consisting of trusted individuals.

Admin Authority

To our knowledge, there will be no explicit overall admin authority for this program. Each DAO will have their own administration.

Authorities

The Proposal PDA is used as the `settlement_authority` for conditional vaults. The `conditional_vault` is in turn used as the mint authority for conditional tokens. The AMM is used as the mint authority for the AMM LP token.

Off-Chain Components

The relevant off-chain components are the crank bot which just calls the `amm::crank_that_twap` instruction to update TWAP data, and the web application which users shall use to trade Metadao proposals. It is important that the trading interface's integrity and accuracy is well implemented so that users trade on accurate information.

Security Features

The programs implement rigorous checks in every instruction to ensure a consistent on-chain state. Duplicate checks of similar accounts are done in loops to ensure consistency. To ensure independance of DAOs created with this program, there is no central authority that could interfere with DAOs, beside the upgrade authority.

4 | 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 Metadao team a list of nit-picks and additional notes which are not part of this report.

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

Table 2: Findings

Name	Severity
[ND-MTD1-M1] Metadata Manipulation	Medium
[ND-MTD1-M2] Proposals can settle other proposals	Medium
[ND-MTD1-L1] Unchecked truncating casts	Low
[ND-MTD1-L2] AMM TWAP keeps accumulating after proposal end	Low

ND-MTD1-M1 – Metadata Manipulation

Severity	Impact	Affected Component	Status
Medium	Frontend Discrepancy	Conditional Vault	Fixed

Description

The metadata of conditional tokens are editable by anyone. In particular, this includes the URL of the metadata from which most frontend will fetch the information to be shown in user interfaces. In particular, the underlying token metadata account isn't verified and can be swapped out. In combination with UTF-8 special chars, attackers may even overwrite the prepended "f" or "p" characters.

Location

- /programs/conditional_vault/src/instructions/add_metadata_to_conditional_tokens.rs#L21

Relevant Code

```rust
#[derive(Accounts)]
pub struct AddMetadataToConditionalTokens<'info> {
    #[account(mut)]
    pub payer: Signer<'info>,
    #[account(
        mut,
        has_one = underlying_token_mint,
    )]
    pub vault: Account<'info, ConditionalVault>,
    #[account(mut)]
    pub underlying_token_mint: Account<'info, Mint>,
    pub underlying_token_metadata: Account<'info, MetadataAccount>,
    #[account(
        mut,
        mint::authority = vault,
        mint::freeze_authority = vault,
        mint::decimals = underlying_token_mint.decimals
    )]
    pub conditional_on_finalize_token_mint: Account<'info, Mint>,
    #[account(
        mut,
        mint::authority = vault,
        mint::freeze_authority = vault,
        mint::decimals = underlying_token_mint.decimals
    )]
    pub conditional_on_revert_token_mint: Account<'info, Mint>,
    /// CHECK: verified via cpi into token metadata
    #[account(mut)]
    pub conditional_on_finalize_token_metadata: AccountInfo<'info>,
    /// CHECK: verified via cpi into token metadata
    #[account(mut)]
    pub conditional_on_revert_token_metadata: AccountInfo<'info>,
    pub token_metadata_program: Program<'info, Metadata>,
    pub system_program: Program<'info, System>,
    pub rent: Sysvar<'info, Rent>,
}
```

Mitigation Suggestion

We recommend approaching this either by making metadata addition a privileged instruction that only trusted parties can call, or potentially deploying a custom metadata server that will generate a correct metadata url, and enforcing use of that server. Another solution would be to make metadata completely on-chain and deterministic, so that anyone can generate them, but not influence their content.

Remediation

The issue has been fixed by making metadata addition a privileged instruction as recommended. The fix was amended in commit `feda41360d4c934307616d854391bc81d3e8bdcf`

ND-MTD1-M2 – Proposals can settle other proposals

Severity	Impact	Affected Component	Status
Medium	State Inconsistency	AMM	Fixed

Description

We found that the `settlement_authority` for the conditional vaults of proposals is set to the `dao.treasury` account, which is the same account for all proposals of the same DAO. This means that one could create two proposals and make the first proposal settle the second proposal's conditional vault to either accepted or revoked. This would completely circumvent the condition that for every proposal the winning market shall decide the conditional vault. Users who are not aware of the first proposal and only trade the second proposal may lose funds this because they didn't know this.

Location

- /programs/autocrat/src/instructions/finalize_proposal.rs#L150-L155
- /programs/autocrat/src/instructions/execute_proposal.rs#L30-L36

Relevant Code

```
// in finalize_proposal.rs:
        for vault in [base_vault.to_account_info(), quote_vault.
            to_account_info()] {
            let vault_program = vault_program.to_account_info();
            let cpi_accounts = SettleConditionalVault {
                settlement_authority: treasury.to_account_info(),
                vault,
            };
            let cpi_ctx = CpiContext::new(vault_program, cpi_accounts).
                with_signer(signer);
            conditional_vault::cpi::settle_conditional_vault(cpi_ctx,
                new_vault_state)?;
// in execute_proposal.rs:
        for acc in svm_instruction.accounts.iter_mut() {
            if acc.pubkey == dao.treasury.key() {
                acc.is_signer = true;
            }
        }

        solana_program::program::invoke_signed(&svm_instruction, ctx.
            remaining_accounts, signer)?;
            }
```

Mitigation Suggestion

We recommend using a per-proposal settlement authority. For this we would turn the proposal account into a PDA, which then could be used as such an authority

Remediation

This issue has been addressed by turning proposal accounts into PDAs, and changing the vault settlement authority to the proposal account. The changes have been added in commit 964e483a61e520edb783e33d66dc7582551f2dcc

ND-MTD1-L1 – Unchecked truncating casts

Severity	Impact	Affected Component	Status
Low	Math Errors	AMM	Fixed

Description

There are a few unchecked truncating casts in the AMM that wouldn't be caught by overflow checks. However only the first one seems to be theoretically attackable, and just in very extreme scenarios.

Location

- /programs/amm/src/instructions/add_liquidity.rs#L65
- /programs/amm/src/instructions/add_liquidity.rs#L68
- /programs/amm/src/state/amm.rs#L119
- /programs/amm/src/state/amm.rs#L154
- /programs/amm/src/state/amm.rs#L159

Relevant Code

```
let base_amount = (((quote_amount as u128 * base_reserve) /
    quote_reserve) + 1) as u64;
// [...]
((quote_amount as u128 * total_lp_supply as u128) / quote_reserve) as
    u64;
// [...]
let output_amount = (numerator / denominator) as u64;
// [...]
((lp_tokens as u128 * self.base_amount as u128) / lp_total_supply as
    u128) as u64
// [...]
((lp_tokens as u128 * self.quote_amount as u128) / lp_total_supply as
    u128) as u64
```

Mitigation Suggestion

We recommend using safe downcasts, such as `u64::try_from(n)` or checking the values with an if clause like `if n > std::u64::MAX`

Remediation

The first three casts have been turned into safe downcasts in commit 6dd88d5078e6c76d434534a8a221fe9eff7c7672. The latter two casts should never overflow, and a comment has been added to explain this.

ND-MTD1-L2 – AMM TWAP keeps accumulating after proposal end

Severity	Impact	Affected Component	Status
Low	Market Manipulation	AMM	Accepted

Description

We found that TWAP accumulation keeps going even after the proposal time has ended. Theoretically, if the proposal isn't resolved immediately, the proposal could switch from fail to pass or the other way around after the proposal end date. This would happen if there's a significant price move at the end. This might break assumptions of users that the markets close at a certain time.

Location

- /programs/amm/src/state/amm.rs#L199-L201

Relevant Code

```
if current_slot < oracle.last_updated_slot + ONE_MINUTE_IN_SLOTS {
        return None;
}
```

Mitigation Suggestion

We recommend stopping accumulation with the proposal end. The `update_twap` function takes the current slot already and could easily deduce the correct time to stop accumulation.

Remediation

For the time being, the MetaDAO has accepted the risk of this issue.

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 up front, 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