Plasma Program Audit Report

– Mad Shield

contact@madshield.xyz

Sep 20, 2024

Table of Contents

1. Introduction

This audit focuses on the Plasma Smart Contract, a Solana-based program designed to mitigate Maximum Extractable Value (MEV), particularly front-running attacks. Plasma addresses MEV by implementing a snapshot mechanism that records the base and quote reserves at specific intervals, thereby preventing manipulators from profiting through liquidity changes during high-frequency trading.

The contract utilizes a snapshot window of 4 slots, during which the Solana blockchain's leader remains constant. By enforcing this mechanism, Plasma ensures that any liquidity changes in reserve balances are not reflected within this window, minimizing the risk of manipulation. This is further reinforced through virtual limit orders, which prevent external liquidity adjustments from being accounted for within the snapshot window, adding another layer of protection against front-running.

The audit was conducted between **July 29th, 2024**, and **September 18th, 2024.** During this period, the Plasma program was thoroughly analyzed, and **no critical vulnerabilities** were discovered. However, **2** general recommendations were made and communicated to the development team for future improvements.

This report outlines the audit process, describes the methodology used, and certifies the program as secure to the best of our knowledge.

1.1 Overview

The **Plasma Smart Contract** implements a robust mechanism to prevent atomic front-running attacks using a combination of **snapshot-based reserve tracking** and integration with **MEV-enabled validators** like the Jito-validator on the Solana blockchain. Plasma's approach is specifically designed to neutralize sandwich attacks by anchoring the market price to a snapshot of the pool's base and quote reserves, updated at regular intervals.

At the core of this mechanism is the concept of **4-slot snapshot windows**, during which the **network leader remains unchanged**. Plasma takes a snapshot of the **base and quote reserves** at the start of each window, ensuring that market conditions remain consistent for the duration of these slots. To prevent manipulation through front-running, all orders are handled in relation to these snapshot prices.

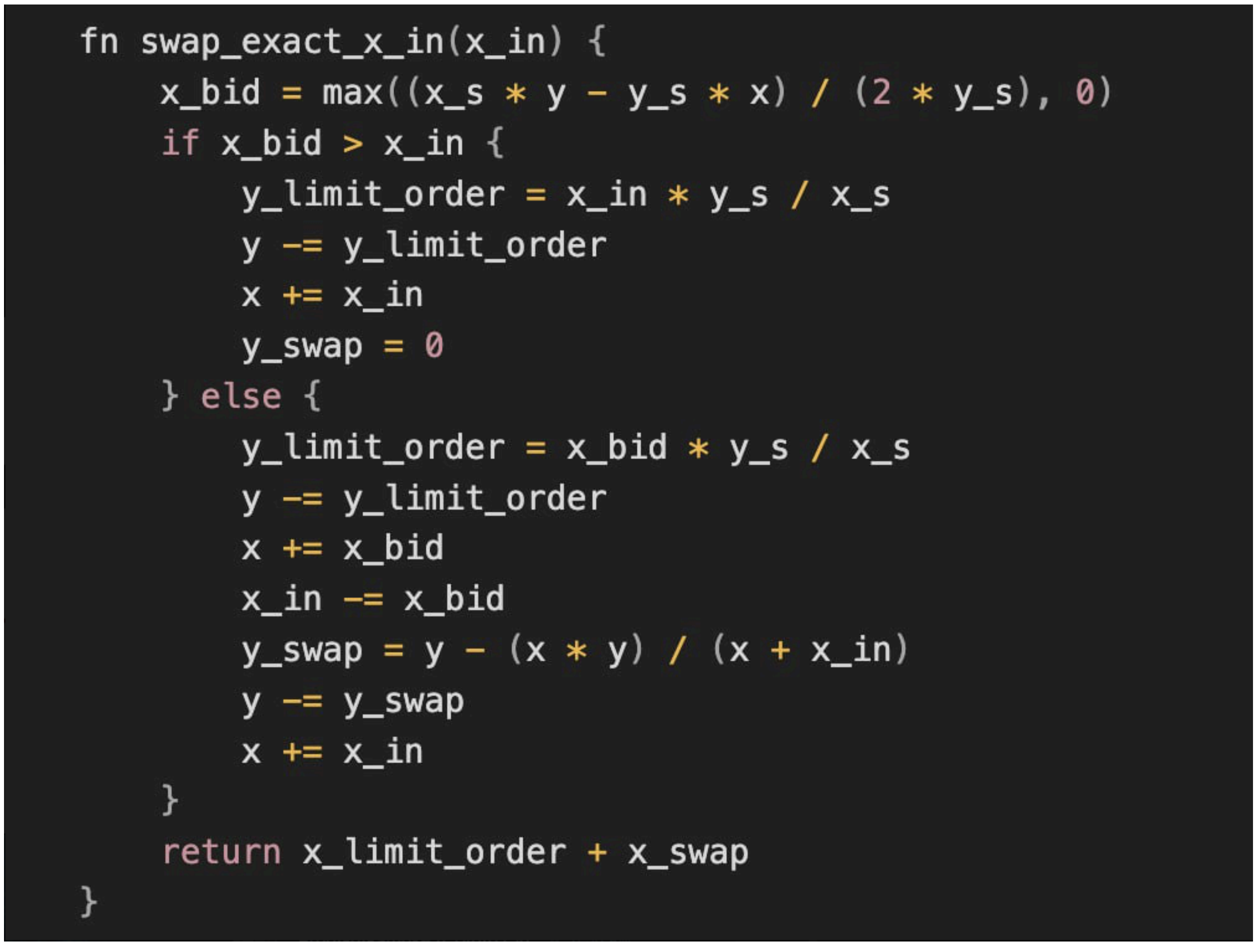
Process for Handling Trades:

- **For Net Buy Orders:** If the trading activity in the current window indicates that traders have net bought up to this point, any incoming sell orders are **first matched against a virtual buy limit order**. The size of this virtual limit order is precisely calculated to **restore the reserve balances to their snapshot values**. Once the virtual limit order is filled, any remaining portion of the sell order is processed normally using the **constant product formula**, reflecting the true market state.
- **For Net Sell Orders:** If traders have net sold up to this point, any incoming buy orders are similarly matched against a **virtual sell limit order** to reset the pool to its snapshot state. The remainder of the order is then processed according to standard market behavior, using the updated pool reserves.

In both cases, the Plasma smart contract ensures that **traders always buy at or above the snapshot price** and **sell at or below it**. This constraint prevents attackers from executing profitable atomic sandwich trades, where they might otherwise front-run and manipulate prices within a single block. By locking trades to the **snapshot price**, Plasma effectively **neutralizes atomic front-running attacks**, ensuring a fairer trading environment for all participants.

Once the virtual limit orders are filled, any excess order size is processed under the **constant product formula** typically used in automated market makers (AMMs). This means the price accurately reflects market conditions at the time the **leader first included a swap transaction** within the 4-slot window, ensuring integrity in price discovery while safeguarding against MEV exploits.

This is shown in the pseudo-algorithm below for `buy_exact_in`. Same goes for the other 3 swap types.



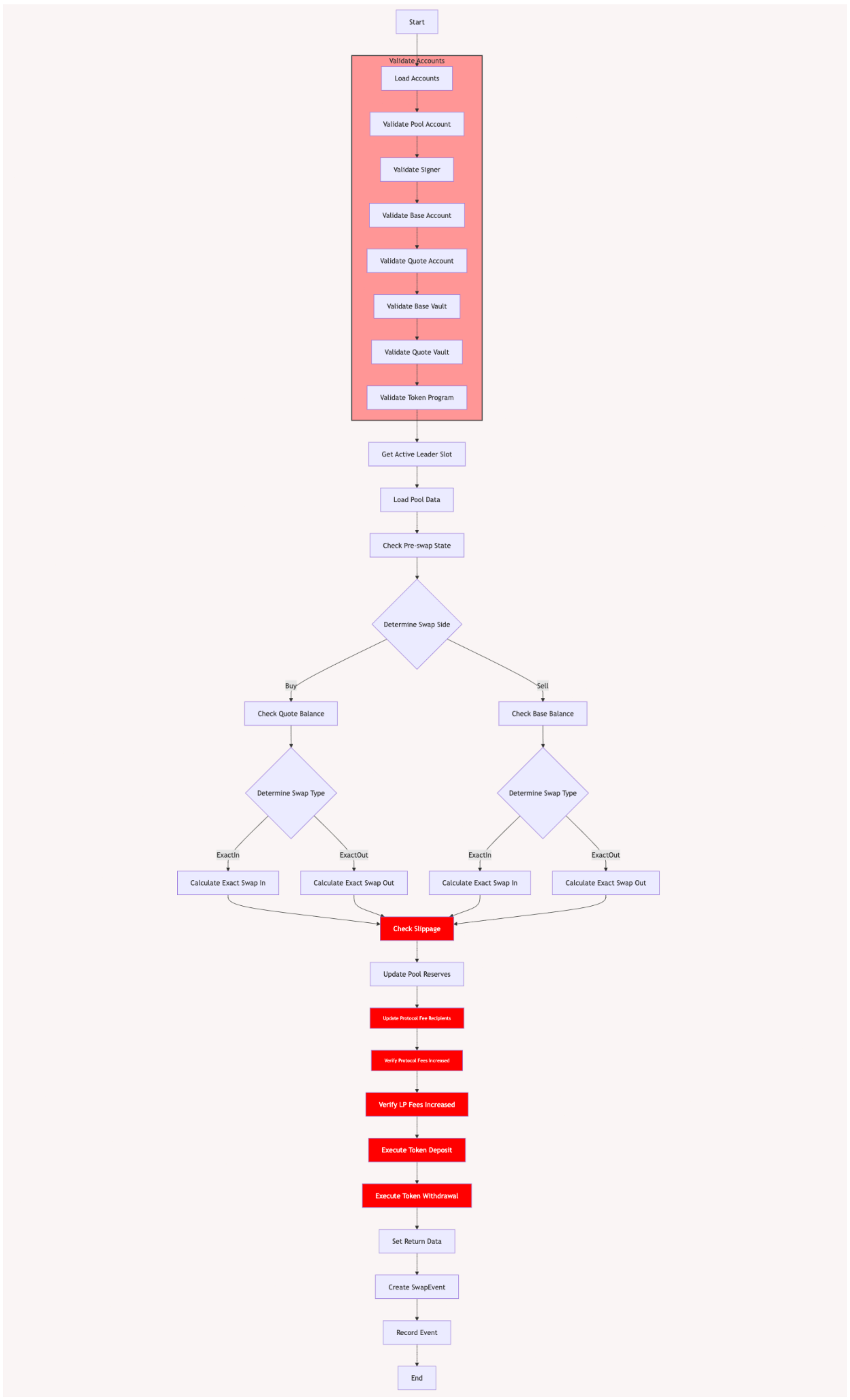
```
fn swap_exact_x_in(x_in) {
    x_bid = max((x_s * y - y_s * x) / (2 * y_s), 0)
    if x_bid > x_in {
        y_limit_order = x_in * y_s / x_s
        y -= y_limit_order
        x += x_in
        y_swap = 0
    } else {
        y_limit_order = x_bid * y_s / x_s
        y -= y_limit_order
        x += x_bid
        x_in -= x_bid
        y_swap = y - (x * y) / (x + x_in)
        y -= y_swap
        x += x_in
    }
    return x_limit_order + x_swap
}
```

The following outlines key steps in each instruction to understand the internal mechanism and identify security-critical actions.

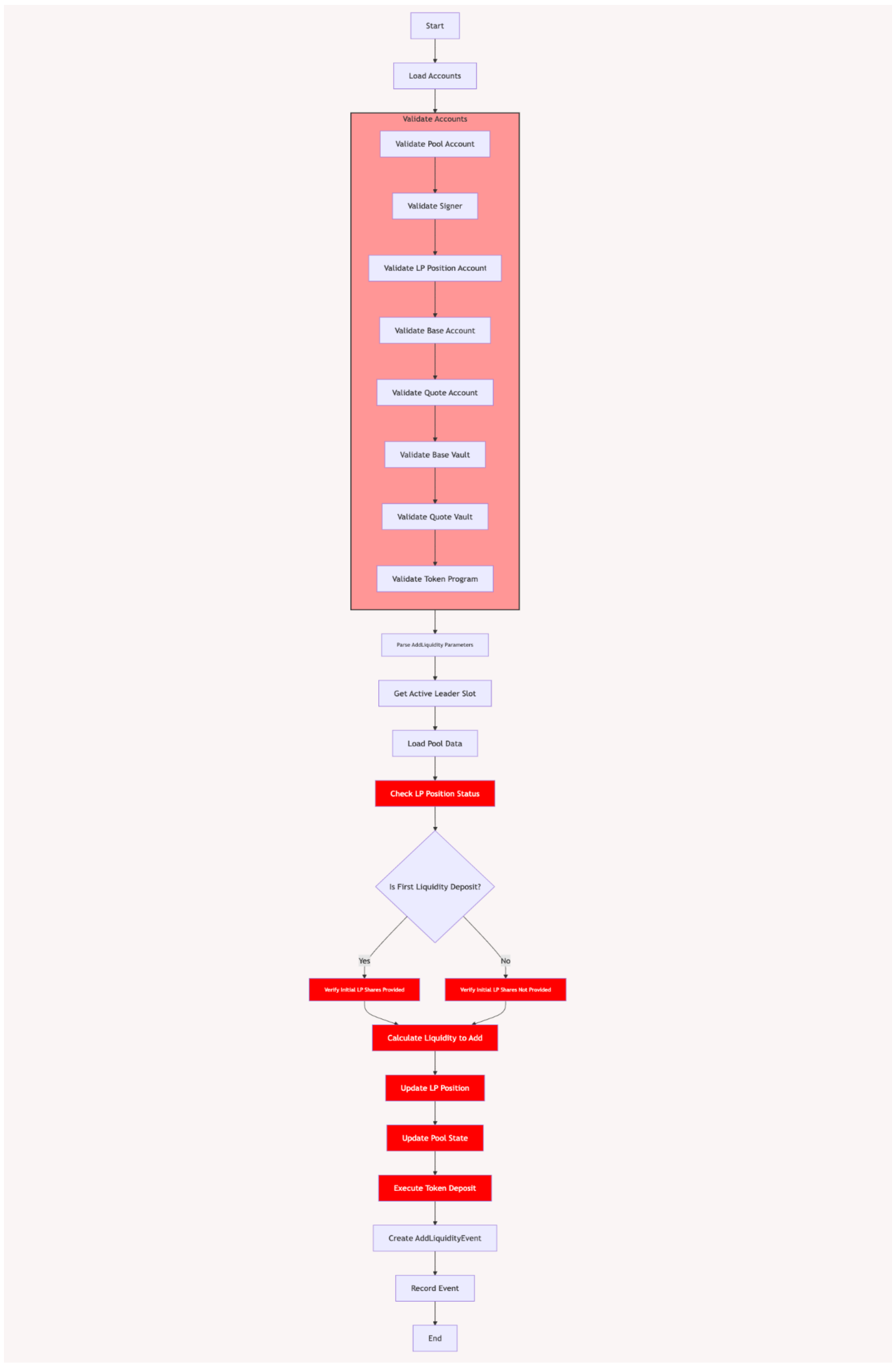
These actions are highlighted in red boxes where more scrutiny during the audit is applied and is brought about to the attention of the team to keep a closer eye on during future development and updates.

1.2 Instructions

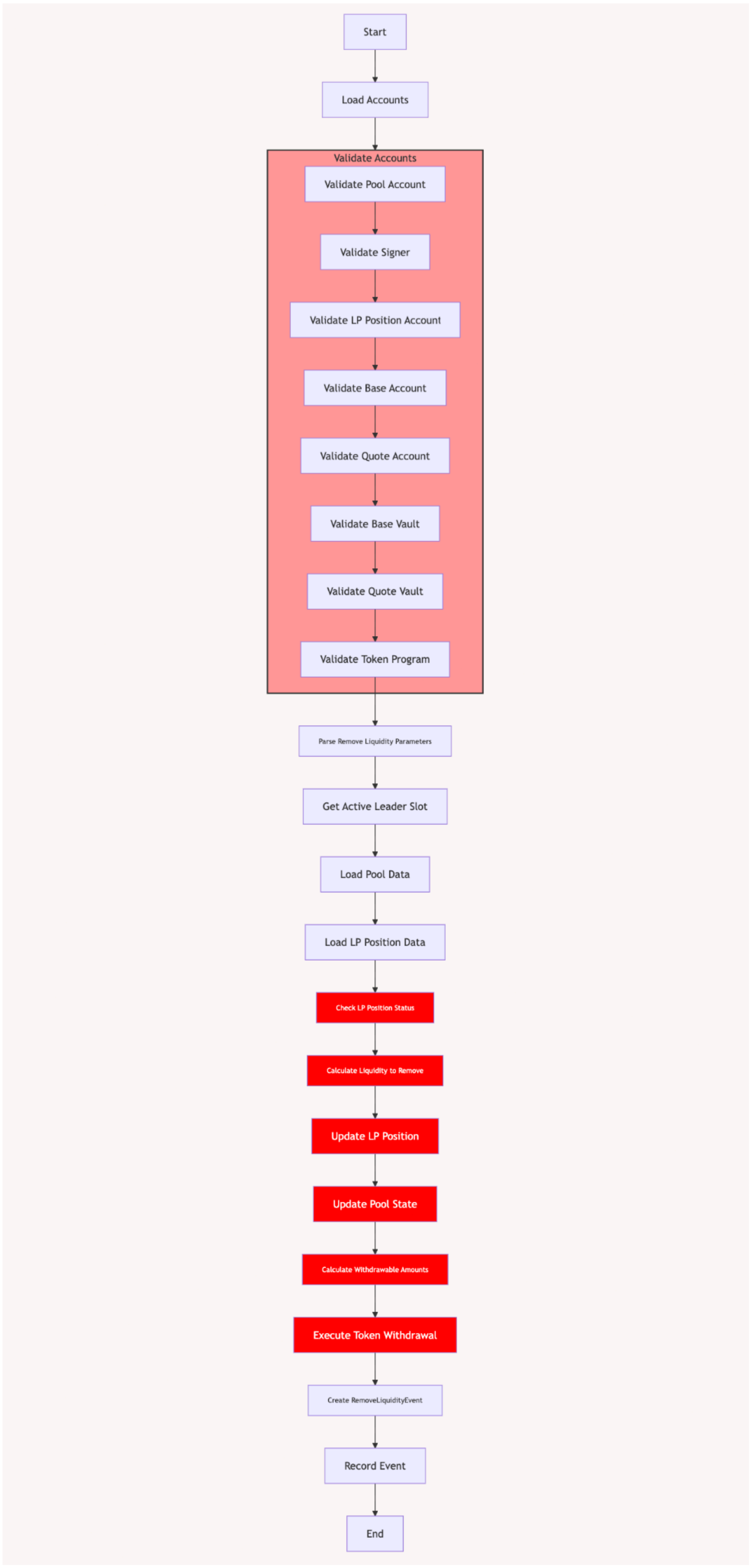
1. Swap



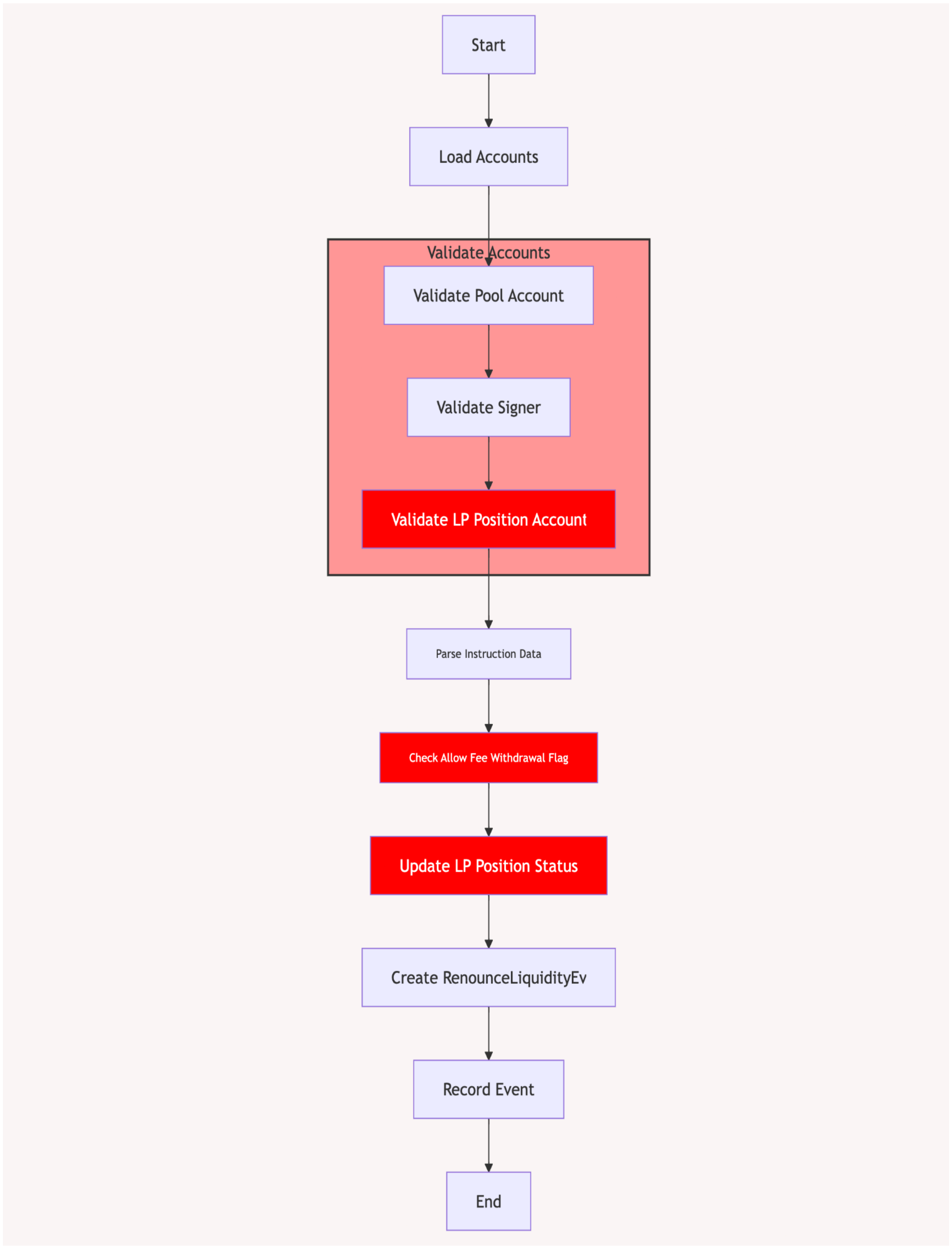
2. Add_Liquidity



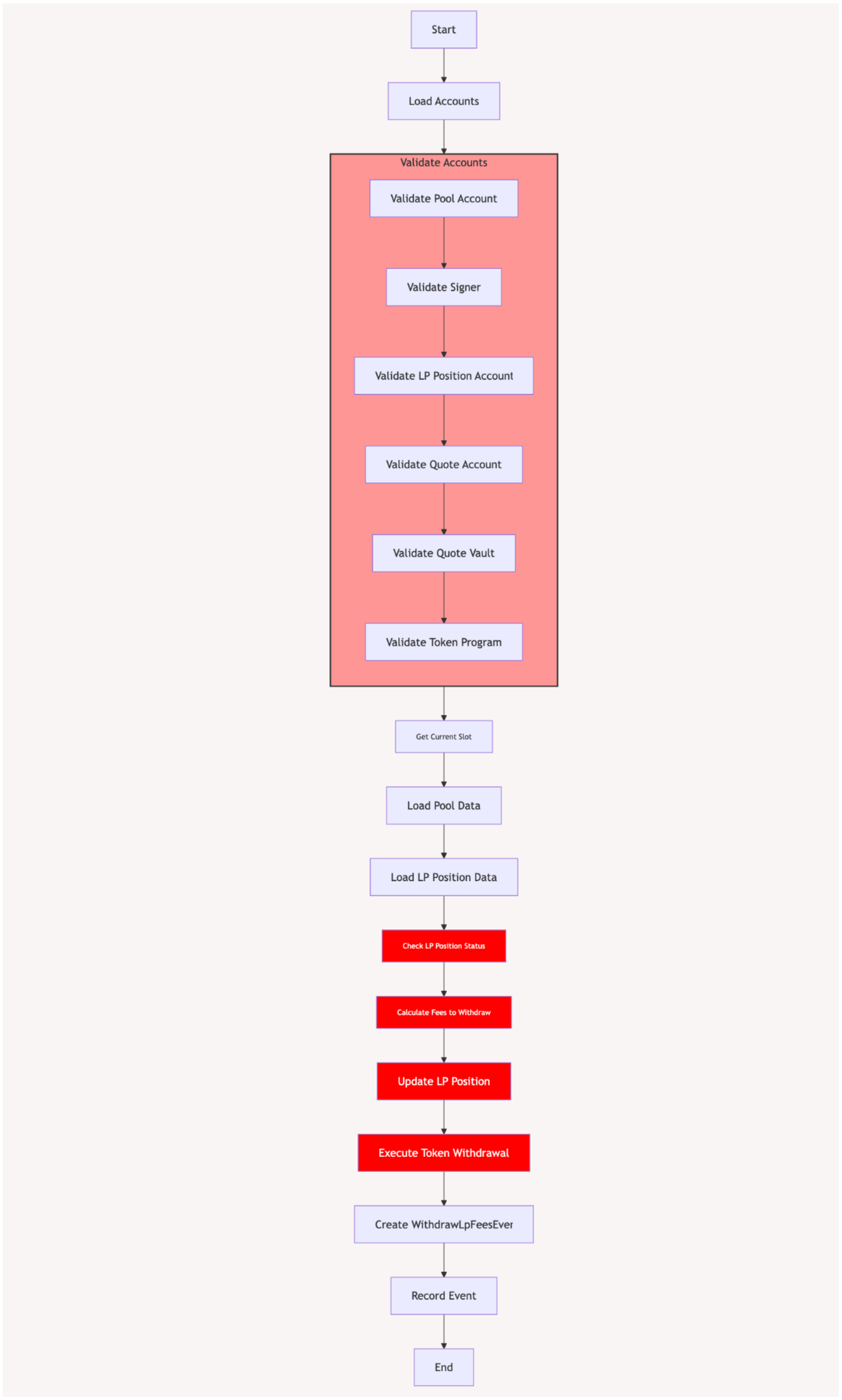
3. Remove_Liquidity



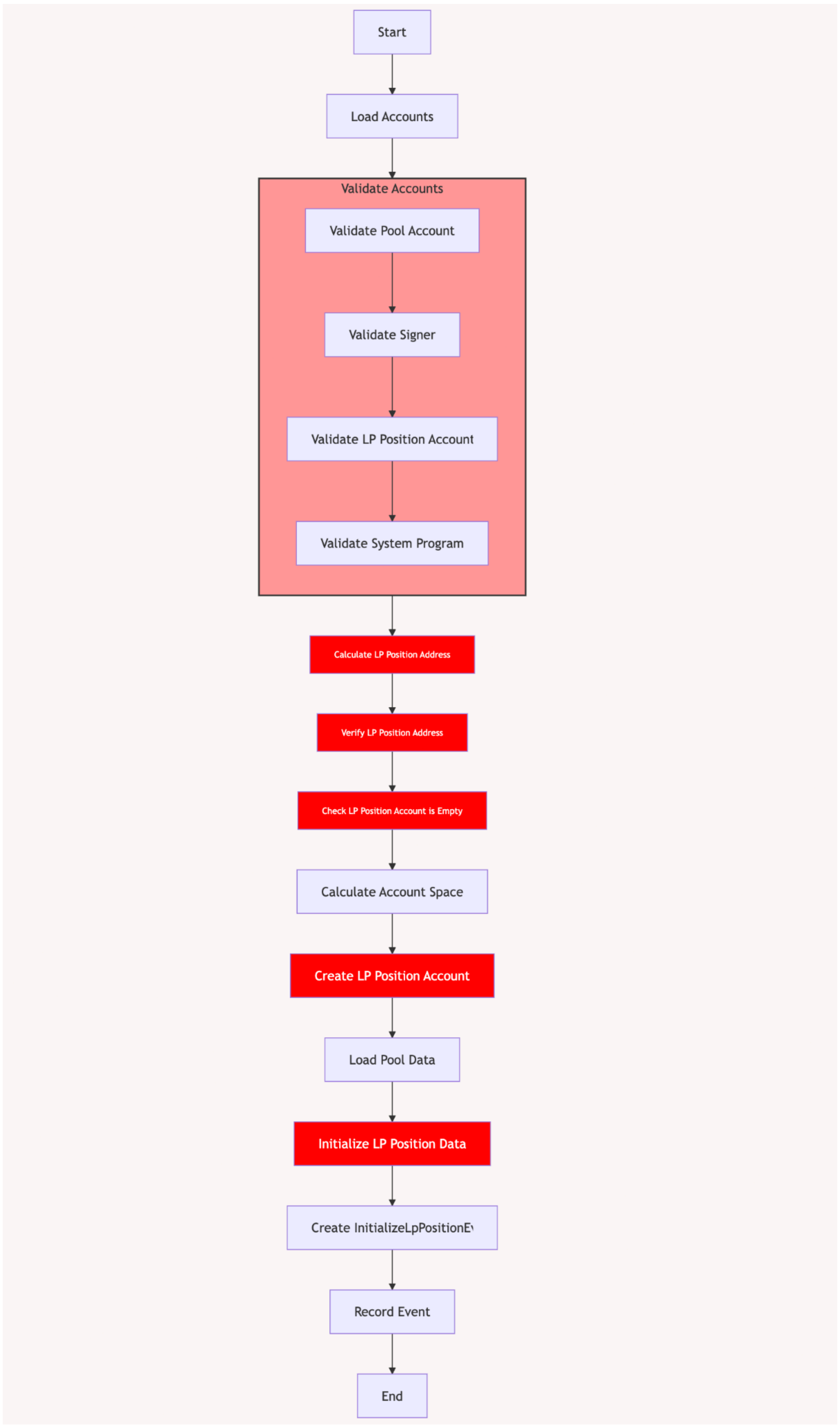
4. Renounce_Liquidity


Start
Load Accounts
Validate Accounts
Validate Pool Account
Validate Signer
Validate LP Position Account
Parse Instruction Data
Check Allow Fee Withdrawal Flag
Update LP Position Status
Create RenounceLiquidityEv
Record Event
End

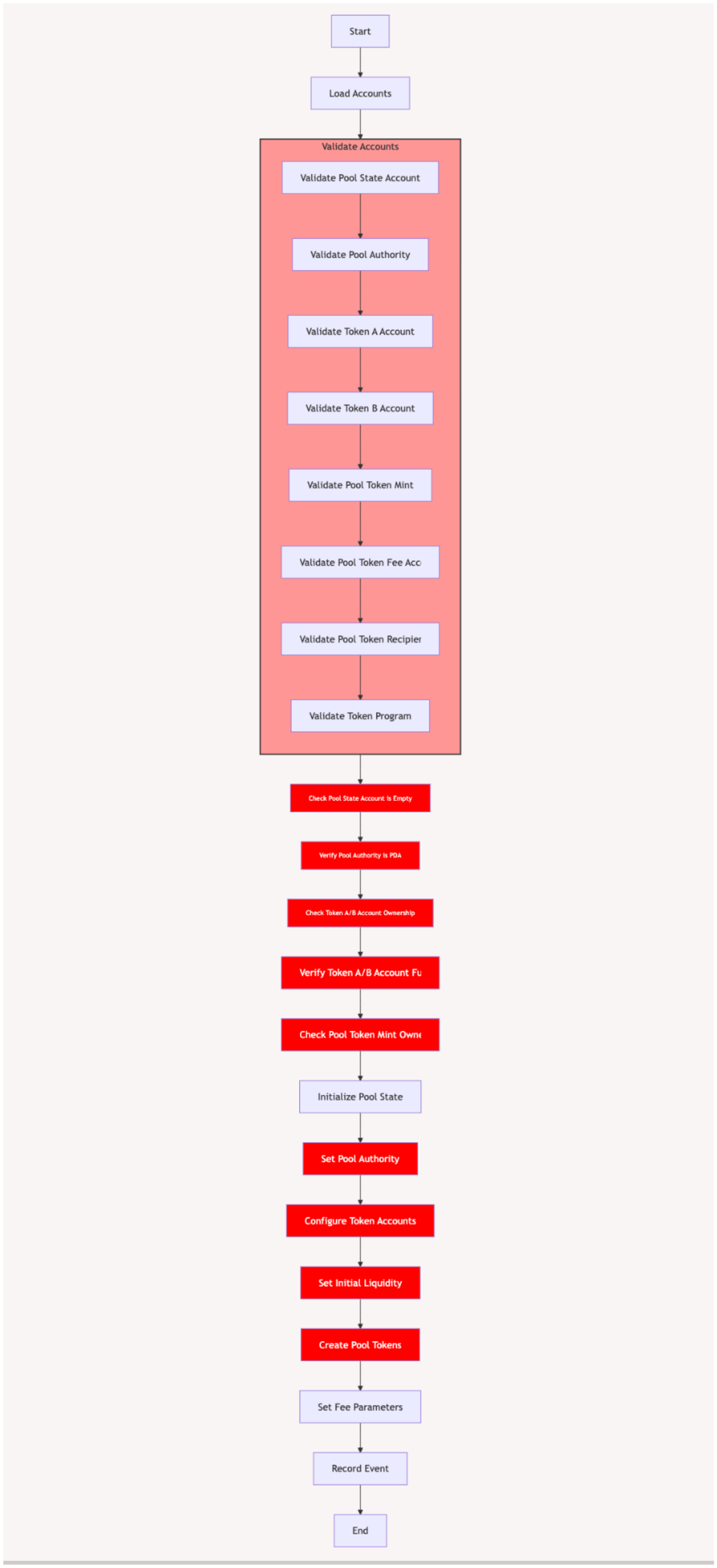
5. Withdraw_Lp_Fees



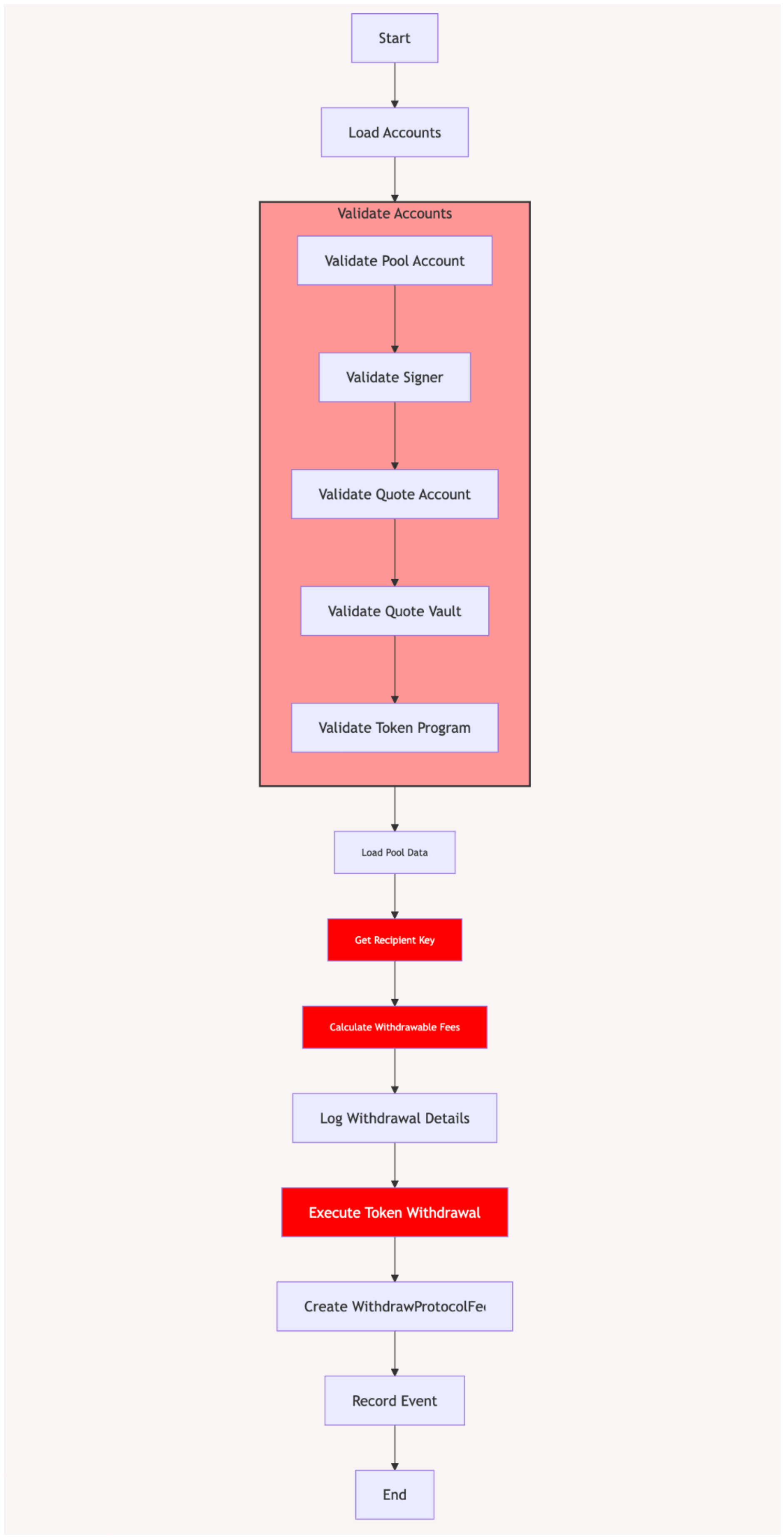
6. Initialize_Lp_Position


Start
Load Accounts
Validate Accounts
Validate Pool Account
Validate Signer
Validate LP Position Account
Validate System Program
Calculate LP Position Address
Verify LP Position Address
Check LP Position Account is Empty
Calculate Account Space
Create LP Position Account
Load Pool Data
Initialize LP Position Data
Create InitializeLpPositionE
Record Event
End

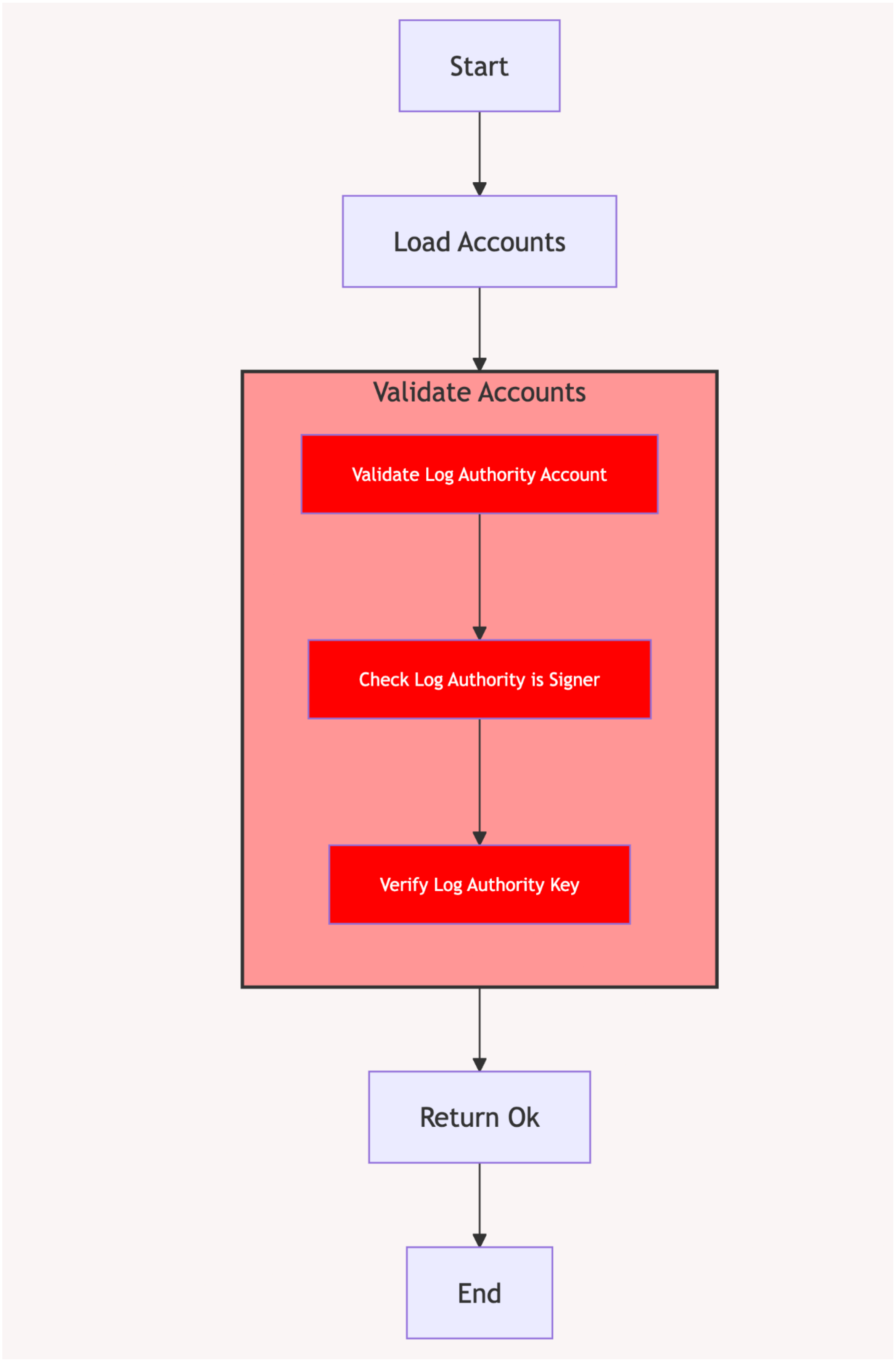
7. Initialize_Pool



8. Withdraw_Protocol_Fees



9. **Log**


Start
Load Accounts
Validate Accounts
Validate Log Authority Account
Check Log Authority is Signer
Verify Log Authority Key
Return Ok
End

2. Scope and Objectives

The primary objectives of the audit are defined as:

- Minimizing the possible presence of any 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 Token Metadata and the goals of the Raydium team.

- 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 the vulnerabilities with an eventual publication of a comprehensive audit report to the public audience for all stakeholders to understand the security status of the programs.

Ellipsis Labs has delivered the plasma program to Mad Shield at the following Github repositories.

Repository URL	https://github.com/Ellipsis-Labs/plasma-private
Commit (start of audit)	940c0f937b02514bed7ed7759953b1c95421253a
Commit (end of audit)	164cded20fa295c601788670577569bd3b34eb2c

Final Camera-Ready Code: https://github.com/Ellipsis-Labs/plasma

3. Methodology

After the initial request from the Ellipsis Labs to review we did a quick read-through of the code to evaluate the scope of the work and recognize early potential footguns.

The audit of the Plasma Smart Contract focused on testing the program's efficacy in mitigating Maximum Extractable Value (MEV), particularly front-running attacks. To ensure a comprehensive analysis, we employed both practical testing and code-based review approaches:

1. **Testing Approach:**
    - We gathered real-world MEV transaction data from a real-time Solana sandwich monitor [1], which collects extensive data on sandwich attacks. This database, containing over 500,000 front-run transactions, was used to evaluate how effectively Plasma mitigated such activities.
    - Our team implemented a crawler to extract data from Raydium, Meteora, Lifinity, and other decentralized exchanges (DEXs) on the Solana blockchain to ensure diverse testing scenarios.
    - The program's limit order mechanism was tested with these real-world samples to determine its ability to prevent the profitability of front-running transactions, a key feature aimed at reducing MEV.
2. **Plasma's Testing Suite:**

    The Plasma contract includes a versatile testing suite designed to simulate various initial conditions for the pools, such as reserve balances and market activity. This testing suite enabled detailed simulations of both **buy and sell scenarios** using the four core swap functions: `buy_exact_in`, `buy_exact_out`, `sell_exact_in`, and `sell_exact_out`. These functions were executed with predefined input/output amounts to verify the correctness and security of the contract's mechanism.

    - **Vanilla Testing:** We ran **every front-run transaction** through the system to evaluate its resistance against single-shot MEV attacks. By running over 500,000 front-run transactions gathered from real-world DEX activity, the system's ability to mitigate simple MEV was validated.
    - **Advanced Testing:** We conducted further tests by simulating **more complex scenarios**. These involved executing multiple front-run transactions for each individual asset **sequentially within the same Solana slot**. The purpose was to test the **srAMM invariance** under heavy load and verify its resilience to high-volume multi-shot front-running.

3. **Findings:**

   While Plasma successfully mitigated front-running attacks in most test cases, where the backrun transaction had a lower price than the frontrun transaction for the targeted asset, we encountered a few edge cases that exposed certain issues:

   1. **Legacy Initialization Constraint:** During the single-shot penetration testing, it was unveiled that a strict condition inherited from legacy AMM implementations, prevents pool initialization with otherwise valid and unproblematic reserve balances. This is explained in **Plasma_GR_01**
   2. **Overflow Issue in Fee Calculation:** We identified an overflow issue in the fee calculation mechanism. Specifically, when calculating the **quote amount before fees**, the approximation formula used caused an overflow. The comment in the code section provides insight into the issue:

In conclusion, while Plasma demonstrates strong resilience against front-running and MEV attacks, addressing the legacy initialization constraint and overflow issue in the fee calculation would further enhance the system's robustness.

This methodology ensured that the Plasma contract was tested under real-world conditions, leading to actionable insights and improvement suggestions.

4. Findings & Recommendations

Our severity classification system adheres to the criteria outlined here.

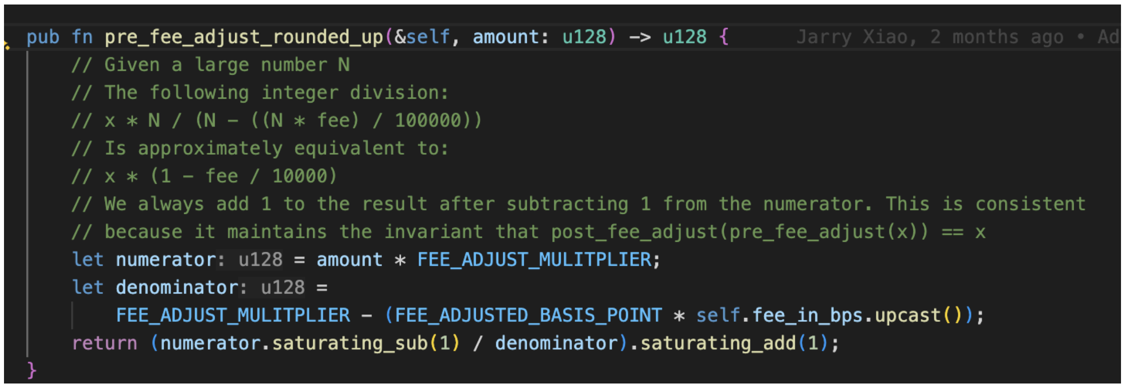
Severity Level	Exploitability	Potential Impact	Examples
Critical	Low to moderate difficulty, 3rd-party attacker	Irreparable financial harm	Direct theft of funds, permanent freezing of tokens/NFTs
High	High difficulty, external attacker or specific user interactions	Recoverable financial harm	Temporary 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
PLASMA_01 [RESOLVED]	Overflow Issue in Fee Calculation	*Medium*
PLASMA_GR_01 [ACKNOWLEDGED]	On the extra dependence on the validate_asset_permissions function	*Informational*

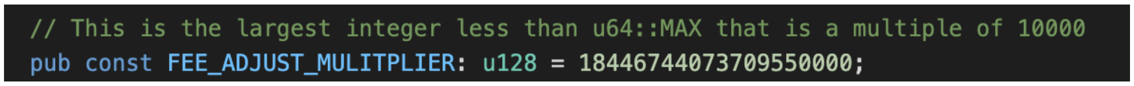
1. PLASMA_01 - Overflow Issue in Fee Calculation [Medium]

An overflow issue was identified in the **fee calculation mechanism**, particularly when handling large values for the **quote amount** before fees. The overflow occurs due to the following approximation formula used in the fee calculation:



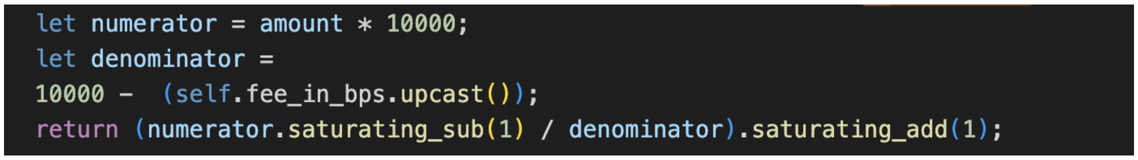
```rust
pub fn pre_fee_adjust_rounded_up(&self, amount: u128) -> u128 {      Jarry Xiao, 2 months ago • Ad
    // Given a large number N
    // The following integer division:
    // x * N / (N - ((N * fee) / 100000))
    // Is approximately equivalent to:
    // x * (1 - fee / 10000)
    // We always add 1 to the result after subtracting 1 from the numerator. This is consistent
    // because it maintains the invariant that post_fee_adjust(pre_fee_adjust(x)) == x
    let numerator: u128 = amount * FEE_ADJUST_MULITPLIER;
    let denominator: u128 =
        FEE_ADJUST_MULITPLIER - (FEE_ADJUSTED_BASIS_POINT * self.fee_in_bps.upcast());
    return (numerator.saturating_sub(1) / denominator).saturating_add(1);
}
```

The issue is triggered when the approximation fails due to an integer overflow during the calculation as the FEE_ADJUST_MULTIPLIER is this huge number.



```rust
// This is the largest integer less than u64::MAX that is a multiple of 10000
pub const FEE_ADJUST_MULITPLIER: u128 = 18446744073709550000;
```

Which limits the AMM's maximum trading range. To address this issue, we recommended to adjust FEE_ADJUST_MULTIPLIER to 1000 and update the function to use the following which has the same result and purpose.



```rust
let numerator = amount * 10000;
let denominator =
10000 -  (self.fee_in_bps.upcast());
return (numerator.saturating_sub(1) / denominator).saturating_add(1);
```

5. General Recommendations

As part of the audit process, we identified a non-critical issue that is summarized below. While this finding does not represent vulnerabilities, addressing them would improve the program's robustness and efficiency. Each issue is assigned a general recommendation (GR) code and is detailed accordingly.

---

1. PLASMA_GR_01- Restrictive Market Initialization Constraints [Informational]

Plasma inherited a legacy condition from the Uniswap AMM implementation, which requires a predefined amount of 1000 LP tokens to be present in a pool to ensure market trades are always possible and to prevent the market creator from "rugging" the pool. Plasma replicates this condition to safeguard market functionality, but it has led to limitations, particularly in the creation of exotic or niche markets.

We recommend relaxing this constraint by requiring only a single unit of LP tokens to be withheld. This adjustment would expand the possible range of trading prices and volumes, empowering the creation of more diverse markets while still maintaining necessary protections against market manipulation. This approach aligns with similar recommendations we made to the Raydium team for their AMM audit. [2]

6. Conclusion

The audit of the Ellipsis Labs' Plasma was conducted by Mad Shield using an in-depth, hands-on approach that emphasizes thorough analysis of the program. Our team strives to stay "boots on the ground," working closely with each unique program to uncover potential vulnerabilities and security gaps.

Mad Shield's commitment to innovation in auditing methodologies is evident throughout our process. We continuously experiment with new techniques to ensure the highest level of security for our clients. This allows us to dive deeper into the code, testing real-world scenarios, and assessing potential weaknesses that might be overlooked by more conventional auditing processes.

No critical vulnerabilities were discovered during this audit, and all findings were promptly communicated to the development team. The suggestions provided are designed to strengthen the codebase and ensure long-term stability and security. Mad Shield remains dedicated to pushing the boundaries of smart contract auditing.

7. References

[1] sandwiched.me – https://sandwiched.me/
[2] Mad Shield Raydium Audit Report –
https://github.com/madshieldio/Publications/blob/main/Raydium/raydium-amm-v-1.0.0.pdf