Pocket Program Audit Report

08.06.2024

– Mad Shield

Contents

01- Introduction

ArmadaFi engaged Mad Shield to audit two programs including the `spl-token-staking` and the `clp-vault` program. The audit included a full analysis of the core functionality of both programs, with subsequent reviews following major releases. We performed multiple intermittent cycles of intense security analysis of the codebase to ensure the highest safety standards.

Our team was well aware of the original algorithm of the `spl-token-staking` program. The re-implementation of the program in the Anchor framework is very well-written, clear and concise which improves significantly upon the first implementation. The Mad Shield team is glad to announce that we discovered **0 vulnerabilities** in the `spl-token-staking` program.

For the `clp-vault` program, we analyzed the program's security measures over the time period between November 2023 to January 2024. In summary, we uncovered a total of **2 vulnerabilities**; 1 Medium and 1 Low-severity vulnerability.

A detailed walk-through of our findings is provided in this report. We communicated all our findings with proper mitigations through our private channels to the ArmadaFi development team. We are glad to confirm that all vulnerabilities reported during the audit have been addressed and resolved accordingly. This report documents the audit process, the assessment methodology, and declares this program to be secure to the best of our knowledge.

1.1 Overview

ArmadaFi provides end-to-end tooling to power the Solana SPL token ecosystem. It is fully customizable to support demands tailored to the needs of all solana-based projects.

This audit covers the security for two of the main programs that underlie the Armada's fleet of products.

- **SPL Token Staking –** A novel solution to allow users to easily setup and interact with tokenized stake and reward pools for use in SPL governance without making unnecessary compromises between voting power and collected rewards.
- **Concentrated Liquidity Pool (CLP) vaults –** A solution built to allow users to create and manage market making positions (MM) in a specific Whirlpool, where 2 assets trade against each other.

This report presents a comprehensive analysis of the implemented changes and their security implications. The main purpose is to identify and remedy program vulnerabilities.

1.1 Account Structure

The main account in the program is the Pocket account which is depicted in the following figure. This account has a size of 151 + the corresponding size of the FeeAccount and Constituent times the number of elements stored in the respective vectors.

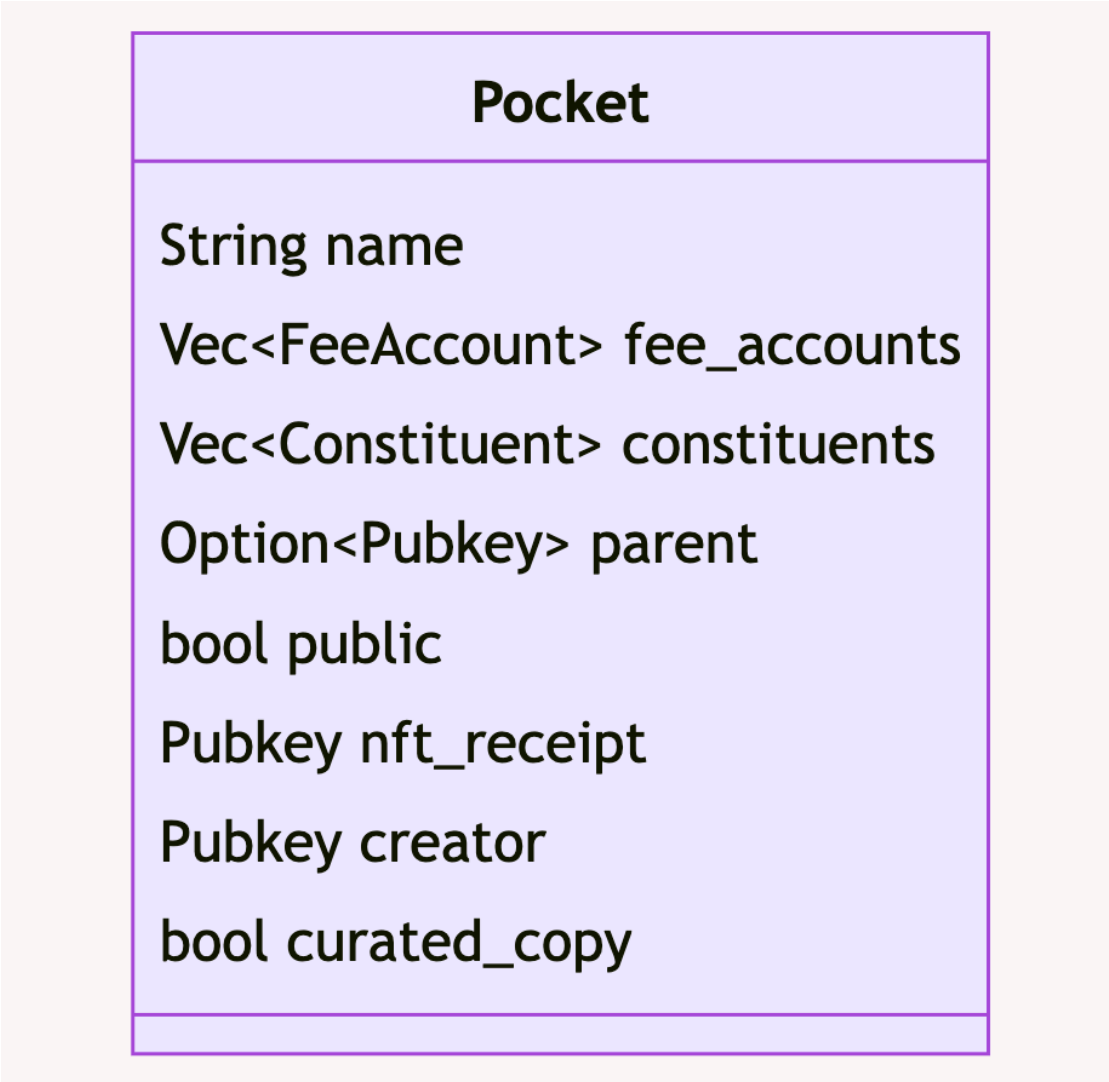
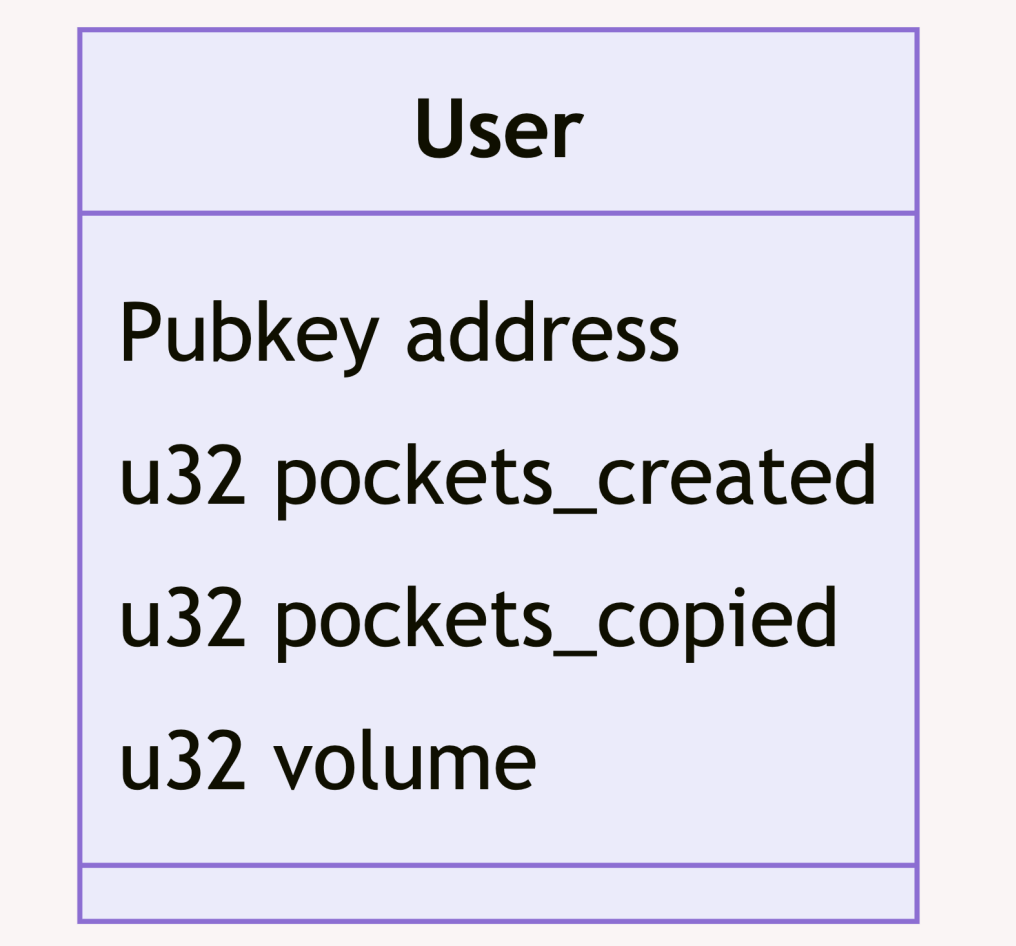


Fig 1: Program structure and account interactions for `Pocket_Defi`

Of importance in this account, is that each pocket can be copied by any other user which is to replicate the strategies aka the allocations used by the creator of the original pocket. In both cases, the pocket creator gets issued a nft_receipt representing their ownership over the pocket, granting them the authority to modify the pocket parameters.

The second account in the program is the User account which is depicted in the following figure. This account is mainly used to calculate the fee_tier a user is assigned.



To calculate the `user_tier`, the program embeds a list of mint addresses that are eligible for points calculated based on the user's activity which is a combination of pockets created, pockets copied and user's volume.

1.2. Overview

The following were the key points of the system that were highlighted during the audit.

1. collect_fees Instruction:

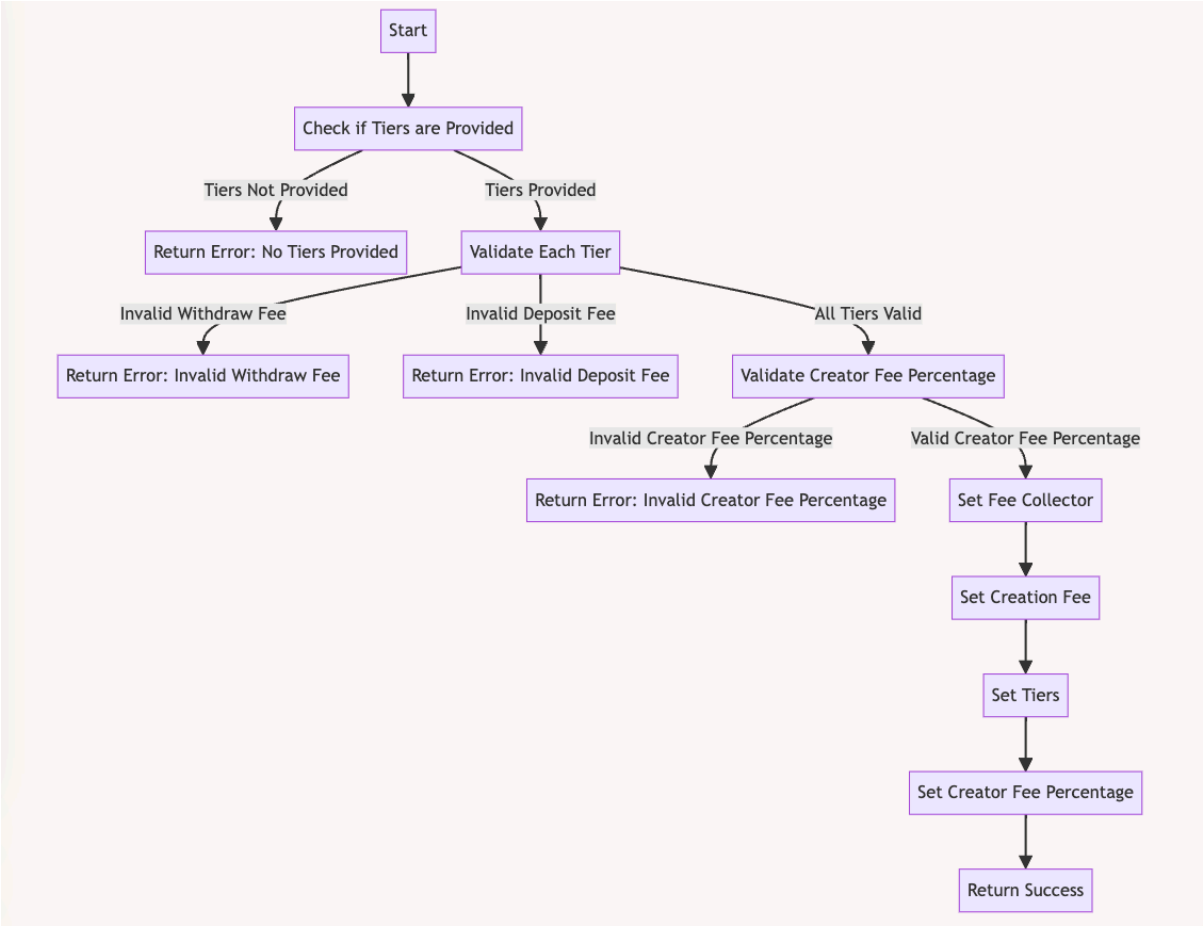
- Functionality: Collects fees from a Pocket account.
- With Parent Pockets Enabled: Fees are distributed between the current pocket and its parent, ensuring hierarchical consistency.
- Without Parent Pockets: All fees go directly to the fee collector.
- Consistency: Ensures parent pockets and associated token accounts are correctly validated and updated.

2. Rebalancing Using Deposit and Withdrawal:

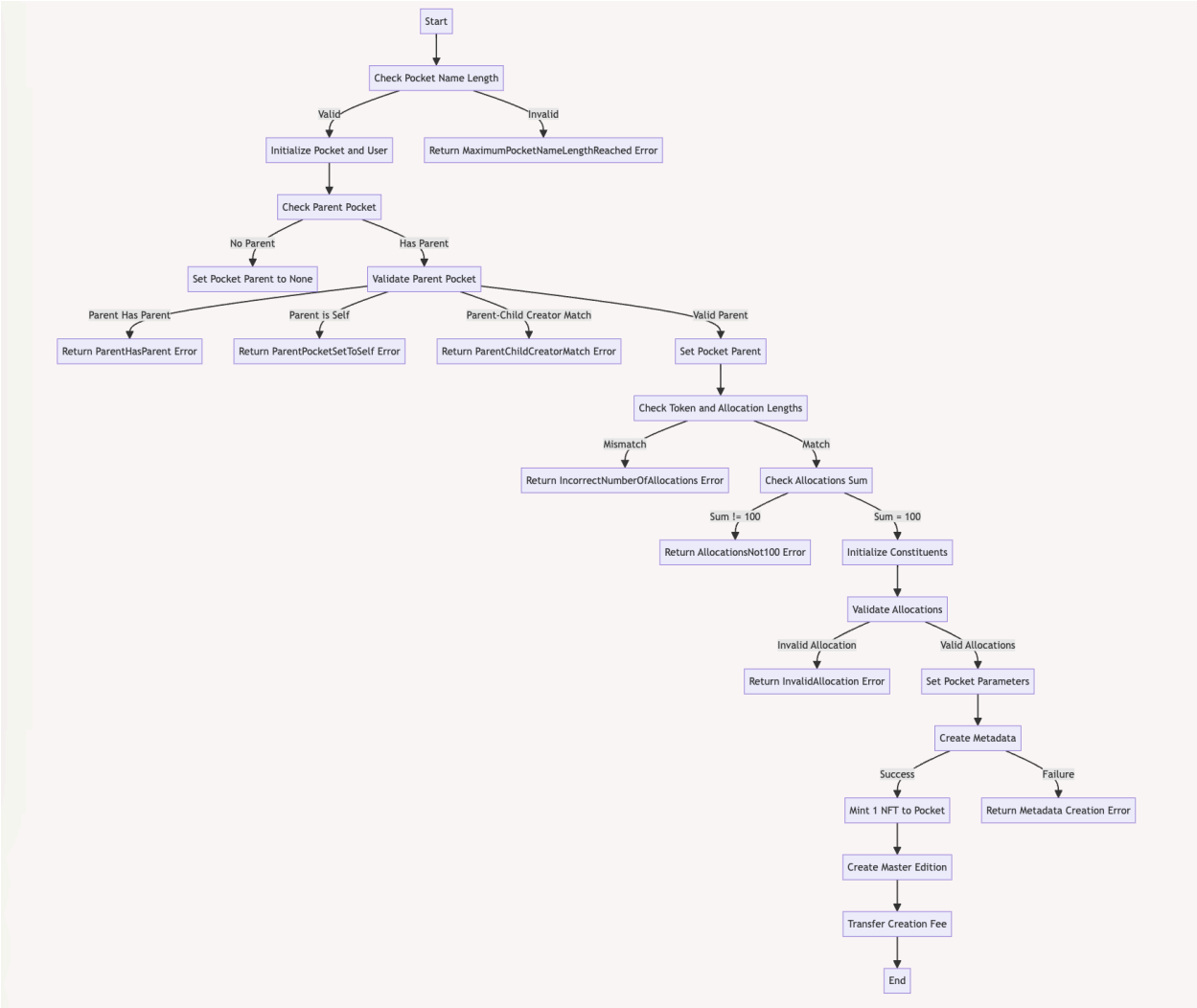
- Consistency Requirement: Ensures asset ratios remain consistent, especially when a pocket has a parent.
- Deposit Function:
    - Updates constituent balances and allocations.
    - Adjusts allocations to maintain a 100% total.
    - Distributes fees according to user tier and fee settings.
- Withdraw Function:
    - Verifies and updates constituent balances.
    - Adjusts remaining allocations when a constituent is removed.
    - Distributes fees similarly to the deposit function.

1.3 Instructions

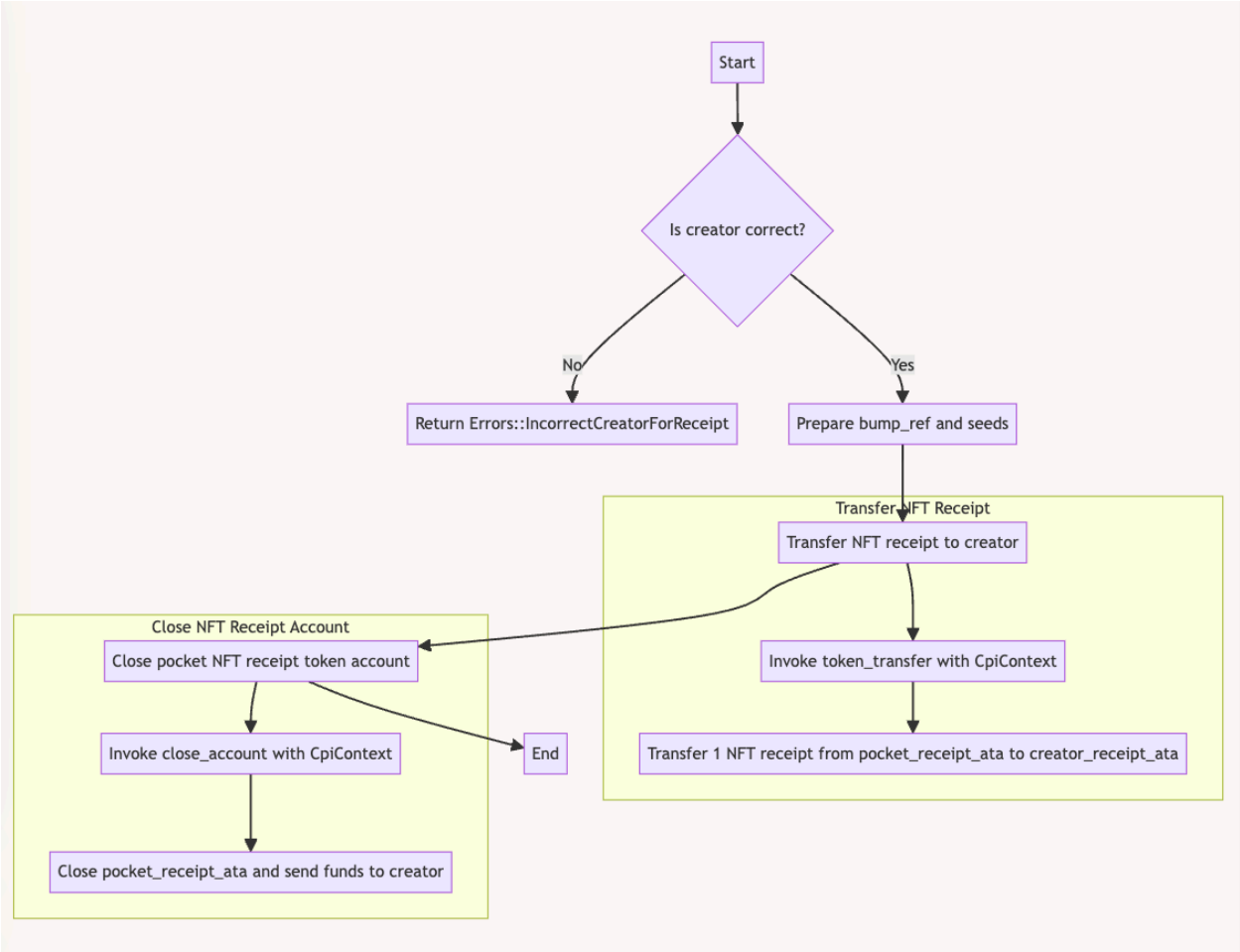
1. initialize_fee_settings / update_fee_settings



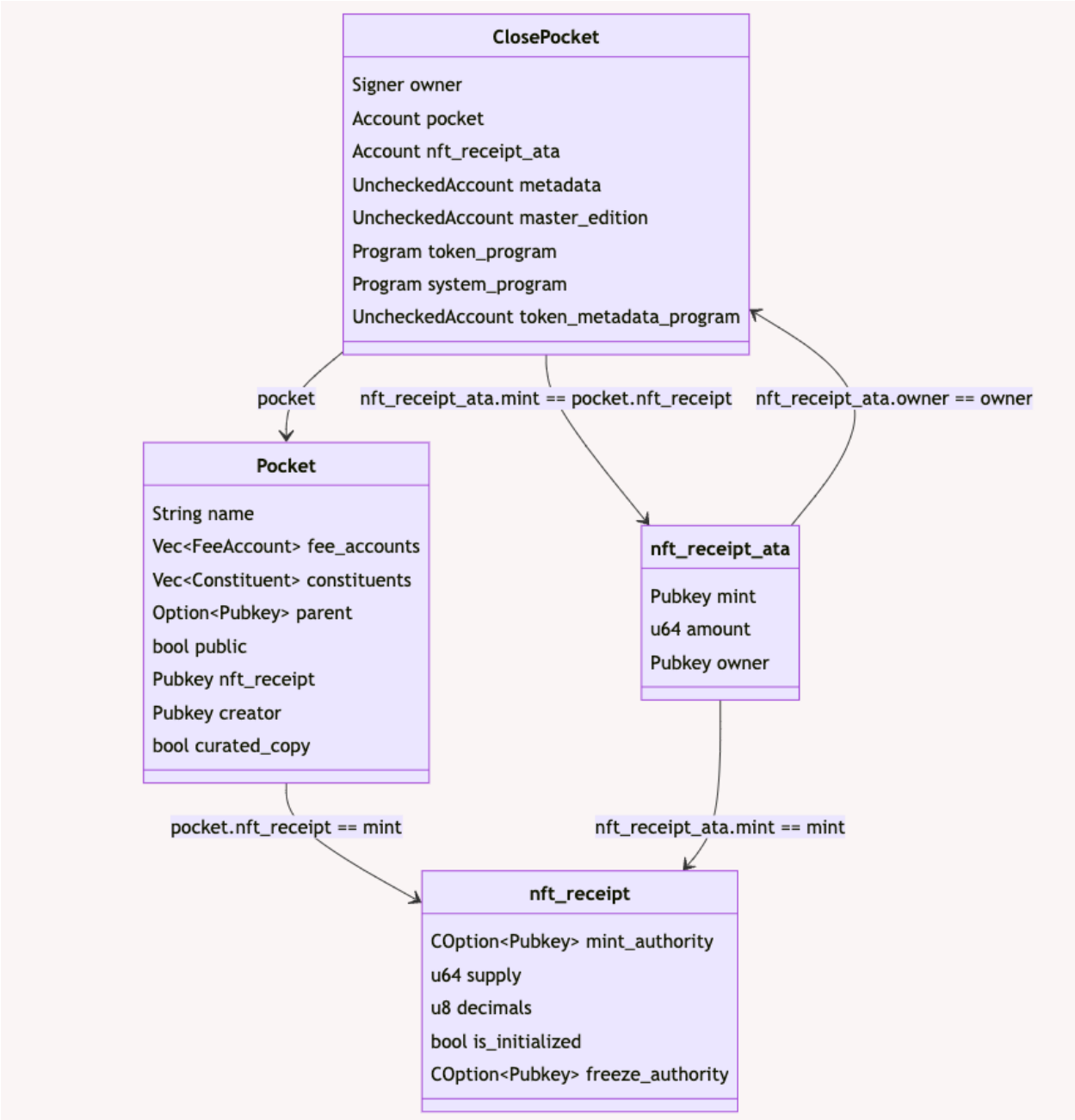
2. Create_pocket

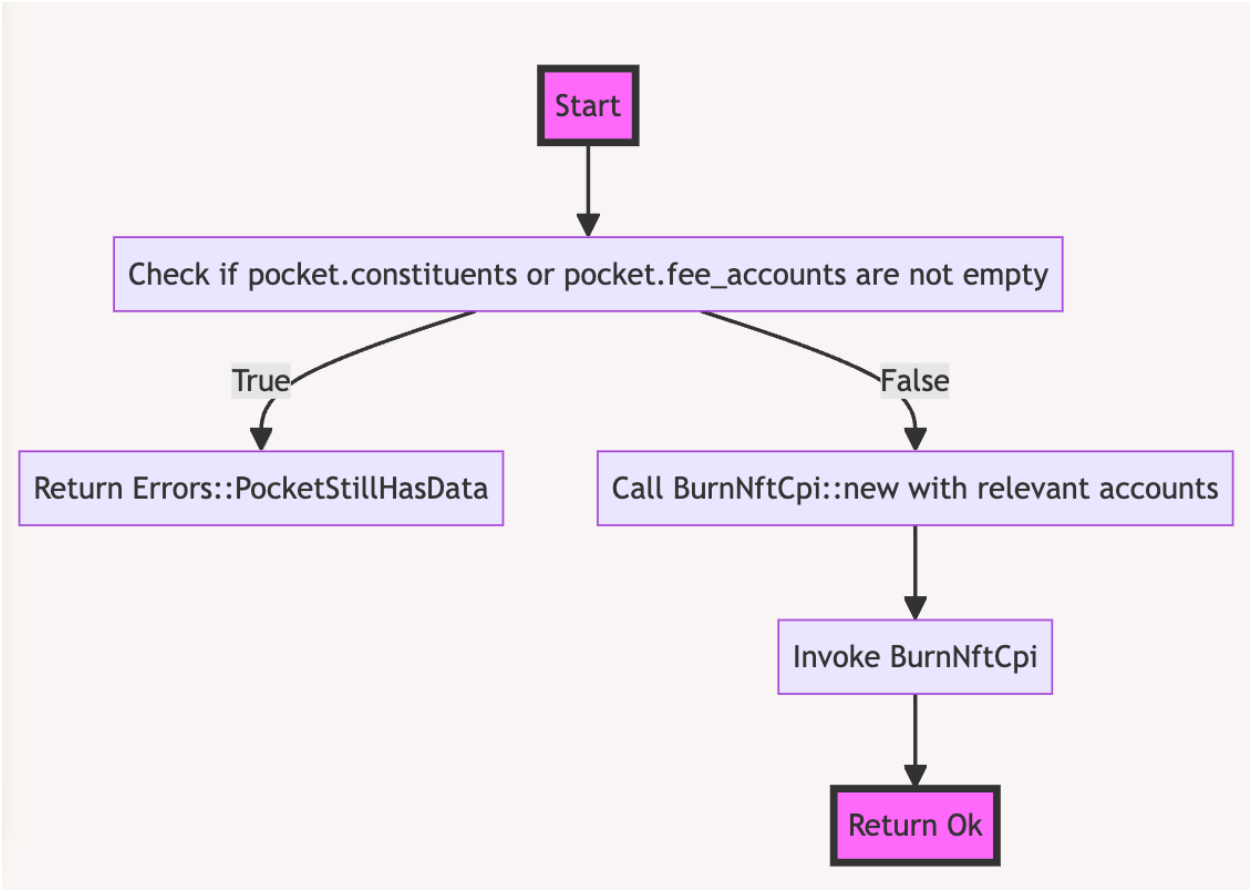


3. Send_receipt



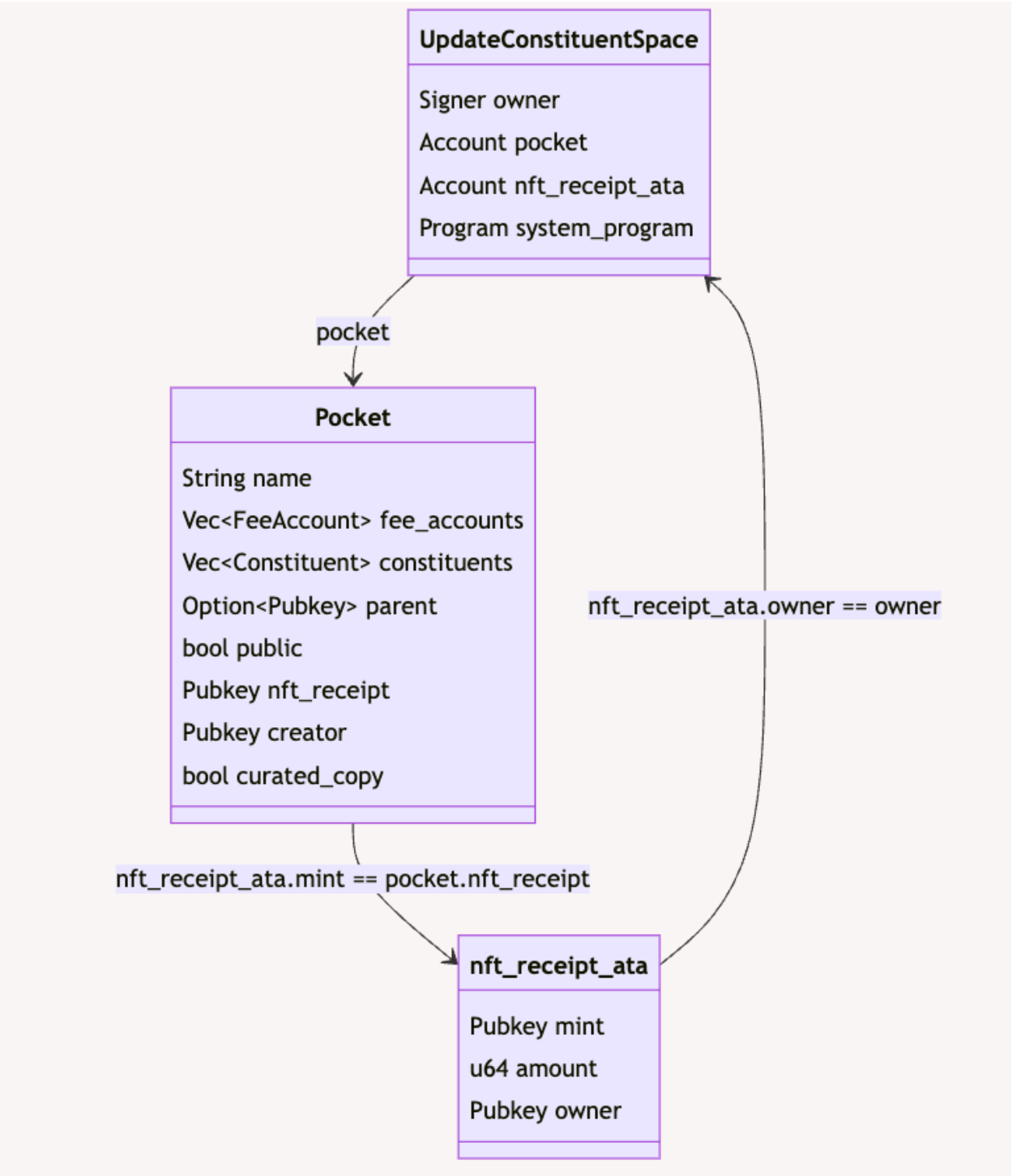
4. close_pocket





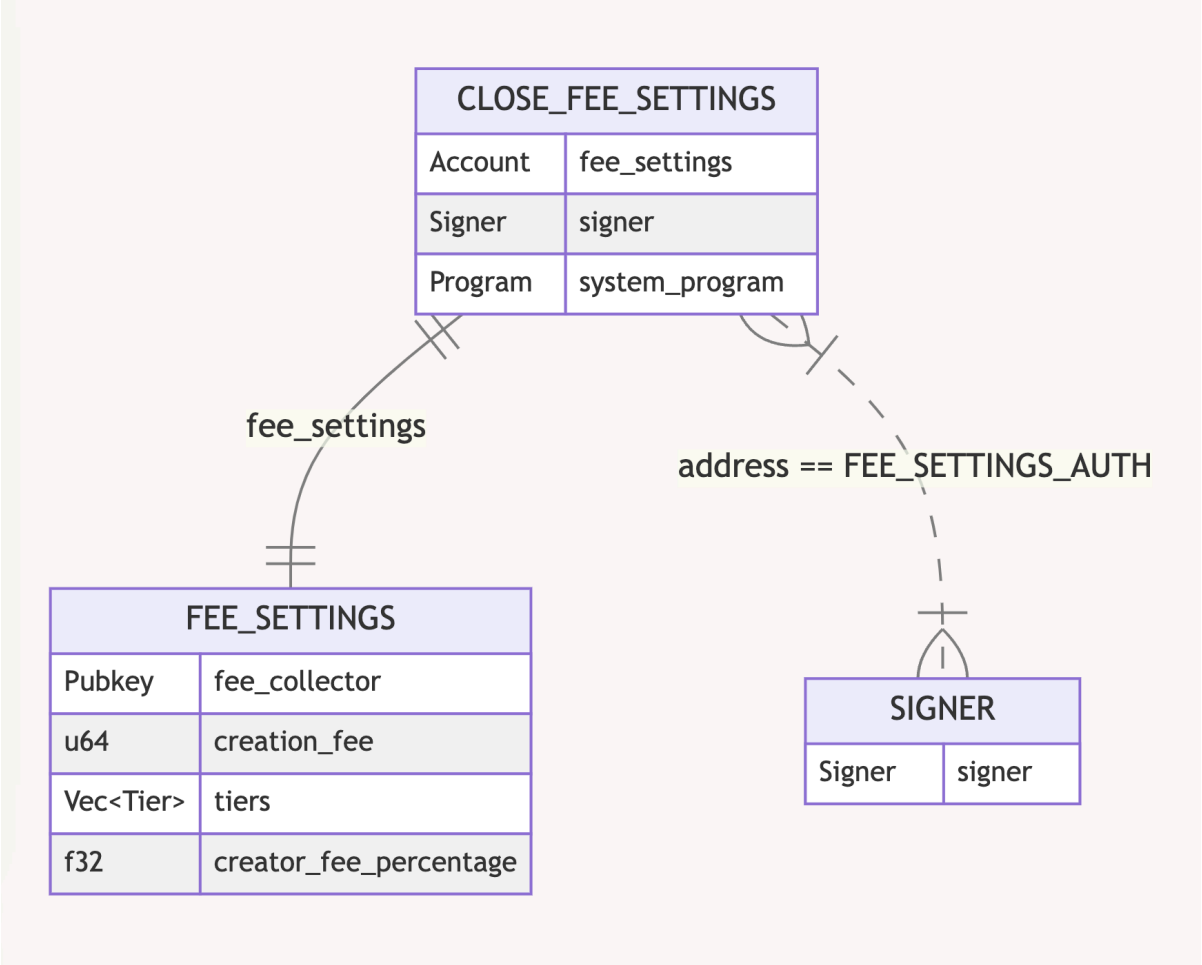
5. update_constituent_space

This instruction is similar to the close_pocket instruction, the only requirement is to ensure only the NFT receipt owner is able to update the space of the account which is verified below.

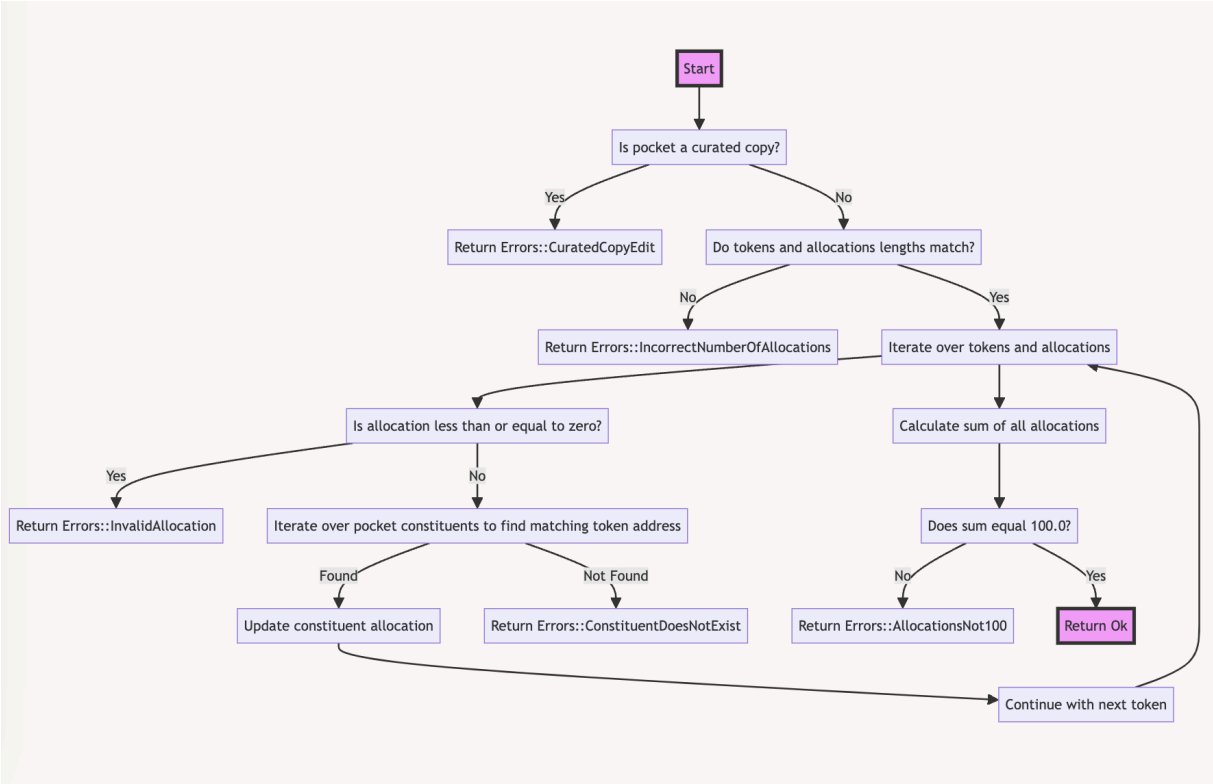

UpdateConstituentSpace
Signer owner
Account pocket
Account nft_receipt_ata
Program system_program
pocket
Pocket
String name
Vec<FeeAccount> fee_accounts
Vec<Constituent> constituents
Option<Pubkey> parent
bool public
Pubkey nft_receipt
Pubkey creator
bool curated_copy
nft_receipt_ata.owner == owner
nft_receipt_ata.mint == pocket.nft_receipt
nft_receipt_ata
Pubkey mint
u64 amount
Pubkey owner

6. close_fee_setting

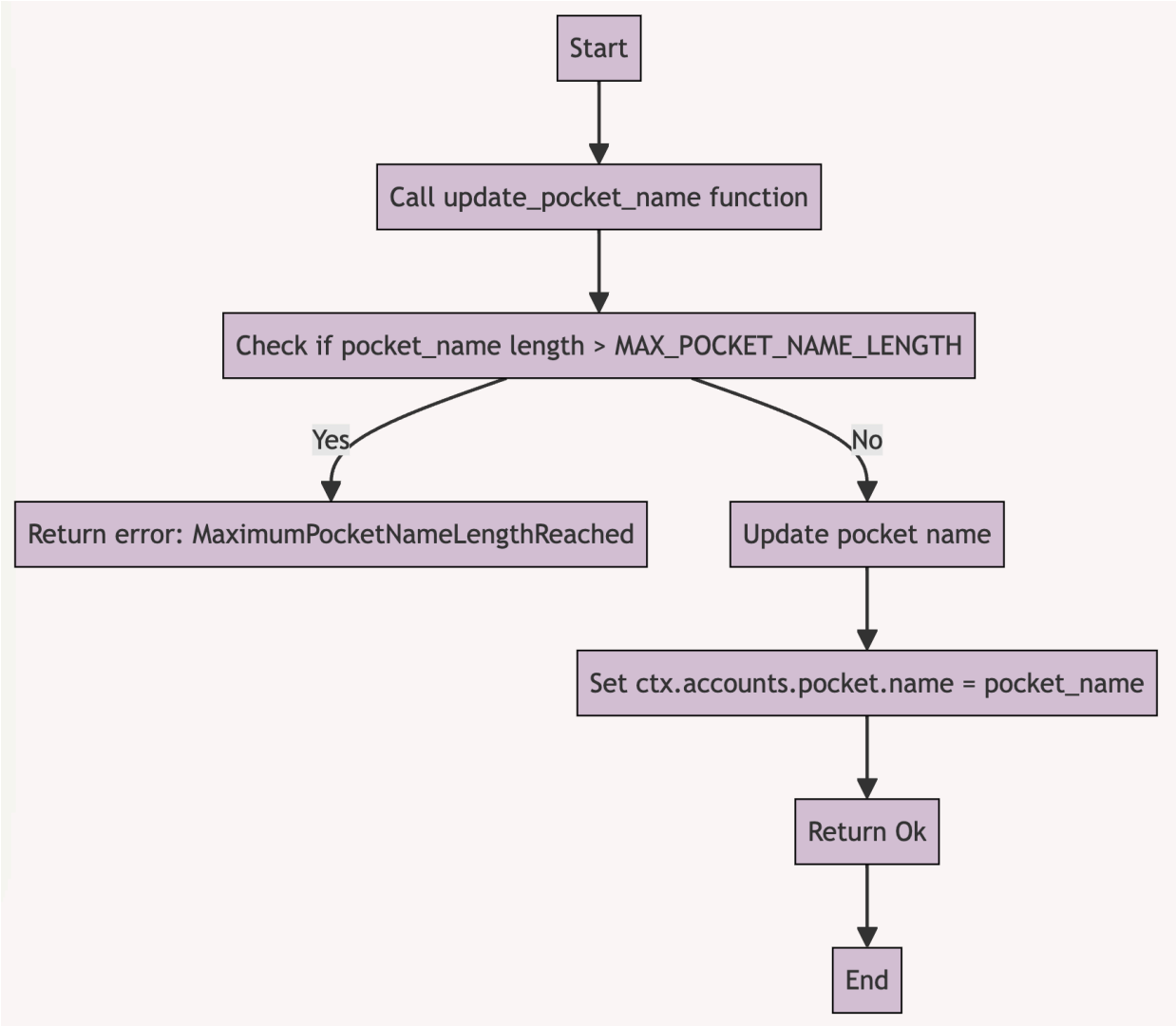
This instruction is similar to the close_pocket instruction, the only requirement is that the corresponding authority must be the `FEE_SETTINGS_AUTH` which is hardcoded in the codebase.

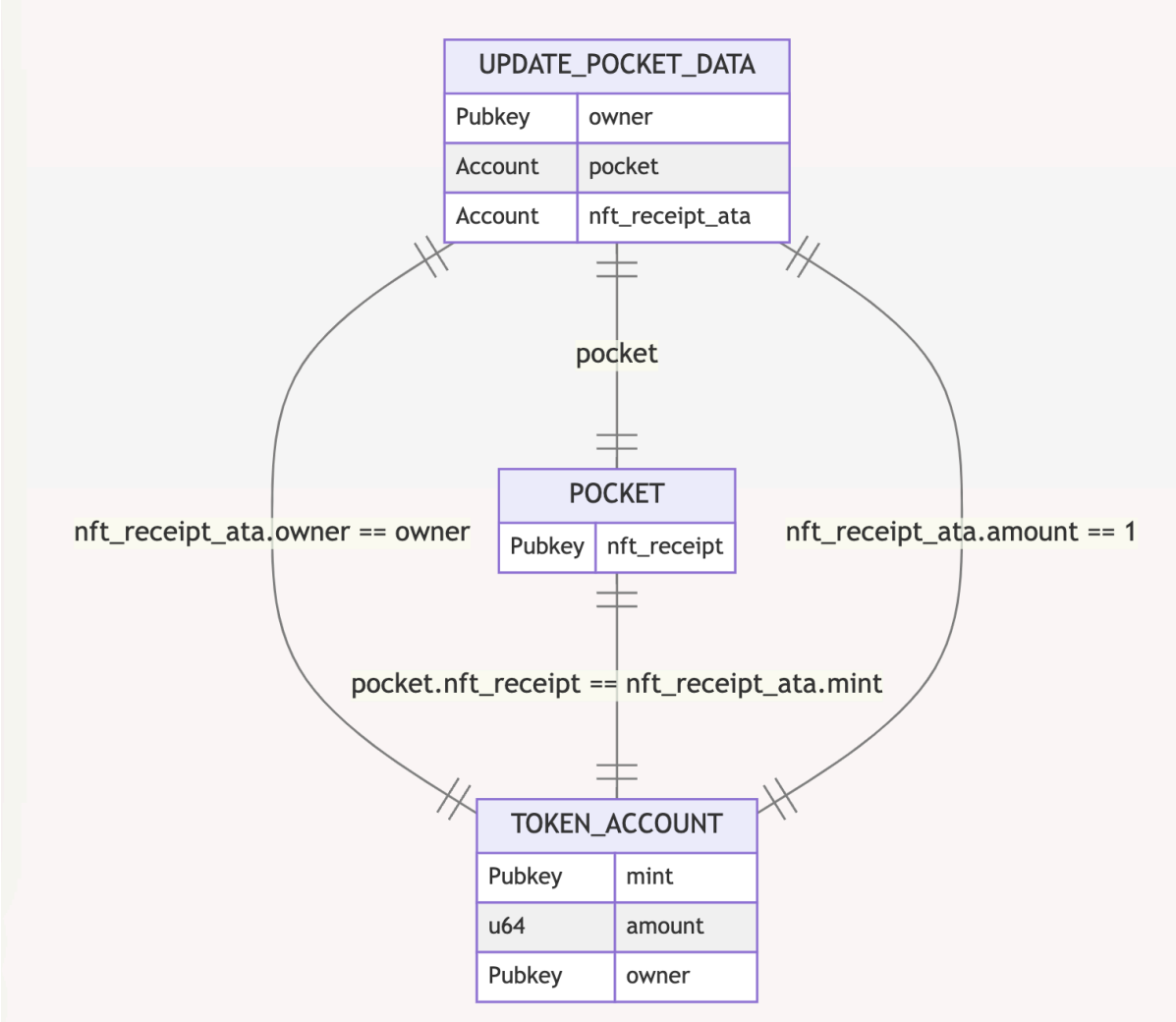


7. edit_allocations



8. Update_pocket_name

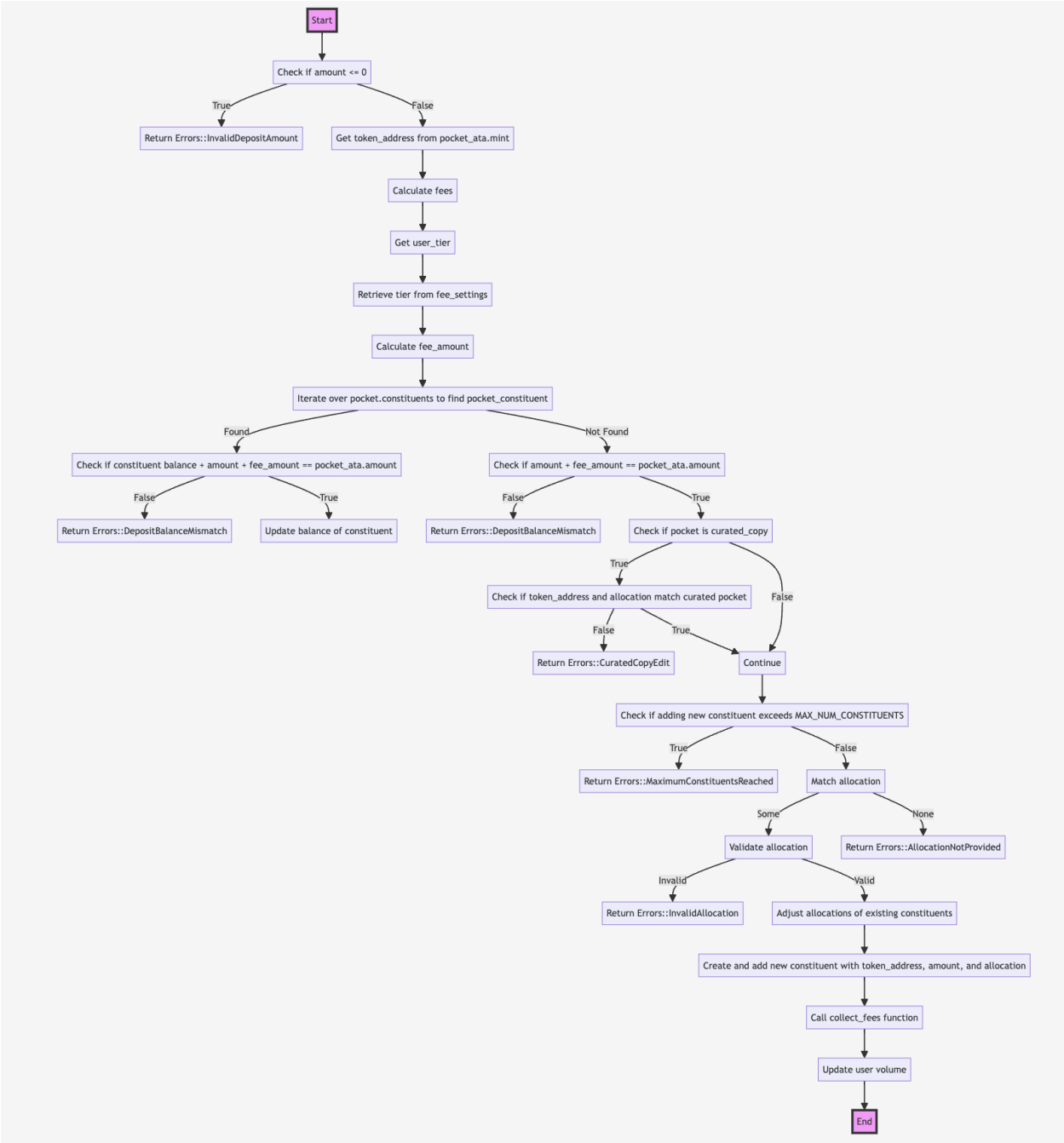




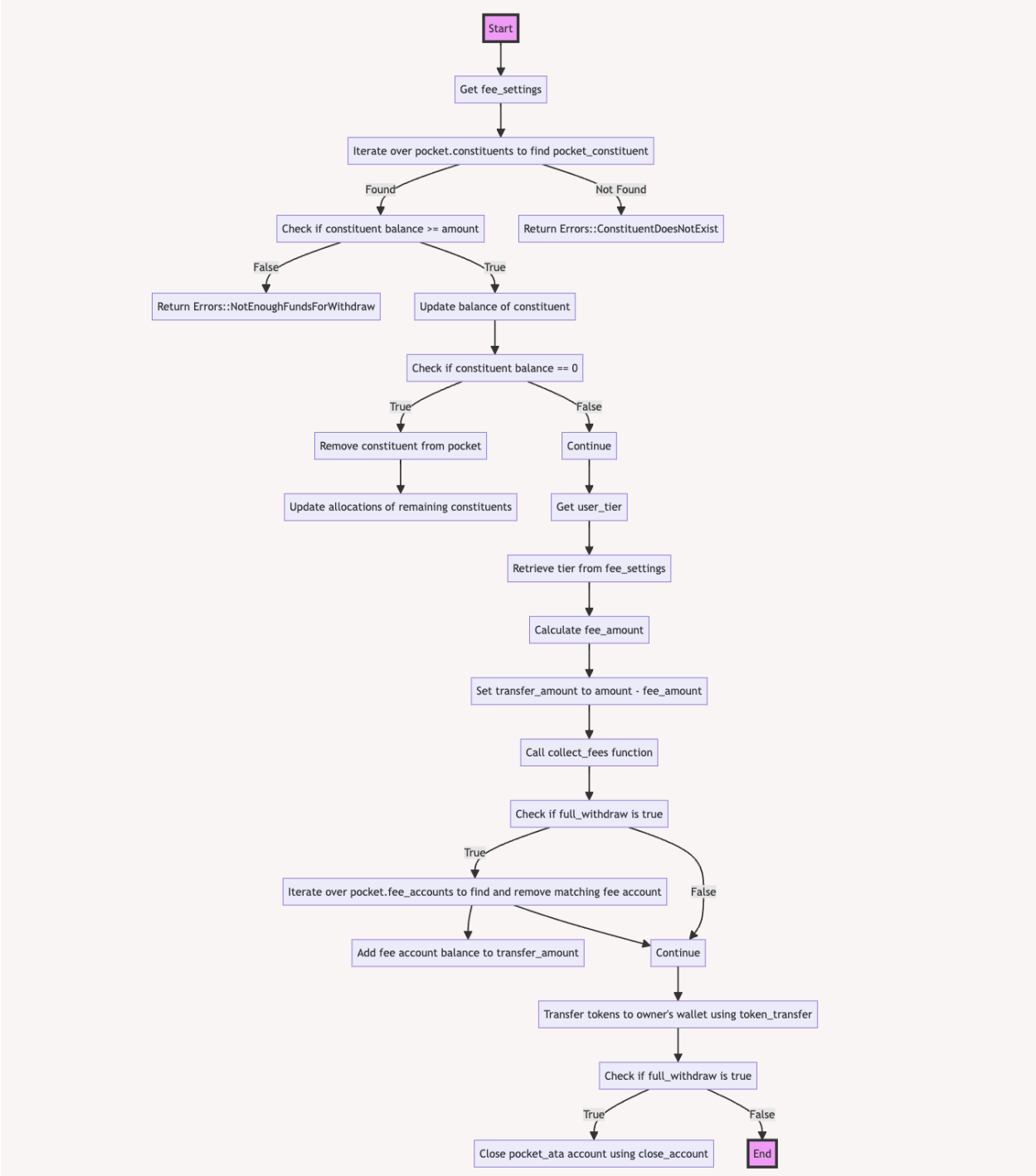
9. update_pocket_privacy

This instruction is similar to update_pocket_name and uses the same account structure constraint as `UpdatePocketData`.

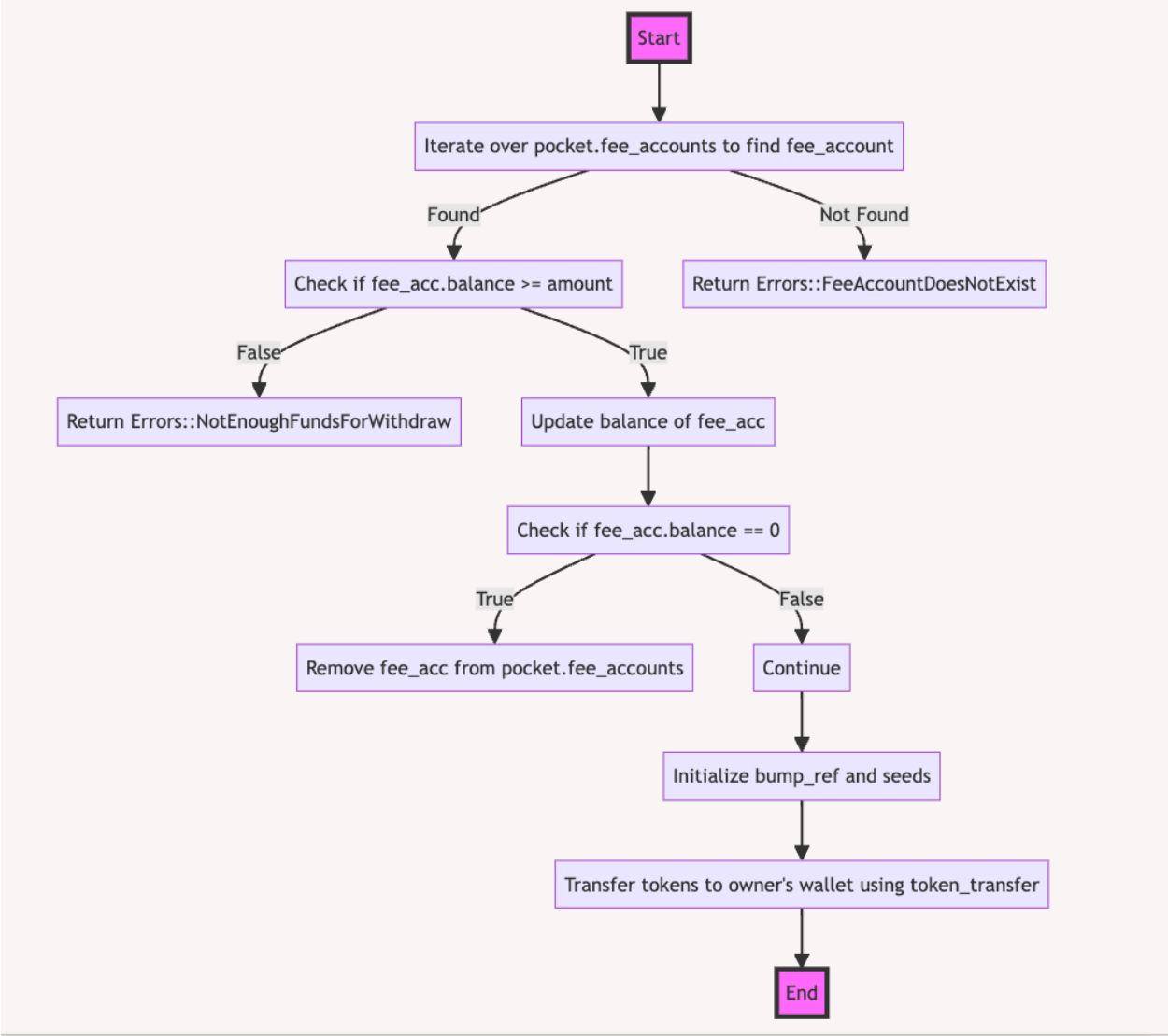
10. deposit


Start
Check if amount <= 0
True
False
Return Errors::InvalidDepositAmount
Get token_address from pocket_ata.mint
Calculate fees
Get user_tier
Retrieve tier from fee_settings
Calculate fee_amount
Iterate over pocket.constituents to find pocket_constituent
Found
Not Found
Check if constituent balance + amount + fee_amount == pocket_ata.amount
Check if amount + fee_amount == pocket_ata.amount
False
True
False
True
Return Errors::DepositBalanceMismatch
Update balance of constituent
Return Errors::DepositBalanceMismatch
Check if pocket is curated_copy
True
False
Check if token_address and allocation match curated pocket
False
True
Return Errors::CuratedCopyEdit
Continue
Check if adding new constituent exceeds MAX_NUM_CONSTITUENTS
True
False
Return Errors::MaximumConstituentsReached
Match allocation
Some
None
Validate allocation
Return Errors::AllocationNotProvided
Invalid
Valid
Return Errors::InvalidAllocation
Adjust allocations of existing constituents
Create and add new constituent with token_address, amount, and allocation
Call collect_fees function
Update user volume
End

11. withdraw


Start
Get fee_settings
Iterate over pocket.constituents to find pocket_constituent
Found
Not Found
Check if constituent balance >= amount
Return Errors::ConstituentDoesNotExist
False
True
Return Errors::NotEnoughFundsForWithdraw
Update balance of constituent
Check if constituent balance == 0
True
False
Remove constituent from pocket
Continue
Update allocations of remaining constituents
Get user_tier
Retrieve tier from fee_settings
Calculate fee_amount
Set transfer_amount to amount - fee_amount
Call collect_fees function
Check if full_withdraw is true
True
Iterate over pocket.fee_accounts to find and remove matching fee account
False
Add fee account balance to transfer_amount
Continue
Transfer tokens to owner's wallet using token_transfer
Check if full_withdraw is true
True
False
Close pocket_ata account using close_account
End

12. withdraw_fees



02. Scopes and Objectives

The main objectives of the audit are defined as:

- Minimizing the possible presence of critical vulnerabilities in the program. This would include detailed examination of the code and edge case scrutinization to find as many vulnerabilities.
- 2-way communication during the audit process. This included for Mad Shield to reach a perfect understanding of the design of `pocket-defi`.
- Provide clear and thorough explanations of all vulnerabilities discovered during the process with potential suggestions and recommendations for fixes and code improvements.
- Clear attention to the documentation of vulnerabilities with the eventual publication of a comprehensive audit report to the public audience for all stakeholders to understand the security state of the programs.

Pocket has delivered these programs to Mad Shield at the following Github repositories.

1. SPL Token Staking

Repository URL	https://github.com/JKronick/PocketAudit
Commit (start of audit)	9b680053078a40db254c2c219469f6523e3d2f01
Commit (end of audit)	6f362716afc66a3c393fd719b1dba7252af54452

The scope encompasses **pull requests** [#15, #16, #17] from the `spl-token-staking` program.

03 . Methodology

The Pocket DeFi team initiated a request for an audit of their program. Upon receiving this request, we began with a preliminary review of the source code to understand the program's specifications and scope. This initial assessment allowed us to identify potential vulnerabilities and areas that need closer scrutiny.

To facilitate a comprehensive understanding of the program's flow, we employed AI-generated analysis tools to create detailed program flow diagrams. These diagrams illustrated the various routes and execution paths of the program, enabling us to identify any missing constraints or potential vulnerabilities in the flow of execution for each instruction.

In addition to the program flow diagrams, we used similar visualization techniques to analyze account constraints. By mapping out the relationships between accounts, we could more easily validate the constraints and ensure they adhered to the specified requirements. This step was crucial in identifying any discrepancies or issues related to account permissions and interactions.

Throughout the audit process, we maintained constant communication with the Pocket DeFi development team. This ongoing dialogue ensured that both our auditing team and the development team had a clear and mutual understanding of the program's specifications and any findings we uncovered. This collaboration helped eliminate any miscommunications or misinterpretations from either side.

As we identified vulnerabilities and areas for improvement, we provided the Pocket DeFi team with detailed recommendations for patches and enhancements. Our suggestions were aimed at strengthening the security and efficiency of the program, ensuring it met industry standards and best practices.

By following this rigorous methodology, we aimed to deliver a thorough and effective audit, providing the Pocket DeFi team with actionable insights and ensuring the robustness of their program.

04 . Findings & Recommendations

Our severity classification system adheres to the criteria outlined here.

Severity Level	Exploitability	Potential Impact	Examples
Critical	Low to moderate difficulty, 3rd-party attacker	Irreparable financial harm	Direct theft of funds, permanent freezing of tokens/NFTs
High	High difficulty, external attacker or specific user interactions	Recoverable financial harm	Temporary freezing of assets
Medium	Unexpected behavior, potential for misuse	Limited to no financial harm, non-critical disruption	Escalation of non-sensitive privilege, program malfunctions, inefficient execution
Low	Implementation variance, uncommon scenarios	Zero financial implications, minor inconvenience	Program crashes in rare situations, parameter adjustments by authorized entities
Informational	N/A	Recommendations for improvement	Design enhancements, best practices, usability suggestions

In the following, we enumerate some of the findings and issues we discovered and explain their implications and corresponding resolutions.

Finding	Description	Severity Level
SHIELD_DP_01 [RESOLVED]	Allocation Calculation Error in Deposit Instruction	*Critical*
SHIELD_DP_02 [RESOLVED]	Repetitive Token Addresses in Constituents	*High*
SHIELD_DP_03 [RESOLVED]	External token transfers to pocket token accounts blocks deposits due to restrictive condition on the token balance	*High*
SHIELD_DP_04 [RESOLVED]	The pocket account size is not correctly calculated for reallocation	*Low*
SHIELD_DP_GR_05 [RESOLVED]	Using a Merkle Tree for PALS NFT ownership validation	*Informational*
SHIELD_DP_GR_06 [ACKNOWLEDGED]	Tokens could be pooled to facilitate easier rebalancing across copied and curated pockets	*Informational*

1. **SHIELD_DP_01 - Allocation Calculation Error in Deposit Instruction *[Critical]***

   In the deposit instruction, If a new constituent is added with an allocation of 100, the program does not return an error. Instead, the allocation of all other constituents becomes zero while the allocation of the new constituent is assigned to 100.

   Since the specification assumes that no allocations can be zero, This becomes more significant especially if there is a balance in the token accounts of other constituents. The balance gets lost if the last constituent is withdrawn first, since this causes a division by zero error during withdrawal of the remaining constituents balances.

   Recommended Fix

   Ensure allocations follow the constraint 0 < allocation < 100, explicitly excluding 100.

```
match allocation {
    Some(allocation) => {
        if
            allocation <= 0.0 ||
            allocation > 100.0 ||
            (allocation == 100.0 && ctx.accounts.pocket.constituents.len() > 0)

        {
            return Err(Errors::InvalidAllocation.into());
        }
        for constituent in &mut ctx.accounts.pocket.constituents {
            constituent.allocation = constituent.allocation * (1.0 - allocation / 100.0);
        }
        let allocations_sum: f32 = ctx.accounts.pocket.constituents.iter().map(|c| c.allocation).sum();
        let new_constituent = Constituent{
            token_address: token_address,
            balance: amount,
            allocation: 100.0 - allocations_sum
        };
        ctx.accounts.pocket.constituents.push(new_constituent);
    }
    None => {
        return Err(Errors::AllocationNotProvided.into());
    }
}
```

   The patch commit: [6dcef0ea]

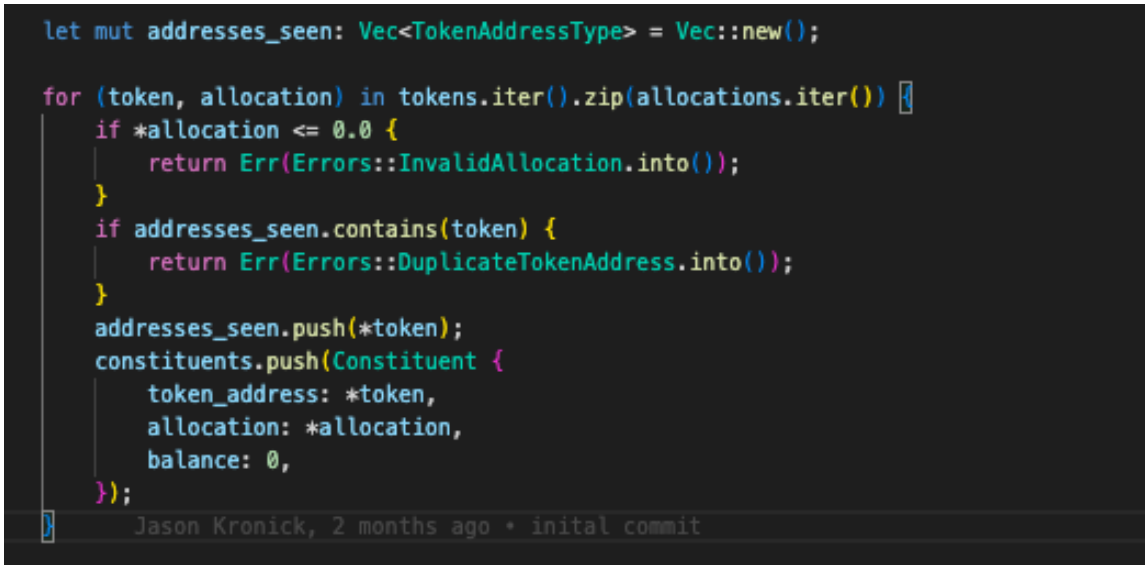
   Upon further inspection, we realized that the assumption during initial development was to rely on the `get_authority_type` utility function to validate the `token_record` account. This function determines the authority type with

2. **SHIELD_DP_02 - Repetitive Token Addresses in Constituents [High]**

In create_pocket instruction, the constituents are not checked to have distinct token addresses which means duplicate token addresses are possible to be present. This will cause an issue in the deposit and withdraw instructions where the balances might not be updated correctly or the balance might be credited or debited from the wrong token accounts.

Recommended Fix

Implement a check to prevent adding the same token address multiple times to a pocket.



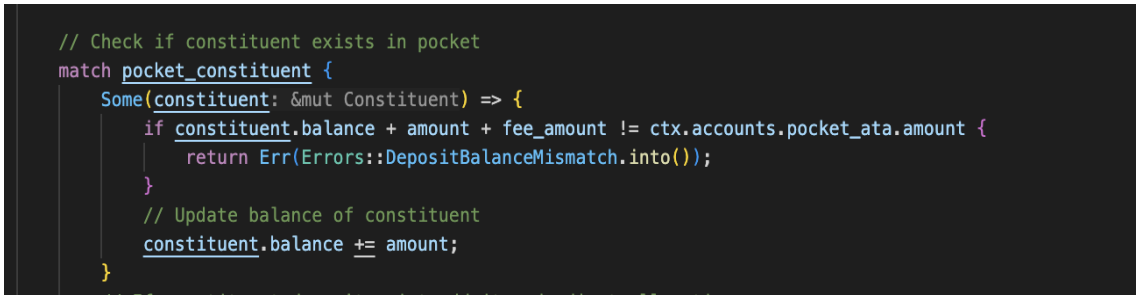
```
let mut addresses_seen: Vec<TokenAddressType> = Vec::new();

for (token, allocation) in tokens.iter().zip(allocations.iter()) {
    if *allocation <= 0.0 {
        return Err(Errors::InvalidAllocation.into());
    }
    if addresses_seen.contains(token) {
        return Err(Errors::DuplicateTokenAddress.into());
    }
    addresses_seen.push(*token);
    constituents.push(Constituent {
        token_address: *token,
        allocation: *allocation,
        balance: 0,
    });
}       Jason Kronick, 2 months ago • inital commit
```

The patch commit: [6dcef0ea]

3. **SHIELD_DP_03 - External token transfers to pocket token accounts blocks deposits due to restrictive condition on the token balance**

In the deposit instruction, the balance of the constituent struct in the pocket is matched against the balance of the deposit instruction to ensure that the correct amount is credited as follows.



```
// Check if constituent exists in pocket
match pocket_constituent {
    Some(constituent: &mut Constituent) => {
        if constituent.balance + amount + fee_amount != ctx.accounts.pocket_ata.amount {
            return Err(Errors::DepositBalanceMismatch.into());
        }
        // Update balance of constituent
        constituent.balance += amount;
    }
```

However, the check can be abused to DDOS the instruction if an attacker transfers a small amount of the token to the ATA.

Recommended Fix

Move the transfer inside the instruction

Patch commit: (to be resolved)

4. **SHIELD_DP_04 - The pocket account size is not correctly calculated for reallocation**

In the `UpdateConstituentSpace` the calculation of the realloc_size of the account is overestimating the actual size of the Pocket account by 30 bytes due to inconsistent definition of the struct during development.

This causes extra rent to be stored in the account and could cause violations of the account structure in future development.

```
0 implementations
pub struct UpdateConstituentSpace<'info> {
    #[account(mut)]
    pub owner: Signer<'info>,
    #[account(mut,
        // realloc is based pocket size = 4 + MAX_POCKET_NAME_LENGTH + 4 + 4 + 32 + 1 + 32 + 32 + 8 + 32
        realloc= (149 + MAX_POCKET_NAME_LENGTH) + constituent_num as usize * (size_of::<Constituent>() + size_of::<FeeAccount>()),
        realloc::payer = owner,
        realloc::zero = false
    )]
    pub pocket: Account<'info, Pocket>,

    #[account(
        token::mint = pocket.nft_receipt,
        token::authority = owner,
        constraint = nft_receipt_ata.amount == 1
    )]
    pub nft_receipt_ata: Account<'info, TokenAccount>,

    pub system_program: Program<'info, System>,
}
```

Recommended Fix

The correct reallocation size is as 151 bytes of static account data and the dynamic vector elements

```
pub struct UpdateConstituentSpace<'info> {
    #[account(mut)]
    pub owner: Signer<'info>,
    #[account(mut,
        // realloc is based pocket size = 4 + MAX_POCKET_NAME_LENGTH + 4 + 4 + 32 + 1 + 32 + 32 + 8 + 32
        realloc= (151) + constituent_num as usize * (size_of::<Constituent>() + size_of::<FeeAccount>()),
        realloc::payer = owner,
        realloc::zero = false
    )]
```

The patch commit: [7629750]

Conclusion

Mad Shield's audit of the Pocket DeFi programs highlighted the importance of ongoing security, penetration testing, and transparent documentation in the expanding Solana DeFi landscape. All identified vulnerabilities have been addressed, confirming the programs' reliability and security. We commend the Pocket Defi team for their cooperation, contributing to the overall effectiveness of this audit.