Deriverse DEX Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

RajKumar

JesJupyter

Ctrus

Assisting Auditors

Alexzoid (Formal Verification)

December 15, 2025

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Deriverse is a decentralized exchange (DEX) built on Solana that combines automated market maker (AMM) liquidity pools with central limit order book (CLOB) functionality for both spot and perpetual futures markets.

Spot Trading uses a hybrid architecture where the constant-product AMM (k = asset_tokens × crncy_tokens) provides baseline liquidity while limit orders on an RB-tree based order book offer price improvement. Traders can execute market orders, place limit orders, or swap tokens directly against the AMM & CLOB. Liquidity providers earn a share of trading fees distributed proportionally from protocol-collected fees.

Perpetual Futures operates on an isolated margin model with configurable leverage (dynamically capped based on daily volatility). The system includes funding rate mechanisms that periodically rebalance long/short positions, automated margin calls with penalty rates feeding an insurance fund, and a rebalancing mechanism for position management. Traders must purchase a "market seat" at a dynamic price based on current participant count before trading perpetuals; this seat can be sold back when exiting. Perpetual markets are created by upgrading existing spot instruments when they meet readiness criteria.

Tokenomics centers on the DRVS native token, which enables participation in on-chain governance voting (fee rates, discount parameters, pool ratios), earning of dividend distributions from protocol fees. Additionally, users can receive airdrop according to their spot/perp trading and LP activities.

Additional Features include a referral program providing fee discounts to referred traders with rewards to referrers, a points program that accumulates trading/LP activity rewards convertible to DRVS via airdrops, and a private mode for controlled launches with whitelisted participants. All protocol state is stored on-chain through Solana PDAs with no custodial control over user funds.

5 Audit Scope

The audit scope was limited to:

```
src/program/perp/leverage_validation.rs
src/program/perp/mod.rs
```

```
src/program/perp/perp_context.rs
src/program/perp/perp_engine.rs
src/program/processor/airdrop.rs
src/program/processor/change_ref_program.rs
src/program/processor/close_account.rs
src/program/processor/deposit.rs
src/program/processor/dividends_allocation.rs
src/program/processor/dividends_claim.rs
src/program/processor/fees_deposit.rs
src/program/processor/fees_withdraw.rs
src/program/processor/mod.rs
src/program/processor/move_spot_avail_funds.rs
src/program/processor/new_base_crncy.rs
src/program/processor/new_holder_account.rs
src/program/processor/new_instrument.rs
src/program/processor/new_operator.rs
src/program/processor/new_perp_order.rs
src/program/processor/new_ref_link.rs
src/program/processor/new_root_account.rs
src/program/processor/new_spot_order.rs
src/program/processor/new_token.rs
src/program/processor/next_voting.rs
src/program/processor/perp_change_leverage.rs
src/program/processor/perp_deposit.rs
src/program/processor/perp_forced_close.rs
src/program/processor/perp_mass_cancel.rs
src/program/processor/perp_order_cancel.rs
src/program/processor/perp_quotes_replace.rs
src/program/processor/perp_statistics_reset.rs
src/program/processor/perp_withdraw.rs
src/program/processor/reset_tree.rs
src/program/processor/set_instr_oracle_feed.rs
src/program/processor/set_instr_ready_for_perp_upgrade.rs
src/program/processor/spot_lp.rs
src/program/processor/spot_mass_cancel.rs
src/program/processor/spot_order_cancel.rs
src/program/processor/spot_quotes_replace.rs
src/program/processor/swap.rs
src/program/processor/upgrade_to_perp.rs
src/program/processor/voting.rs
src/program/processor/withdraw_spl_tokens.rs
src/program/processor/withdraw.rs
src/program/spot/engine.rs
src/program/spot/mod.rs
src/program/spot/orders.rs
src/program/spot/spot_context.rs
src/program/constants.rs
src/program/create_client_account.rs
src/program/helper.rs
src/program/instruction_data.rs
src/program/log.rs
src/program/mod.rs
src/program/settings.rs
src/state/perps/perp_infos.rs
src/state/perps/perp_trade_header.rs
src/state/spots/spot_account_header.rs
src/state/spots/spot_accounts.rs
src/state/spots/spot_infos.rs
src/state/candles.rs
src/state/client_community.rs
src/state/client_drv.rs
src/state/client_primary_account_header.rs
src/state/client_primary.rs
```

```
src/state/community_account_header.rs
src/state/community.rs
src/state/holder.rs
src/state/instrument.rs
src/state/mod.rs
src/state/pdf.rs
src/state/perps.rs
src/state/root.rs
src/state/spots.rs
src/state/token.rs
src/state/types.rs
src/utils/access_manager.rs
src/utils/from_account_info.rs
src/utils/mod.rs
src/utils/new_types.rs
src/lib.rs
src/state/perps/perp_accounts.rs
```

6 Executive Summary

Over the course of 25 days, the Cyfrin team conducted an audit on the Deriverse DEX smart contracts provided by Deriverse. In this period, a total of 99 issues were found.

The findings consist of 20 High, 37 Medium and 18 Low severity issues with the remainder being informational. The audit uncovered significant vulnerabilities across multiple protocol components including:

Access Control Failures: Critical authorization checks were missing in several functions. The `sell_market_seat` function allowed any user to force-sell another user's market seat by passing `check_signer = false`, enabling unauthorized closure of perpetual positions. Similarly, `change_points_program_expiration` and `voting_reset` instructions lacked signature verification, allowing permissionless execution of admin-gated functionality.

Governance Manipulation: The voting system contained multiple exploitable flaws. Users could cast votes with tokens they no longer held by exploiting timing of deposits and withdrawals during voting periods. Additionally, votes could be submitted after voting periods officially ended, and no quorum requirement existed to prevent a small minority of token holders from controlling protocol governance decisions.

Perpetual Engine Vulnerabilities: Several perpetual trading functions used stale price data for critical calculations. Functions like `perp_change_leverage` and `sell_market_seat` failed to update the underlying price before performing margin call checks, funding rate settlements, and leverage validations. The margin call detection logic also ignored the liquidation threshold parameter, creating a discrepancy between when the system detects margin calls and when liquidations actually occur. A duplicate function call bug (`is_long_margin_call` called twice instead of checking both long and short) allowed users with underwater short positions to withdraw funds that should have been locked.

Fee and Incentive Calculation Errors: Makers received no rebates when orders were filled via swap operations due to incorrect fee rate handling. The fee discount ratio was incorrectly reset to 1.0 whenever a new currency was added to the protocol, eliminating users' previously earned discounts. Dividend calculations could use stale rates if the allocation function wasn't called before user operations. Also, the fee calculation in the `spot_lp` instruction used an incorrect decimal factor, which led to fee overpayment.

Denial of Service Vectors: Protocol initialization could be permanently blocked by attackers pre-funding PDA addresses with lamports, causing `create_account` calls to fail. Additionally, hardcoded token account sizes (165 bytes) caused initialization failures for Token-2022 mints with extensions.

Post Audit Recommendations

Due to the significant number of High severity findings it is statistically likely that more serious vulnerabilities still remain which were not discovered during the audit engagement; the auditors kept producing new findings right up until the last day of the audit. Hence it is recommended that prior to deploying significant capital on-chain in a

production environment, another audit be conducted during which no Critical or High severity findings should be found.

Summary

Project Name	Deriverse DEX
Repository	protocol-v1
Commit	1edfaa3c670e...
Fix Commit	871e06d12eaa...
Audit Timeline	Oct 29th - Dec 2nd, 2025
Methods	Manual Review, Formal Verification

Issues Found

Critical Risk	0
High Risk	20
Medium Risk	37
Low Risk	18
Informational	24
Gas Optimizations	0
Total Issues	99

Summary of Findings

[H-01] Users can cast vote without actually holding tokens	Resolved
[H-02] Missing Version Validation for Private Clients Account in `new_private_client`	Resolved
[H-03] Airdrop Implementation Issues: Unvalidated Ratio and Potential Transfer Direction Mismatch	Resolved
[H-04] `update_records` always resets `fees_ratio` to 1 whenever a new currency is added	Resolved
[H-05] Stale Funds Data in Leverage Validation Due to Missing Funding Rate Settlement	Resolved
[H-06] Fee Discount Calculation Ignores Base Currency Value Differences	Resolved
[H-07] Missing ps check causes Dividend Loss	Resolved
[H-08] Price manipulation during initial ps minting can cause user losses	Resolved
[H-09] Incorrect Fee Scaling in `spot_lp`	Resolved
[H-10] DoS attack exhausting account space and halting spot trading	Resolved

[H-11] Attacker can cause losses to others by partially closing their position at an unfavorable price	Resolved
[H-12] Makers' rebates are not paid in case of swap	Resolved
[H-13] `perp-change-leverage` uses stale `perp-underlying-px`	Resolved
[H-14] Shorts can withdraw full available funds instead of being restricted to margin call limits in `perp-withdraw`	Resolved
[H-15] Margin call detection functions ignore liquidation threshold	Resolved
[H-16] Multiple inconsistencies in `sell-market-seat`	Resolved
[H-17] Improper header handling in `SpotFeesReport` logging causes DoS on swap instruction	Resolved
[H-18] Missing Signer Verification in `voting_reset` Function Allows Unauthorized Execution	Resolved
[H-19] Using the underlying price instead of the perp price can cause problems	Resolved
[H-20] Spot price manipulation can lead to unfair liquidations	Resolved
[M-01] `create_account` can be dosed with pre-funding	Resolved
[M-02] Fixed token account size causes initialization failures for token accounts whose mints have token 22 extensions are active	Resolved
[M-03] Accounts may be created with incorrect rent-exemption due to `Rent::default` usage	Resolved
[M-04] `change_points_program_expiration` is permissionless	Resolved
[M-05] Anyone can sell anyone else's market seat	Resolved
[M-06] Voting is allowed even after voting period's end time.	Resolved
[M-07] Missing Slippage Protection in Market Seat Buy/Sell Operations	Resolved
[M-08] Missing Quorum Requirement in Governance Voting	Resolved
[M-09] Inconsistent Token Account Address Requirements Between Deposit and Withdraw	Resolved
[M-10] Silent Error Handling in `clean_generic` Introduces Multiple Risks	Resolved
[M-11] `ClientCommunityState::update` function does not update the `rate` before calculating new `dividends_value`	Resolved
[M-12] Lack of slippage in `spot_lp` liquidity operations due to relying on the changing `header.crncy_tokens` and `header.asset_tokens`	Resolved
[M-13] Referral Incentives Disabled for All Legitimate Users During Any Liquidation	Resolved
[M-14] Slippage Guard Could be Too Loose For Leveraged Perp Markets	Resolved
[M-15] Referral discount is not applied when `fees_prepayment` is zero in `PerpEngine::fill`	Resolved
[M-16] Operator precedence Issue during points program expiration validation in `change_points_program_expiration`	Resolved
[M-17] Attacker can extract value by buying and selling the seat	Resolved

[M-18] Fee Prepayment Locked Due to Asset Record Cleanup after `fees_deposit`	Resolved
[M-19] Decimal Mismatch in Fee Prepayment Accounting Causes Incorrect Balance Tracking	Resolved
[M-20] Users can sell their market seat without paying loss coverage	Resolved
[M-21] Users are getting back their `soc-loss-funds` while selling their market seat	Resolved
[M-22] Users can provide old price feeds to trade in their favor	Resolved
[M-23] Users incur losses when selling seats	Resolved
[M-24] `perp_statistics_reset` can be used by users to skip `collactable-losses` while selling market seat	Resolved
[M-25] Missing Array Synchronization in `dividends_claim` Prevents Users from Claiming Dividends for Newly Added Base Currencies	Resolved
[M-26] `min_qty` Bypass via `IOC` limit order	Resolved
[M-27] Wrong accounting of fee in perp engine	Resolved
[M-28] User's chosen leverage is overwritten to `max-leverage` on every perp operation	Resolved
[M-29] Redundant State Updates in `fill` Function Cause Issues	Resolved
[M-30] Missing Trade Count Updates in `reversed_swap` Function for Buy Orders	Resolved
[M-31] Missing `change_funding_rate` Call After Price Update in `perp_mass_cancel` and `perp_order_cancel`	Resolved
[M-32] Inefficient rebalancing can cause the loss of users	Resolved
[M-33] Incorrect Margin Call State Detection After Liquidation in `perp_withdraw`	Resolved
[M-34] Stale edge price in liquidation tree after perp withdraw call	Resolved
[M-35] Missing Update of `perp_spot_price_for_withdrowal` in `perp_withdraw` Function	Resolved
[M-36] Margin call uses stale edge price	Resolved
[M-37] Insurance fund decrease when `margin_call_penalty_rate` is less than `rebates_rate`	Resolved
[L-01] Casting from `u64` to `i64` causes genuine deposit requests to fail in `deposit` function	Resolved
[L-02] Expired Private Client Cannot Be Re-added to Queue	Resolved
[L-03] Account Count Validation Mismatch in `new_base_crncy` Instruction	Resolved
[L-04] Inflexible Voting System Prevents Rapid Parameter Adjustments	Resolved
[L-05] Dividend Calculation Uses Stale Token Balance in Subsequent `update()` Calls After `fees_deposit` with DRVS	Resolved
[L-06] Griefing Attack: Malicious Takers Can Force Order Cancellation by Partial Filling Below Minimum Quantity	Resolved

[L-07] Missing `last_time` Update in `spot_lp` Causes Incorrect Daily Trade Statistics	Resolved
[L-08] Inconsistent Price Reference Used for Trade Execution Logic in `new_spot_order` and `swap`	Resolved
[L-09] Missing Signer and New Account Validation for `asset_token_program_acc` in `new_instrument`	Resolved
[L-10] Referrer cannot be set after account creation	Resolved
[L-11] Inconsistent Price Calculation for Fee in Spot LP Trading	Resolved
[L-12] `get_place_buy_price` function does not currently support the maximum `MAX_SUPPLY` value.	Resolved
[L-13] `get_current_leverage` calculates leverage incorrectly for long positions	Resolved
[L-14] Margin call limit bypass	Resolved
[L-15] Missing Fixing Window Data Accumulation After Daily Reset in `drv_update`	Resolved
[L-16] `get_by_tag` tries to access out of bound index	Resolved
[L-17] Incorrect Amount Logged in `PerpWithdrawReport`	Resolved
[L-18] Silent Failure in `voting_reset`	Resolved
[I-01] Incorrect Price Validation When Creating `NewInstrumentData` Struct during `NewInstrumentInstruction` instruction	Resolved
[I-02] typo error in variables	Resolved
[I-03] Transfers are noop when `lamports_diff` is zero	Resolved
[I-04] Entrypoint panics on empty `instruction_data`	Resolved
[I-05] Unnecessary Lamports Transfer Without Checking Existing Balance	Resolved
[I-06] Incomplete Balance Check Missing Transaction Fee and Reserve Balance	Resolved
[I-07] Eligible intruments may not be propagated	Resolved
[I-08] Insufficient Self-Referral Protection Still Allows Multiple Account Self-Referral	Acknowledged
[I-09] Attacker can exploit Dividend allocation by depositing large mounts of DRVS Tokens	Acknowledged
[I-10] Forced Oldest-Order Eviction Enables Griefing	Resolved
[I-11] User cannot claim dividends after withdrawing their DRVS tokens	Resolved
[I-12] Missing Validation for `order_type` in NewSpotOrderData	Resolved
[I-13] Rounding Error Accumulation in Partial Order Fills Leads to Unfair Cost Distribution	Resolved
[I-14] Returning true when the current time has reached the `expiration_time` in `is_vacant`	Resolved
[I-15] Missing System Program Check in Instructions like `new_operator`(Inconsistency)	Resolved

[I-16] Inefficient `free_index` Selection in Assets Array Causes Performance Degradation	Resolved
[I-17] Withdraw Instruction Trying to Create New Asset Records for Non-Existent Tokens Instead of Failing	Resolved
[I-18] Users Cannot Change Their Vote Once Cast in a Voting Period	Resolved
[I-19] Using a stale `perp_price_delta` to calculate the `perp_funding_rate`	Acknowledged
[I-20] Incorrect Token Program Error Message in Airdrop Flow	Resolved
[I-21] Incorrect Boundary Condition in `check_pool_fees` Function Excludes Valid Minimum Token Transactions	Resolved
[I-22] wrong error emitted from `spot-order-cancel`	Resolved
[I-23] Redundant Flag `READY_TO_DRV_UPGRADE` Appears Unused	Resolved
[I-24] Incorrect mask check in `check_points` function	Resolved

7 Findings

7.1 High Risk

7.1.1 Users can cast vote without actually holding tokens

Description: In `voting.rs` users can vote for protocol decisions with the `drvs` tokens they posses in their respective client accounts. However this mechanism can be abused by users where users can vote with inflated token balances by exploiting the timing of deposits and withdrawals during an active voting period. A malicious user would deposit a few tokens in a period and withdraw those tokens and yet can vote with those tokens, without actually holding them.

Step by step walkthrough: Voting Period 1: Slots 60-100 (voting_counter = 1) Voting Period 2: Slots 100-140 (voting_counter = 2) Voting Period 3: Slots 140-180 (voting_counter = 3)

Step 1: Initial Deposit (Slot 50 - Before Period 1): User deposits 50,000 DRVS tokens.

```
 // In `client_community.rs` `update()` during deposit
if available_tokens != self.header.drvs_tokens {
    // 50,000 != 0 - TRUE

    community_state.header.upgrade()?.drvs_tokens += 50,000;

    if self.header.slot <= community_state.header.voting_start_slot {
        // 0 <= 60 - TRUE
        self.header.current_voting_counter = 0;
        self.header.current_voting_tokens = 50,000;
    }
    self.header.slot = 50;
    self.header.drvs_tokens = 50,000;
}
```

Step 2: User cast vote in Period 1 (Slot 80):

```
// In voting.rs
if data.voting_counter != community_state.header.voting_counter {
    // 1 == 1 - Check passes
}

if client_community_state.header.slot <= community_state.header.voting_start_slot {
    // 50 <= 60 - TRUE
    voting_tokens = client_community_state.header.drvs_tokens; // 50,000
    client_community_state.header.current_voting_counter = 1;
    client_community_state.header.current_voting_tokens = 50,000;
} else {
    voting_tokens = client_community_state.header.current_voting_tokens;
}

if voting_tokens > 0 {
    if client_community_state.header.last_voting_counter >=
    ↪  client_community_state.header.current_voting_counter {
        // 0 >= 1 - FALSE, check passes
    }

    // Vote is cast successfully
    community_account_header.voting_incr += 50,000; // Example: user votes INCREMENT
    client_community_state.header.last_voting_counter = 1;
    client_community_state.header.last_voting_tokens = 50,000;
}
```

Step 3: User deposits additional 50,000 tokens in period 2(Slot 120):

```
// In client_community.rs update() during deposit
```

```
if available_tokens != self.header.drvs_tokens {
    // 100,000 != 50,000 - TRUE

    community_state.header.upgrade()?.drvs_tokens += 50,000;

    if self.header.slot <= community_state.header.voting_start_slot {
        // 50 <= 100 - TRUE
        self.header.current_voting_counter = 2; // Updates to period 2
        self.header.current_voting_tokens = 100,000; //
    }
    self.header.slot = 120; // Updated to current slot after the check
    self.header.drvs_tokens = 100,000;
}
```

Step 4: User withdraws all 100,000 tokens in Slot 121

```
// In client_community.rs update() during withdrawal
if available_tokens != self.header.drvs_tokens {
    // 0 != 100,000 - TRUE

    community_state.header.upgrade()?.drvs_tokens -= 100,000;

    if self.header.slot <= community_state.header.voting_start_slot {
        //120 <= 100 - FALSE (since this is false, we never enter here and voting tokens are not updated
        // if block NOT ENTERED - `current_voting_tokens` stays at 100,000!
    }
    self.header.slot = 121;
    self.header.drvs_tokens = 0; // User now holds ZERO tokens
}
```

Since attacker has not voted for this round, he calls voting and casts vote with 100000 tokens which he does'nt even posses

```
// In voting.rs
if data.voting_counter != community_state.header.voting_counter {
    // 2 == 2 - Check passes
}

if client_community_state.header.slot <= community_state.header.voting_start_slot {
    // 121 <= 100 - FALSE, we dont enter here
} else {
    voting_tokens = client_community_state.header.current_voting_tokens; // 100,000! (stale tokens)
}

if voting_tokens > 0 {
    if client_community_state.header.last_voting_counter >=
    ↪  client_community_state.header.current_voting_counter {
        // 1 >= 2 - FALSE, check passes
    }

    // User votes with 100000 tokens that he does'nt even hold
    match data.choice {
        VoteOption::DECREMENT => {
            community_account_header.voting_decr += 100,000; //
        }
        VoteOption::INCREMENT => {
            community_account_header.voting_incr += 100,000; //
        }
        _ => community_account_header.voting_unchange += 100,000, //
    }

    client_community_state.header.last_voting_counter = 2;
    client_community_state.header.last_voting_tokens = 100,000;
```

```
}
```

Impact: Users can cast votes without even holding tokens.

Recommended Mitigation: recommendation can be made once we're more days into audit and have knowledge of whole system.

Deriverse Fixed in commit: https://github.com/deriverse/protocol-v1/commit/aa2c1caaaa32b05f63d50056c50040d0ede

Cyfrin: Verified.

7.1.2 Missing Version Validation for Private Clients Account in `new_private_client`

Description: The `new_private_client()` function fails to validate that the provided `private_clients_acc` account matches the protocol version specified in `root_state`.

Since different protocol versions use different PDA seeds (which include the version), each version has its own isolated `private_clients_acc`. However, the function does not verify this correspondence, potentially allowing cross-version account misuse.

```
// src/program/processor/new_root_account.rs:291
let seed = get_seed_bytes(version, account_type::PRIVATE_CLIENTS);
// get_seed_bytes includes version in the seed (lines 73-77)
let bump_seed = check_new_program_account(
    private_clients_acc,
    &drvs_auth,
    program_id,
    &seed,
    AccountType::PrivateClients,
)?;
```

However, in `new_private_client()`, the function only validates:

1. The admin is the operator
2. Private mode is enabled
3. Wallet owner is system program

It **does not validate**:

- `private_clients_acc.owner` matches `program_id`
- `private_clients_acc` discriminator tag is `PRIVATE_CLIENTS`
- `private_clients_acc` discriminator version matches `root_state.version`
- `private_clients_acc` PDA address matches the expected address for `root_state.version`

```
// src/program/processor/new_private_client.rs:70-100
let private_clients_acc: &AccountInfo<'_> = next_account_info!(account_iter)?;
// ...
let root_state: &RootState = RootState::from_account_info(root_acc, program_id)?;

//  No validation of `private_clients_acc` here

if !root_state.is_private_mode() {
    bail!(DeriverseErrorKind::MustBeInPrivateMode);
}

// Directly uses private_clients_acc without version check
let private_clients = private_clients_acc
    .try_borrow_data()
    .map_err(|err| drv_err!(err.into()))?;
```

Impact: Cross-Version Account Manipulation: An operator authorized for one protocol version (e.g., version 1) could potentially pass a `private_clients_acc` from a different version (e.g., version 2), allowing unauthorized modification of another version's private client queue. This violates the intended isolation between protocol versions.

Recommended Mitigation: Verify the account address matches the expected PDA derived from `root_state.version`

Deriverse: Fixed in commit 6165c9a.

Cyfrin: Verified.

7.1.3 Airdrop Implementation Issues: Unvalidated Ratio and Potential Transfer Direction Mismatch

Description: The `airdrop` function is intended to convert user-earned `points` into DRVS tokens. However, the implementation has the following issues:

Issue 1: Unvalidated Ratio Parameter

The `AirdopOnChain` trait implementation only validates data format, not the `ratio` value:

```
impl AirdopOnChain for AirdropData {
    fn new(instruction_data: &[u8]) -> Result<&Self, DeriverseError> {
        bytemuck::try_from_bytes::<Self>(instruction_data)
            .map_err(|_| drv_err!(InvalidClientDataFormat))  // Only format check
    }
}
```

The `ratio` is then used directly without any bounds checking:

```
amount = ((amount as f64) * data.ratio) as u64;  // No validation on ratio!
```

This allows users to set `ratio` to arbitrary values (e.g., `1,000,000.0`, etc.), this could lead to token supply inflation or other calculation errors.

Issue 2: Potential Transfer Direction Mismatch (Requires Team Confirmation)

The current implementation transfers tokens FROM the user(`drvs_client_associated_token_acc`) to the program(`drvs_program_token_acc`):

```
let transfer_to_taker_ix = spl_token_2022::instruction::transfer_checked(
    &spl_token_2022::id(),
    drvs_client_associated_token_acc.key,  // FROM: User's account
    drvs_mint.key,
    drvs_program_token_acc.key,            // TO: Program's account
    signer.key,                            // Authority: User
    &[signer.key],
    amount,
    decs_count as u8,
)?;

invoke(
    &transfer_to_taker_ix,
    &[
        token_program.clone(),
        drvs_client_associated_token_acc.clone(),  // User account (source)
        drvs_mint.clone(),
        drvs_program_token_acc.clone(),            // Program account (destination)
        signer.clone(),                            // User signature
    ],
)?;
```

This is inconsistent with:

- The function name "airdrop" which typically implies sending tokens to users
- The expected behavior where users get rewarded for their accumulated points

Normally, the protocol should

- either directly transfer tokens to the user
- or mint tokens to the user,
- or mint them and immediately deposit them, followed by updating the client state with `client_state.add_asset_tokens(amount as i64)?;`.

Given that, this issue still requires the team's confirmation.

Impact:

- With unvalidated `ratio`, users could set extremely large values (e.g., `1,000,000.0`)
- The implementation could be inconsistent with the initial design, causing loss for the users.

Recommended Mitigation:

1. If `ratio` should be protocol-controlled, remove it from instruction data and calculate it based on protocol state.
2. If the `Potential Transfer Direction Mismatch` is confirmed, it is recommended to replace the transfer logic. If it's not a bug, the function name and documentation should be updated to reflect this behavior clearly.

deriverse: Fixed in commit f03ba7.

Cyfrin: Verified.

7.1.4 `update_records` always resets `fees_ratio` to 1 whenever a new currency is added

Description: The `update_records` function is used to update the `client_community_acc` by including more `ClientCommunityRecord` for every currency token. However, within this function the `fees_ratio` is always being reset to `1.0` whenever a new currency is added. This occurs because the function iterates over all existing `ClientCommunityRecord` entries and sets the `fees_ratio` to `1.0` during each iteration.

```
            self.data = unsafe {
                Vec::from_raw_parts(
                    dividends_ptr as *mut ClientCommunityRecord,
                    self.header.count as usize,
                    self.header.count as usize,
                )
            };
            for (d, b) in self.data.iter_mut().zip(community_state.base_crncy.iter()) {
                d.crncy_token_id = b.crncy_token_id;
                d.fees_ratio = 1.0;
            }
```

Scenario:

1. The user makes a prepayment of x currency tokens and becomes eligible for a 50% discount.
2. Later, a new currency is added by admin.
3. The user then deposits some amount of DRVS tokens. However, when the `update_records` function is called, the addition of the new currency causes the `fees_ratio` to reset to 1.0, thereby removing the user's previously earned discount.

Impact: This will result in a loss for users who have already made a prepayment for a fee discount, as they would now be required to pay higher fees.

Recommended Mitigation: Iterate over only newly added ClientCommunityRecord entries to mitigate this issue.

Deriverse: Fixed in commit 66c878.

Cyfrin: Verified.

7.1.5 Stale Funds Data in Leverage Validation Due to Missing Funding Rate Settlement

Description: In `perp_change_leverage`, the code calls `check_soc_loss` but omits `check_funding_rate` before validating leverage constraints. As a result, `check_client_leverage` uses stale funds data when computing evaluation and checking leverage limits. This can allow invalid leverage changes.

In `perp_change_leverage.rs`:

```
    engine.check_soc_loss(client_state.temp_client_id)?;
    if engine.check_short_margin_call()? < MAX_MARGIN_CALL_TRADES {
        engine.check_long_margin_call()?;
    }
    engine.check_rebalancing()?;
    engine.change_edge_px(client_state.temp_client_id);

    engine.check_client_leverage(client_state.temp_client_id)?;
```

However, there should be `check_funding_rate` call before leverage validation to ensure the `fund` is up-to-date.

```
    pub fn check_funding_rate(&mut self, temp_client_id: ClientId) -> Result<bool, DeriverseError> {
        let info = unsafe { &mut *(self.client_infos.offset(*temp_client_id as isize)) };
        let info5 = unsafe { &mut *(self.client_infos5.offset(*temp_client_id as isize)) };
        let perps = info.total_perps();
        let mut change = false;
        if perps != 0 {
            if self.state.header.perp_funding_rate != info5.last_funding_rate {
                let funding_funds = -(perps as f64
                    * (self.state.header.perp_funding_rate - info5.last_funding_rate))
                    .round() as i64;
                if funding_funds != 0 {
                    info.add_funds(funding_funds).map_err(|err| drv_err!(err))?;
                    info5.funding_funds = info5
                        .funding_funds
                        .checked_add(funding_funds)
                        .ok_or(drv_err!(DeriverseErrorKind::ArithmeticOverflow))?;
                    self.state.header.perp_funding_funds -= funding_funds;
                    change = true;
```

If funding rate has changed since last settlement, info.funds is stale, leading to incorrect evaluation and leverage constraint checks.

```
    engine.check_client_leverage(client_state.temp_client_id)?;

    pub fn check_client_leverage(&self, temp_client_id: ClientId) -> DeriverseResult {
        let info = unsafe { &*(self.client_infos.offset(*temp_client_id as isize)) };
        let info2 = unsafe { &*(self.client_infos2.offset(*temp_client_id as isize)) };
        let evaluation = self.calculate_evaluation(info)?;
        let perp_value = self.calculate_perp_value(info);

        self.check_client_leverage_from(info, info2, evaluation, perp_value)
    }
```

In contrast, the `new_perp_order` correctly calls `check_funding_rate`:

```
    engine.check_funding_rate(client_state.temp_client_id)?;
    engine.check_rebalancing()?;
    engine.change_edge_px(client_state.temp_client_id);
```

```
    engine.check_client_leverage_shift(
        perp_client_info,
        perp_client_info2,
        old_leverage,
        old_perps,
    )?;
```

Also, `perp_order_cancel` also lacks this update.

Impact: Leverage validation may be based on outdated funding payments.

Recommended Mitigation: Add `engine.check_funding_rate(client_state.temp_client_id)?;` to ensure ensures `info.funds` reflects the latest funding rate settlements before leverage validation.

Deriverse: Fixed in commit 74f9650.

Cyfrin: Verified.

7.1.6 Fee Discount Calculation Ignores Base Currency Value Differences

Description: The fee discount calculation in `fees_deposit` uses the raw token amount divided by decimal factor, without considering the actual value differences between base currencies. This can lead to unfair discount distribution when multiple base currencies with different values are supported.

In `fees_deposit`, the `prepayment` amount is calculated as:

```
let dec_factor = get_dec_factor(community_state.base_crncy[crncy_index].decs_count) as f64;
let prepayment = data.amount as f64 / dec_factor;
let fees_discount = community_state.fees_discount(prepayment);
```

The calculation only normalizes for decimal places but does not account for the actual value of different base currencies. For example:

- Depositing 1000 USDC (worth $1000)
- Depositing 1000 tokens of a low-value base currency (worth $0.001 each = $1 total)
- Deposit 1000 SOL would be quite impossible

Both would receive the same discount rate, despite a 1000x value difference.

This becomes more problematic as **the protocol expands to support additional base currencies through governance, as mentioned in the documentation. Different base currencies may have vastly different market values, but the current implementation treats them equally based on raw token count.**

From the documentation https://deriverse.gitbook.io/deriverse-v1/launchpad/launchpad#supported-base-currency:

```
Supported Base Currency
Current Support:

USDC: Circle USD Stablecoin

Future Expansion:

- Additional base currencies may be added through governance

- Multi-denomination support under consideration

- Community can propose new base currencies
```

Impact: Unfair discount distribution: Users depositing low-value base currencies can achieve the same discount thresholds as users depositing high-value currencies with much less actual value

Recommended Mitigation: Store a value normalization factor for each base currency that represents its relative value.

Deriverse: Fixed in commit 35602457.

Cyfrin: Verified.

7.1.7 Missing ps check causes Dividend Loss

Description: In the perp engine's fee distribution logic, both `match_ask_orders` and `match_bid_orders` functions allocate pool fees without checking if there is any liquidity provider in the spot pool. This causes a loss of dividends for DRVS token holders when there is no liquidity provider.

The code always allocates a portion of fees to `pool_fees` regardless of whether the `ps` is zero or not:

```
let delta = self.state.header.protocol_fees - prev_fees;
let pool_fees = ((1.0 - self.spot_pool_ratio) * delta as f64) as i64;
self.state.header.protocol_fees -= pool_fees;
self.state.header.pool_fees = self
    .state
    .header
    .pool_fees
    .checked_add(pool_fees)
    .ok_or(drv_err!(DeriverseErrorKind::ArithmeticOverflow))?;
```

Impact: When `ps == 0`, fees that should be 100% allocated to `protocol_fees` (for dividends) are incorrectly split, with `(1.0 - spot_pool_ratio) * delta` going to `pool_fees` instead.

DRVS token holders receive less in dividends than they should when there is no liquidity provider, as a portion of fees is incorrectly allocated to the pool.

Recommended Mitigation: Add a check for `ps == 0` before allocating pool fees in **both** `match_ask_orders()` and `match_bid_orders()` functions:

```
let delta = self.state.header.protocol_fees - prev_fees;
let pool_fees = if self.state.header.ps == 0 {
    0  // No pool allocation when there is no liquidity provider
} else {
    ((1.0 - self.spot_pool_ratio) * delta as f64) as i64
};
self.state.header.protocol_fees -= pool_fees;
self.state.header.pool_fees = self
    .state
    .header
    .pool_fees
    .checked_add(pool_fees)
    .ok_or(drv_err!(DeriverseErrorKind::ArithmeticOverflow))?;
```

This ensures that when there is no liquidity provider in the spot pool (`ps == 0`), all fees remain in `protocol_fees` and are properly distributed as dividends to DRVS token holders via the `dividends_allocation` function.

Note: This fix must be applied to **both** `match_ask_orders()` and `match_bid_orders()` functions, as they both contain the same buggy logic.

Deriverse: Fixed in commit 5dd59a.

Cyfrin: Verified.

7.1.8 Price manipulation during initial ps minting can cause user losses

Description: When a new instrument is created and the pool is empty (`ps == 0`), the first liquidity provider's deposit amount is calculated using a price derived from `last_px`, `best_bid`, and `best_ask`. A malicious user can

manipulate `best_bid` or `best_ask` by placing orders before the first LP provider, causing the first LP provider to deposit funds at an incorrect price, resulting in user fund loss.

Scenario:

1. A new instrument is created with `last_px = 1000`, representing a fair initial price.
2. An attacker places a very low ask order (e.g., price = 1) before any liquidity provider participates.
3. The first liquidity provider attempts to add liquidity, and the price is computed as `px_f64 = last_px.max(best_bid).min(best_ask)`, resulting in `px_f64 = 1000.max(0).min(1) = 1`, causing the LP provider to provide currency tokens based on this manipulated price.
4. Result: the liquidity provider ends up supplying significantly more asset tokens than intended based on the true fair-market price. later, the attacker can swap asset for currency at a more favorable price and extract value, causing a loss to the honest participant.

Impact: This may lead to user losses because the liquidity must be supplied at a price that has been manipulated.

Recommended Mitigation: A potential mitigation is to enable users to choose the asset and currency amounts during the initial liquidity provision, similar to the mechanism used in Uniswap V2.

Deriverse: Fixed in commit 66c878, cf0573.

Cyfrin: Verified.

7.1.9 Incorrect Fee Scaling in `spot_lp`

Description: The fee calculation in the `spot_lp` instruction uses an incorrect decimal factor when converting `last_px` to a human-readable price when trying to determine how much money is needed to purchase the minimum quantity of an asset. The code divides `last_px` by `get_dec_factor(asset_token_decs_count)` instead of the correct decimals(should be `9 + instr_state.asset_token_decs_count - instr_state. crncy_token_decs_count`), leading to incorrect fee calculations when `9 != crncy_token_decs_count`.

In `src/program/processor/spot_lp.rs`, the fee calculation incorrectly converts last_px:

```
fees = 1
    + ((instr_state.header.last_px as f64
        / get_dec_factor(instr_state.header.asset_token_decs_count) as f64)
        as i64)
        .max(1);
```

`last_px` precision: According to `src/program/spot/engine.rs` and `src/state/instrument.rs`, `last_px` is calculated with precision `dec_factor = 10^(9 + asset_token_decs_count - crncy_token_decs_count)`. ** So the final precision for `px` would be `9`.**

```
        let mut px = (self.state.header.crncy_tokens as f64 * self.df
            / self.state.header.asset_tokens as f64) as i64; // <= crncy_token_decs_count + 9 +
            ↪  asset_token_decs_count - crncy_token_decs_count - asset_token_decs_count = 9
```

In fact the `px` should be divided by `dec_factor` which is: `get_dec_factor ( 9 + instr_state.asset_token_decs_count - instr_state. crncy_token_decs_count)` but currently it's only `get_dec_factor (asset_token_decs_count)`

When `crncy_token_decs_count != 9` (e.g., `crncy_token_decs_count = 6`), the fee will be calculated incorrectly.

Example: Suppose `asset_token_decs_count = 8` and `crncy_token_decs_count = 7`, and `asset_token = 2 crncy (ignoring decimals)`. Using the current calculation:

```
px = 2 * 1e9
(instr_state.header.last_px as f64
        / get_dec_factor(instr_state.header.asset_token_decs_count) as f64) = 2*1e9 / 1e8 = 20.
```

The code suggests that ~20 minimum units of crncy are required to purchase 1 minimum unit of asset. In reality, the correct value should be 0.2 minimum units of crncy, reflecting the difference in decimal precision.

Impact: When crncy_token_decs_count != 9, users are charged incorrect fees, which may lead to overpayment.

Recommended Mitigation: Update the fee calculation to use the correct decimal factor.

Deriverse: Fixed in commit 65dfb8.

Cyfrin: Verified.

7.1.10 DoS attack exhausting account space and halting spot trading

Description: The temporary spot client ID allocation system reserves space in both client_infos_acc and client_infos2_acc accounts when orders are created, but deallocation only occurs when specific cleanup functions are invoked. An attacker can exploit this by creating many small orders and avoiding cleanup paths, gradually exhausting available account space and causing a DoS condition.

Attack Vector: An attacker can create numerous small limit orders across multiple accounts, never invoke move_spot_avail_funds, and avoid any actions that would trigger finalize_spot. As a result, each temporary client ID remains allocated indefinitely. Over time, this leads to progressive exhaustion of memory space in both the client_infos_acc and client_infos2_acc accounts, preventing legitimate users from operating normally.

A Solana account has a maximum storage limit of a 10 megabytes. Based on the size of SPOT_CLIENT_INFO_SIZE(32 bytes), the client_infos_acc and client_infos2_acc account can store approximately 312,500 entries. For an instrument such as XY/USDC, where the asset XY is priced at 1 USDC, a user would need to place an order of at minimum 2 XY tokens to create a single temporary client ID. To reach the full 312,500 allocations, the attacker would theoretically need 2 × 312,500 tokens.

Impact: It can cause a DoS of the spot functionality, but it would require the attacker to have sufficient funds to execute the attack.

Recommended Mitigation: Deallocate the space as soon as the user's last order has been executed, or provide a dedicated deallocation mechanism that allows finalize_spot to be invoked for users when needed.

Deriverse: Fixed in commit e0773a, 105e46.

Cyfrin: Verified.

7.1.11 Attacker can cause losses to others by partially closing their position at an unfavorable price

Description: In our new perp order flow, we verify whether the leverage is valid using check_client_leverage_from. However, this check is only executed if any of the following conditions are true:

```
current_perps.signum() != old_perps.signum()
current_leverage > old_leverage
old_leverage == i64::MAX as f64
```

A user can exploit this behavior to sell their position at an artificially unfavorable price(a very low price when long, or a very high price when short). This can result in socialized losses being imposed on other users.

Example: A user is able to create a new order only a few lamports away from their liquidation point. For example, when the user is long, they may create an ask order extremely close to liquidation and potentially exploit the mechanism.

Specifically, if there are no bid orders—or only negligible ones—the user can place an ask order at a very low price. Immediately afterward, using another address, they can purchase their own ask order by creating a bid order at the same low price. This causes loss, which are then socialized to other participants.

This exploit relies on the condition that no meaningful bid liquidity is present.

Scenerio: A user has a 1 BTC long position with 10× leverage. Their funds are –90k, and the edge price is 90k, with a zero-liquidation threshold.

Now, suppose there are no bid orders, and both the mark price and index price is $90,100. The user then creates an ASK order for 0.5 BTC at $80,000. In the portfolio(perp), the position becomes 0.5 BTC. In the order, the perp becomes 0.5 BTC.

At this point: The old leverage is 90k / 100 = 900, and the old total perp is 1 BTC. The new leverage is also unchanged (still 90k / 100 = 900). old_leverage is not i64::MAX as f64. And importantly, both the current and old perp values have the same signum.

`check_client_leverage_from` will not run because the sign remains the same, and the current leverage is equal to the previous leverage — which itself was not at the maximum.

After this, a user can create a bid order from a different address and purchase the ask order at 80k, enabling them to exploit the mechanism.

Later, when the user's original position of 0.5 BTC is liquidated, it will generate a loss that becomes socialized across other users.

Impact: The impact is high, as it results in losses for other users.

Recommended Mitigation:

1. Add a check to ensure that if `margin_call` is false and loss is greater than zero, the transaction should revert.

```
    if loss > 0 && !args.margin_call {
          return Err;
    }
```

2. When a user is in a long position and creates an ask order, the order price should be greater than the critical price. Conversely, when the user is in a short position, the bid order price should be lower than the critical price.

Deriverse: Fixed in commit 26d87c, 319890.

Cyfrin: Verified.

7.1.12 Makers' rebates are not paid in case of swap

Description: When orders are matched, we choose between AMM and orderbook based on which offers better prices. In case of `swap` when orderbook doesnt exist, we trade with only AMM, when it exists we choose better of both.. we create a temporary client's primary account.. while looking for available orders we pass `client-community` as `none`

```
//inside swap.rs
    if buy {
        if price > px || engine.cross(price, OrderSide::Ask) {
            (q, traded_sum, trades, _) =
                engine.match_orders(price, qty, &mut client_state, None, OrderSide::Ask)?;
        }
```

inside `engine.rs`'s `match_orders()` function, `fee-rate & ref-discounts` are zero because `client-community` was `None`

```
        let fill_static_args = FillStaticArgs {
            fee_rate,
            ref_discount,
            taker_client_id: client.id,
            side,
        };
```

Incase when orderbook offered better prices, we go through that route.. inside `match_orders();` here

```
                    if remaining_qty > 0 {
                        self.fill(
                            &mut line,
```

```
                        &mut remaining_qty,
                        &mut sum,
                        &mut trades,
                        &mut total_fees,
                        &mut total_rebates,
                        &mut fees_prepayment,    // 0
                        &fill_static_args, // contains fee rate as 0
                        //ref_discount,
                        //client.id,
                        //fee_rate,
                        //side,
                    )?;
                }
            }
```

Inside `fill()`, since we had `fill_static_args.fee_rate == 0.0` we dont charge/deduct fee and dont give rebates to makers because both(fee rate and rebates) came out to be zero in this case.... Makers's orders are filled without receiving rebate and user traded without paying protocol fee.

```
            let (fees, rebates) = if fill_static_args.fee_rate == 0.0 {
                (0, 0)
            } else {
                (
                    self.get_fees(
                        fees_prepayment,
                        traded_crncy,
                        fill_static_args.fee_rate,
                        fill_static_args.ref_discount,
                    )?,
                    ((traded_crncy as i128 * self.rebates_rate) >> DEC_PRECISION) as i64,
                )
            };
```

Impact: Protocol loses intended fees from swaps & makers dont receive rebates.

Recommended Mitigation: Change the current condition to this:

```
let (fees, rebates) = if self.fee_rate == 0.0 {
    (0, 0)
} else {
    (
        self.get_fees(
            fees_prepayment,
            traded_crncy,
            fill_static_args.fee_rate,
            fill_static_args.ref_discount,
        )?,
        ((traded_crncy as i128 * self.rebates_rate) >> DEC_PRECISION) as i64,
    )
};
```

Deriverse Fixed in commit: b368de

Cyfrin: Verified.

7.1.13 `perp-change-leverage` uses stale `perp-underlying-px`

Description: The perp_change_leverage function fails to call engine.state.set_underlying_px(accounts_iter)? before performing various checks which involve upated/fresh `perp_underlying_px` like check_long_margin_call, check_short_margin_call & check_client_leverage. This function is responsible for synchronizing the `perp_underlying_px` field with the current last_px (since oracle support is disabled). Without this call, all subsequent calculations use a stale `perp_underlying_px` value from whenever it was last updated by another function.

```
    //@audit  present in perp_withdraw, perp_order_cancel, perp_mass_cancel, etc.
    //engine.state.set_underlying_px(accounts_iter)?;
    engine.check_soc_loss(client_state.temp_client_id)?;
    if engine.check_short_margin_call()? < MAX_MARGIN_CALL_TRADES {
        engine.check_long_margin_call()?;
    }
    engine.check_rebalancing()?;
    engine.change_edge_px(client_state.temp_client_id);

    engine.check_client_leverage(client_state.temp_client_id)?;
```

In contrast other functions which invlove `perp_underlying_px` correctly call set_underlying_px to update underlying price to last price (`new_perp_order`, `perp-withdraw`, `perp-order-cancel`, `perp-mass-cancel` etc) NOTE: similar issue is in `perp-statistics-reset`

Impact: All the following calculations and checks involving `perp_underlying_px` would be based on stale value which may be problematic for traders as it may favor them or work against them.

Recommended Mitigation: Add the missing price update which syncs `perp_underlying_px` with `last_px`

```
engine.state.set_underlying_px(accounts_iter)?;
```

Deriverse: Fixed in commit: https://github.com/deriverse/protocol-v1/commit/4f7bc8ac68325aa93b339ff91c0ac794ea17ffd9

Cyfrin: Verified.

7.1.14 Shorts can withdraw full available funds instead of being restricted to margin call limits in `perp-withdraw`

Description: In perp_withdraw.rs, the margin call detection logic fails to detect for shorts ,where is_long_margin_call() is called twice instead of checking both long and short positions:

```
        engine.check_long_margin_call()?;
        engine.check_short_margin_call()?;
        //@audit both are long, one should be short
        let margin_call = engine.is_long_margin_call() || engine.is_long_margin_call();
        if !margin_call {
            engine.check_rebalancing()?;
}
```

When a user has a short perpetual position that is in margin call (underwater/undercollateralized), the `margin_call` variable incorrectly evaluates to false because:

- `is_long_margin_call()` returns `false` for short positions
- The duplicate call also returns `false`
- Result: [margin_call = false || false = false]

Impact:

1. Bypassed Withdrawal Restrictions The withdrawal amount calculation differs based on margin call status:

```
let amount = if margin_call {
    // Limited withdrawal - only excess margin above requirements
    let margin_call_funds = funds.min(engine.get_avail_funds(client_state.temp_client_id, true)?);
    if margin_call_funds <= 0 {
        bail!(ImpossibleToWithdrawFundsDuringMarginCall);
    }
    // Restricted amount
} else if data.amount == 0 {
    funds  // Full funds available - NO RESTRICTIONS
```

```
} else {
    // Normal withdrawal
};
```

This lets users with underwater short positions withdraw full available funds instead of being restricted to margin call limits. This allows extraction of collateral that should be locked to cover their short position's potential losses.

2. Incorrect Rebalancing Execution When `margin_call` is incorrectly `false` for short positions in margin call, `check_rebalancing()`is called when it shouldn't be..

Recommended Mitigation: Replace current condition to this:

```
let margin_call = engine.is_long_margin_call() || engine.is_short_margin_call();
```

Deriverse: Fixed in commit: https://github.com/deriverse/protocol-v1/commit/4f7bc8ac68325aa93b339ff91c0ac794ea17ffd9

Cyfrin: Verified.

7.1.15 Margin call detection functions ignore liquidation threshold

Description: The `is_long_margin_call()` and `is_short_margin_call()` functions used to determine whether margin calls are active do not account for the `liquidation_threshold` parameter. This creates a discrepancy between when the system detects margin calls and when actual margin call occur.

```
pub fn is_long_margin_call(&self) -> bool {
    let root = self.long_px.get_root::<i128>();
    if root.is_null() {
        false
    } else {
        self.state.header.perp_underlying_px < (root.max_node().key() >> 64) as i64
    }
}

pub fn is_short_margin_call(&self) -> bool {
    let root = self.short_px.get_root::<i128>();
    if root.is_null() {
        false
    } else {
        self.state.header.perp_underlying_px > (root.min_node().key() >> 64) as i64
    }
}
```

Actual Liquidation Logic:

```
pub fn check_long_margin_call(&mut self) -> Result<i64, DeriverseError> {
    let mut trades = 0;
    // Applies liquidation threshold
    let margin_call_px = (self.state.header.perp_underlying_px as f64
        * (1.0 - self.state.header.liquidation_threshold)) as i64;

    loop {
        let root = self.long_px.get_root::<i128>();
        if root.is_null() || trades >= MAX_MARGIN_CALL_TRADES {
            break;
        }
        let node = root.max_node();
        let px = (node.key() >> 64) as i64;

        // Compares edge price to threshold-adjusted price
        if px > margin_call_px {
            // ... liquidation logic ...
        }
```

```
    }
    Ok(trades)
}
```

Same `liquidation_threshold` is applied during `check_short_margin_call` also.

Margin call remains false even when margin calls are occurring, which causes the following issues:

- The `perp_spot_price_for_withdrawal` freezing mechanism fails to activate when it should.
- In the perpetual withdrawal flow, `get_avail_funds` is called only with `margin_call` false.
- Rebalancing is invoked even during active margin-call conditions.

Impact: This allows users to withdraw more than they should during active margin calls, effectively bypassing the intended withdrawal restrictions.

Recommended Mitigation: Use the liquidation threshold into margin-call detection functions to ensure accurate margin-call detection and prevent incorrect withdrawal and rebalancing behavior.

Deriverse: Fixed in commit 4f7bc8.

Cyfrin: Verified.

7.1.16 Multiple inconsistencies in `sell-market-seat`

Description: The sell_market_seat contains two inconsistencies when compared to other perp functions in the codebase:

Issue 1: stale underlying price used for funding rate calculation:

- The function calls `change_funding_rate()` without first updating the underlying price via `set_underlying_px()`:

```
// sell_market_seat.rs
let mut engine = PerpEngine::new(
    &ctx,
    signer,
    system_program,
    community_state.perp_fee_rate(),
    community_state.margin_call_penalty_rate(),
    community_state.spot_pool_ratio(),
)?;

let instrument: &mut InstrAccountHeader = InstrAccountHeader::from_account_info(
    ctx.instr_acc,
    program_id,
    Some(data.instr_id),
    root_state.version,
)?;

// @audit change_funding_rate() called WITHOUT set_underlying_px() first, stale price is being used
engine.change_funding_rate();
```

The funding rate calculation depends on `perp_underlying_px` to determine the deviation between perpetual and spot prices. Without updating this value, the global funding rate is calculated using stale price data.

Issue 2: Missing client's funding rate update: After calling `change_funding_rate()`, the function never calls `check_funding_rate(client_state.temp_client_id)` to apply pending funding payments to the exiting client:

```
engine.change_funding_rate();  // Updates global funding rate

let info = client_state.perp_info()?;

//@audit check_funding_rate(client_state.temp_client_id) is NOT called
```

```
// Pending funding is NOT applied to client's funds

if info.perps != 0 || info.in_orders_funds != 0 || info.in_orders_perps != 0 {
    bail!(ImpossibleToClosePerpPosition);
}

// Using stale funds value that doesn't include pending funding
let collactable_losses = info.funds().min(client_state.perp_info4()?.soc_loss_funds);
engine.state.header.perp_insurance_fund += collactable_losses;

// Client receives funds without pending funding being applied
client_state.add_crncy_tokens((info.funds() - collactable_losses).max(0))?;
```

The`check_funding_rate()` function applies any pending funding payments based on the client's position:

- When funding_rate > 0: longs pay shorts
- When funding_rate < 0: shorts pay longs Even though the function requires `info.perps == 0` (no open position), the client may still have uncollected funding payments from when they previously held a position that have not yet been settled to their `info.funds` balance.

Impact: The stale underlying price causes the protocol to apply an incorrect global funding rate & by not updating client's funding payments before seat withdrawal we use stale `info.funds` which results in users either receiving less or more money than they should during position closure,

Recommended Mitigation: Call both functions prior, such that global rate and user's funds are fresh.

Deriverse: Fixed in commit 319890.

Cyfrin: Verified.

7.1.17 Improper header handling in `SpotFeesReport` logging causes DoS on swap instruction

Description: In `match_orders`, `ref_payment` handles `client.header` being None by defaulting to 0, but `ref_client_id` uses `ok_or` and returns an error if `header` is None. This causes the function to fail when `total_fees` > 0 and `header` is None.

```
            solana_program::log::sol_log_data(&[bytemuck::bytes_of::<SpotFeesReport>(
                &SpotFeesReport {
                    tag: log_type::SPOT_FEES,
                    fees: total_fees,
                    ref_payment,
                    ref_client_id: client
                        .header
                        .as_ref()
                        .ok_or(drv_err!(DeriverseErrorKind::ClientPrimaryAccountMustBeSome))?
                        .ref_client_id,
                    ..SpotFeesReport::zeroed()
                },
            )]);
```

This will always happen when the user is using the swap instruction.

Impact: The impact is high, as this issue results in a permanent DOS whenever the swap instruction is executed and `total_fees > 0` is true.

Recommended Mitigation: Consider handling this gracefully instead of reverting when the header is None.

Deriverse: Fixed in commit 4f7bc8.

Cyfrin: Verified.

7.1.18 Missing Signer Verification in `voting_reset` Function Allows Unauthorized Execution

Description: The `voting_reset` function in `src/program/processor/voting_reset.rs` only verifies that the `admin` account's public key matches the operator address, but does not verify that the `admin` account is actually a signer of the transaction. **This allows an attacker to execute the `voting reset` instruction by passing account that matches the operator address but no signing is required, resetting critical voting parameters without proper authorization.**

```
let root_state: &RootState = RootState::from_account_info(root_acc, program_id)?;

if root_state.operator_address != *admin.key {
    bail!(InvalidAdminAccount {
        expected_address: root_state.operator_address,
        actual_address: *admin.key
    })
}
```

Impact: An attacker could execute `voting_reset` without possessing the operator's private key, resetting critical voting parameters without proper authorization.

Recommended Mitigation: Add a signer verification check before the operator address check.

```
if !admin.is_signer {
    bail!(MustBeSigner {
        address: *admin.key
    })
}
```

Deriverse: Fixed in commit bfc0c96.

Cyfrin: Verified

7.1.19 Using the underlying price instead of the perp price can cause problems

Description: Currently, we use the perp price only for buy/sell operations and for funding rate calculations. However, when calculating the perp value we rely on the underlying price:

Scenerio with perp withdrawal: The mark price is 100k, and the index price is also 100k. A user buys a 1 BTC perpetual contract at 100k with 10× leverage. The notional position is 1 BTC, and the user's account balance becomes –90k. Later, the index price increases to 120k, while the mark price remains unchanged. At this point, the user is able to withdraw 18k, bringing their account balance to –108k. However, the actual perp price is still 100k for 1 BTC. However, the user's funds are –108k, and their position can only cover 100k

This issue arises because we are using the underlying price to calculate the perp value.

Impact: The impact is high, as it allows users to withdraw more funds than they should based on the actual perp price.

Recommended Mitigation: Consider using the perp price instead of the underlying price to mitigate this issue.

Deriverse: Fixed in commit 1ef948.

Cyfrin: Verified.

7.1.20 Spot price manipulation can lead to unfair liquidations

Description: When perpetual instruments are configured without an oracle feed, the system uses the spot market's `last_px` as the `perp_underlying_px` for liquidation calculations. The spot price (`last_px`) can be manipulated through order book orders or AMM trades, allowing attackers to trigger unfair liquidations of healthy perpetual positions. This vulnerability enables malicious actors to force liquidations at manipulated prices, causing significant financial losses to users.

When no oracle is configured, the spot price directly becomes the perpetual underlying price:

Liquidations are triggered based on the manipulated `perp_underlying_px`:

```
pub fn check_long_margin_call(&mut self) -> Result<i64, DeriverseError> {
    let margin_call_px = (self.state.header.perp_underlying_px as f64
        * (1.0 - self.state.header.liquidation_threshold)) as i64;

    // If edge_px > margin_call_px, position gets liquidated
    // ...
}

pub fn check_short_margin_call(&mut self) -> Result<i64, DeriverseError> {
    let margin_call_px = (self.state.header.perp_underlying_px as f64
        * (1.0 + self.state.header.liquidation_threshold)) as i64;

    // If edge_px < margin_call_px, position gets liquidated
    // ...
}
```

Impact: The impact is high, as healthy positions may be liquidated unfairly. Additionally, an attacker may be able to exploit this behavior for profit.

Recommended Mitigation: Consider using an external oracle or TWAP.

Deriverse: Fixed in commit fc0013.

Cyfrin: Verified.

7.2 Medium Risk

7.2.1 `create_account` can be dosed with pre-funding

Description: Instructions like `new_holder_account` create PDAs using `system_instruction::create_account` with `invoke_signed`. Per Solana system program rules, `create_account` fails if the target already **exists with non-zero lamports or data**.

An attacker can pre-calculate the PDA and transfer SOL to it, causing subsequent `create_account` to fail, blocking account initialization and the following logic flow.

```
let account_size = size_of::<HolderAccountHeader>();
invoke_signed(
    &system_instruction::create_account(
        holder_admin.key,
        holder_acc.key,
        Rent::default().minimum_balance(account_size),
        account_size as u64,
        program_id,
    ),
    &[holder_admin.clone(), holder_acc.clone()],
    &[&[HOLDER_SEED, holder_admin.key.as_ref(), &[bump_seed]]],
)
.map_err(|err| drv_err!(err.into()))?;
```

For Reference: https://x.com/r0bre/status/1887939134385172496

Note: the `system_instruction::create_account` also exists in `src/program/processor/new_holder_account.rs`, `src/program/processor/new_root_account.rs`, `src/program/create_client_account.rs`, `src/state/candles.rs`, `src/state/instrument.rs`, `src/state/perps/perp_trade_header.rs`, `src/state/spots/spot_account_header.rs`, `src/state/token.rs`

Impact:

- Any actor can block creation of critical PDAs (`holder accounts`, `root accounts`, `headers`, `tokens`, `etc`) by pre-funding them with a minimal lamport amount.

Proof of Concept:

```
/// Test to demonstrate PDA DOS attack vulnerability
///
/// This test simulates an attacker pre-calculating the PDA address
/// and pre-funding it to prevent the admin from creating the Holder Account
#[tokio::test]
async fn test_holder_account_dos_attack() {
    let program_id = Pubkey::from_str(PROGRAM_ID).unwrap();

    // Create program test environment
    let mut test = ProgramTest::new(
        "smart_contract",
        program_id,
        processor!(smart_contract::process_instruction),
    );

    // Setup admin and attacker accounts
    let admin_signer = Keypair::from_bytes(CLIENTS[0].kp).unwrap();
    let attacker = Keypair::new();

    // Add accounts to test environment with initial SOL balance
    test.add_account(
        admin_signer.pubkey(),
        Account {
            lamports: 10_000_000_000, // 10 SOL
            data: vec![],
```

```
        owner: system_program::id(),
        executable: false,
        rent_epoch: 0,
    },
);
test.add_account(
    attacker.pubkey(),
    Account {
        lamports: 10_000_000_000, // 10 SOL
        data: vec![],
        owner: system_program::id(),
        executable: false,
        rent_epoch: 0,
    },
);

// Attacker pre-calculates the PDA address using the same seeds as the protocol
let (pda_address, _) = Pubkey::find_program_address(
    &[HOLDER_SEED, admin_signer.pubkey().as_ref()],
    &program_id
);

println!(" Attacker pre-calculated PDA address: {}", pda_address);

// Start the test context
let mut ctx = test.start_with_context().await;

// Attacker pre-funds the PDA address to make it balance non-zero
let transfer_instruction = system_instruction::transfer(
    &attacker.pubkey(),
    &pda_address,
    1_000_000, // Transfer 0.001 SOL, mimic minimum balance
);

let transfer_transaction = Transaction::new_signed_with_payer(
    &[transfer_instruction],
    Some(&attacker.pubkey()),
    &[&attacker],
    ctx.last_blockhash,
);

let result = ctx.banks_client.process_transaction(transfer_transaction).await;
assert!(result.is_ok(), "Attacker should be able to pre-fund PDA address");

println!(" Attacker successfully pre-funded PDA address");

// Admin attempts to create Holder Account
println!("Admin attempts to create Holder Account");
let mut tx = Transaction::new_with_payer(
    &[Instruction::new_with_bytes(
        program_id,
        &[0], // NewHolderInstruction
        vec![
            AccountMeta {
                pubkey: admin_signer.pubkey(),
                is_signer: true,
                is_writable: true,
            },
            AccountMeta {
                pubkey: pda_address,
                is_signer: false,
                is_writable: true,
            },
```

```
                AccountMeta {
                    pubkey: system_program::ID,
                    is_signer: false,
                    is_writable: false,
                },
            ],
        )],
        Some(&admin_signer.pubkey()),
    );
    tx.sign(&[&admin_signer], ctx.last_blockhash);

    // Admin creation should fail because PDA is pre-occupied
    let result = ctx.banks_client.process_transaction(tx).await;

    if result.is_err() {
        println!(" DOS ATTACK SUCCESSFUL: Admin failed to create holder account");
    } else {
        println!(" Test failed: Admin creation should have failed but didn't");
    }
}
```

Test Output:

```
running 1 test
[2025-10-30T01:37:51.898533000Z INFO  solana_program_test] "smart_contract" builtin program
 Attacker pre-calculated PDA address: 5zAb3ZhCNjTwoMxK39fheR4fXbuArMJVN9bhaeQkZHrq
[2025-10-30T01:37:52.011771000Z DEBUG solana_runtime::message_processor::stable_log] Program
↪  11111111111111111111111111111111 invoke [1]
[2025-10-30T01:37:52.011914000Z DEBUG solana_runtime::message_processor::stable_log] Program
↪  11111111111111111111111111111111 success
 Attacker successfully pre-funded PDA address
Admin attempts to create Holder Account
[2025-10-30T01:37:52.013452000Z DEBUG solana_runtime::message_processor::stable_log] Program
↪  Drvrseg8AQLP8B96DBGmHRjFGviFNYTkHueY9g3k27Gu invoke [1]
[2025-10-30T01:37:52.013485000Z DEBUG solana_runtime::message_processor::stable_log] Program
↪  Drvrseg8AQLP8B96DBGmHRjFGviFNYTkHueY9g3k27Gu invoke [1]
[2025-10-30T01:37:52.013963000Z DEBUG solana_runtime::message_processor::stable_log] Program
↪  11111111111111111111111111111111 invoke [1]
[2025-10-30T01:37:52.014048000Z DEBUG solana_runtime::message_processor::stable_log] Program
↪  11111111111111111111111111111111 invoke [2]
[2025-10-30T01:37:52.014070000Z DEBUG solana_runtime::message_processor::stable_log] Create Account:
↪  account Address { address: 5zAb3ZhCNjTwoMxK39fheR4fXbuArMJVN9bhaeQkZHrq, base: None } already in use
[2025-10-30T01:37:52.014101000Z DEBUG solana_runtime::message_processor::stable_log] Program
↪  11111111111111111111111111111111 failed: custom program error: 0x0
{"code":100,"error":{"Custom":0},"location":{"file":"src/program/processor/new_holder_account.rs","line
↪  ":63},"msg":"System error Custom program error: 0x0"}
[2025-10-30T01:37:52.014285000Z DEBUG solana_runtime::message_processor::stable_log] Program
↪  Drvrseg8AQLP8B96DBGmHRjFGviFNYTkHueY9g3k27Gu failed: custom program error: 0x64
[2025-10-30T01:37:52.014303000Z DEBUG solana_runtime::message_processor::stable_log] Program
↪  Drvrseg8AQLP8B96DBGmHRjFGviFNYTkHueY9g3k27Gu failed: custom program error: 0x64
 DOS ATTACK SUCCESSFUL: Admin failed to create holder account
test instructions::test_holder_dos::test_holder_account_dos_attack ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out; finished in 0.24s
```

Recommended Mitigation: Do not rely on `create_account` for PDAs. Instead, support both fresh and pre-funded PDA flows by:

1. Funding (if needed),
2. allocating

3. assigning the PDA with invoke_signed.

Deriverse: Fixed in commit df95974 and 9b8e442.

Cyfrin: Verified.

7.2.2 Fixed token account size causes initialization failures for token accounts whose mints have token 22 extensions are active

Description: The `create_token` function in `token.rs` uses a hardcoded size of 165 bytes when creating SPL token accounts, regardless of whether the mint has Token 2022 extensions enabled. This will cause initialisation failures for mints with extensions that require additional account space.

```
let spl_lamports = rent.minimum_balance(165);
invoke(
    &system_instruction::create_account(
        creator.key,
        program_acc.key,
        spl_lamports,
        165, // @audit why is it hardcoded to 165?
        token_program_id.key,
    ),
    &[creator.clone(), program_acc.clone()],
)
```

The constant 165 bytes is the legacy SPL-Token size. Allocating only 165 bytes causes `initialize_account3` to fail with `InvalidAccountData`here

Impact:

- Token creation will fail for any Token 2022 mint with extensions

Recommended Mitigation: In the case when the mint's owner is token 2022 program, calculate the size first and then allocate that calculated space instead of a hardcoded 165 bytes.

```
let account_size = if token_program == TokenProgram::Token2022 {
    use spl_token_2022::extension::StateWithExtensions;
    StateWithExtensions::<spl_token_2022::state::Account>::try_get_account_len(mint)?
} else {
    165 // Standard SPL token account size
};

let spl_lamports = rent.minimum_balance(account_size);
invoke(
    &system_instruction::create_account(
        creator.key,
        program_acc.key,
        spl_lamports,
        account_size as u64,
        token_program_id.key,
    ),
    &[creator.clone(), program_acc.clone()],
)
```

Deriverse: Fixed in commit: https://github.com/deriverse/protocol-v1/commit/6e8b8b011e69c356a81e4cc1f8cbe14adf5bf0a6

Cyfrin: Verified.

7.2.3 Accounts may be created with incorrect rent-exemption due to `Rent::default` usage

Description: This dos vulnerability arises from the use of hardcoded `rent` parameters during account creation, leading to underfunded accounts that fail rent exemption requirements.

The flawed implementation relies on `Rent::default` instead of the on-chain `Rent sysvar` for rent calculations, creating accounts with insufficient funds for rent exemption.

```
    let rent = &Rent::default();
    ...
    let spl_lamports: u64 = rent.minimum_balance(165);
```

The implementation never reads the on-chain Rent `sysvar` (e.g., `Rent::get` or passing the rent `sysvar` account), so the computed minimum balances may not reflect the cluster's actual rent parameters.

Impact: As a result, accounts can be created non–rent-exempt, causing DOS.

Recommended Mitigation: It is recommended to use `Rent::get` instead of `Rent::default`.

Deriverse: Fixed in commit d319206.

Cyfrin: Verified.

7.2.4 `change_points_program_expiration` is permissionless

Description: The `change_points_program_expiration` function fails to verify that the `admin` account has actually signed the transaction. While the function checks that the provided admin account key matches the expected `operator_address` in the root state, it does not verify that the account is a signer using `admin.is_signer`. The comment indicates that the admin should be a signer (// *`create_account` can be dosed with pre-funding* - Admin (Signer)), but this requirement is not enforced in the code so any normal user can pass admin pubkey and execute this instruction without even real admin signing it.

```
// Only checks that the admin key matches, but not checking if the admin has actually signed the
↪  transaction;
if root_state.operator_address != *admin.key {
    bail!(InvalidAdminAccount {
        expected_address: root_state.operator_address,
        actual_address: *admin.key,
    });
}
// Missing: if !admin.is_signer { ... }
```

Impact: Unauthorised access to supposiely admin gated functionality.

Recommended Mitigation: Make sure the `admin` adress not only matches the stored address but also is the signer

```
    if !admin.is_signer {
        bail!(InvalidAdminAccount {
            expected_address: root_state.operator_address,
            actual_address: *admin.key,
        });
    }
```

Deriverse: Fixed in commit: 8a479a

Cyfrin: Verified.

7.2.5 Anyone can sell anyone else's market seat

Description: In the `sell_market_seat` function the `check_signer` parameter is incorrectly set to false, allowing any user to force-sell another user's market seat without proper authorization.

```
let mut client_state = ClientPrimaryState::new_for_perp(
    program_id,
    client_primary_acc,
    &ctx,
```

```
    signer,
    system_program,
    0,
    false,  // alloc = false (correct)
    false,  // @audit why false? can I supply others's
)?;
```

`new_for_perp` calls `ClientPrimaryAccountHeader::from_account_info_unchecked` since we have set the `check_signer` flag to false, the `from_account_info_unchecked` only makes sure that the account `signer` is signer, however it does not make sure that the signer account is indeed the one `wallet_address` stored in `ClientPrimaryAccountHeader`.. which means anyone could pass a anyone's valid `ClientPrimaryState` and sell their seat without consent.

Attack Scenario

- Attacker finds any `client_primary_acc` with a market seat
- Attacker call `sell_market_seat` with their own wallet as `signer` and Victim's `client_primary_acc`
- No validation occurs to check if the signer owns the client account
- Victim's market seat is sold and funds go to the victim's account, and the attacker closed victim's position without consent

Impact: Any user can close any other user's seats

Recommended Mitigation: Pass the `is_signer` flag set to true like we have done in `buy_market_seat`

```
// ...existing code...
let mut client_state = ClientPrimaryState::new_for_perp(
    program_id,
    client_primary_acc,
    &ctx,
    signer,
    system_program,
    0,
    false,
    true,   // check_signer = true
)?;
// ...existing code...
```

Deriverse Fixed in commit: 9a5678

Cyfrin: Verified.

7.2.6 Voting is allowed even after voting period's end time.

Description: The `voting()` function does not validate whether the voting period has ended before processing and recording votes. The function only checks the voting end time in the `finalize_voting()` call at the very end, but by that point the user's vote has already been cast and included in the voting tallies. This allows the last user to submit a vote even after the official voting period has expired, as long as they call the function before anyone else triggers the finalization.

```
//here vote has been included even if end time has been passed of this period
        match data.choice {
            VoteOption::DECREMENT => {
                community_account_header.voting_decr += voting_tokens;
            }
            VoteOption::INCREMENT => {
                community_account_header.voting_incr += voting_tokens;
            }
            _ => community_account_header.voting_unchange += voting_tokens,
        }
....
```

```
// the call to finalize voting is made later, so the late user's vote has been included
        community_state.finalize_voting(time, clock.slot as u32)?;
```

Impact:

- Malicious actors can cast votes after the voting period has officially ended
- Late voter(if is holding many tokens) gain knowledge of current vote tallies and can strategically vote to influence outcomes in his favor.

Recommended Mitigation: Implement a check which errors when user tries to cast vote affter official `CommunityState.header.voting_end_time` time prior to casting vote & making call to `finalize voting`

Deriverse Fixed in commit: [a5194d]9https://github.com/deriverse/protocol-v1/commit/a5194d26218f0828e83481ebb2a6f7071

Cyfrin: Verified.

7.2.7 Missing Slippage Protection in Market Seat Buy/Sell Operations

Description: The `buy_market_seat()` and `sell_market_seat()` functions calculate seat prices dynamically based on the current `perp_clients_count` at execution time, but provide no slippage protection. Users cannot specify maximum/minimum acceptable prices, exposing them to unexpected price changes.

In `buy_market_seat()`, the seat price is calculated as:

```
let seat_price = PerpEngine::get_place_buy_price(
    instrument.perp_clients_count,
    instrument.crncy_token_decs_count,
)?;

instrument.seats_reserve += seat_price;
let price = data.amount + seat_price;
// ... price is deducted without validation
client_state.sub_crncy_tokens(price)?;
```

Similarly, in `sell_market_seat()`, the sell price is calculated:

```
let seat_price = PerpEngine::get_place_sell_price(
    instrument.perp_clients_count,
    instrument.crncy_token_decs_count,
)?;

client_state.add_crncy_tokens(seat_price)?;
```

The price calculation functions (`get_place_buy_price()` and `get_place_sell_price()`) use a bonding curve model where the price increases with each additional seat. The price is calculated based on:

```
pub fn get_place_buy_price(supply: u32, dec_factor: u32) -> Result<i64, DeriverseError> {
    let df = get_dec_factor(dec_factor);
    Ok(get_reserve(supply + 1, df)? - get_reserve(supply, df)?)
}

pub fn get_place_sell_price(supply: u32, dec_factor: u32) -> Result<i64, DeriverseError> {
    let df = get_dec_factor(dec_factor);
    Ok(get_reserve(supply, df)? - get_reserve(supply - 1, df)?)
}
```

The problem:

1. Between transaction submission and execution, other legitimate market transactions can change `perp_clients_count`, causing the actual execution price to differ from what the user expected
2. Users have no way to specify a maximum acceptable price for buying or minimum acceptable price for selling

3. The price is calculated and immediately applied without any validation against user expectations
4. During periods of high market activity, concurrent seat purchases/sales can cause significant price drift

Impact:

- **Unpredictable Execution:** Users have no guarantee that their transaction will execute at an acceptable price, even in normal market conditions
- **Poor User Experience:** Users cannot protect themselves from unfavorable price movements caused by legitimate concurrent market activity

Recommended Mitigation: Add slippage protection by allowing users to specify maximum/minimum acceptable prices in the instruction data, and validate the calculated price against these limits before execution.

Deriverse: Fixed in commit a8181f3.

Cyfrin: Verified

7.2.8 Missing Quorum Requirement in Governance Voting

Description: The governance voting system in `finalize_voting()` lacks a quorum requirement, allowing protocol parameters to be changed based on relative vote counts without validating that a minimum percentage of total token supply has participated. This enables a small minority of token holders to control protocol governance decisions, undermining the decentralized nature of the system.

```
let decr = community_account_header.voting_decr;
let incr = community_account_header.voting_incr;
let unchange = community_account_header.voting_unchange;
let tag = community_account_header.voting_counter % 6;

if decr > unchange && decr > incr {
    // Apply DECREMENT - no quorum check
    match tag {
        0 => { community_account_header.spot_fee_rate -= 1; }
        // ... other parameters
    }
} else if incr > unchange && incr > decr {
    // Apply INCREMENT - no quorum check
    match tag {
        0 => { community_account_header.spot_fee_rate += 1; }
        // ... other parameters
    }
}
```

While `voting_supply` is tracked and set to `drvs_tokens` (total supply), it is never used to validate that sufficient tokens participated in the vote:

```
community_account_header.voting_supply = community_account_header.drvs_tokens;
```

The Problem:

1. No minimum participation threshold (quorum) is checked before applying voting results
2. A single voter with a small amount of tokens can determine protocol changes if no one else votes
3. The total voting power (`decr + incr + unchange`) is never compared against `voting_supply` or any minimum threshold
4. This violates common governance best practices where significant decisions require meaningful community participation

Impact:

- **Governance Attack Vector:** Malicious actors can wait for low-activity periods to push through unfavorable parameter changes
- **Undermined Decentralization:** The voting system fails to ensure decisions represent a meaningful portion of the community

Recommended Mitigation: Add a quorum requirement that validates minimum participation before applying voting results. The quorum should be a percentage of the total voting supply.

```
        let total_votes = decr + incr + unchange;
        let voting_supply = community_account_header.voting_supply;

        // Add quorum check (e.g., require at least 5% participation)
        const MIN_QUORUM_PERCENTAGE: i64 = 5; // 5% of voting supply
        let min_quorum = (voting_supply * MIN_QUORUM_PERCENTAGE) / 100;
```

deriverse: Fixed in commit b4e1045.

Cyfrin: Verified.

7.2.9 Inconsistent Token Account Address Requirements Between Deposit and Withdraw

Description: The `deposit()` and `withdraw()` functions have inconsistent requirements regarding token account addresses:

- `deposit()`: Allows deposits from any token account owned by the signer, not requiring an Associated Token Account (ATA)
- `withdraw()`: Requires withdrawals to go to the signer's ATA address only

Deposit Function - No ATA Requirement:

```
// deposit.rs lines 145-154
let old_token = check_spl_token(
    client_associated_token_acc,
    program_token_acc,
    token_state,
    token_program_id,
    mint,
    signer.key,  // Only verifies owner is signer
    &pda,
    data.token_id,
)?;
```

The `check_spl_token()` function (in `helper.rs`) only validates:

- Token account owner is the signer (line 207)
- Mint address matches
- Token program matches

```
        if client_token != *mint.key {
            bail!(InvalidMintAccount {
                token_id: token_state.id,
                expected_address: *mint.key,
                actual_address: client_token,
            });
        }

        let client_token_owner = client_token_acc.data.borrow()[32..].as_ptr() as *const Pubkey;

        if *client_token_owner != *signer {
            bail!(DeriverseErrorKind::InvalidTokenOwner {
                token_id: token_state.id,
```

```
                address: *client_token_acc.key,
                expected_adderss: *signer,
                actual_address: *client_token_owner,
            });
        }
```

It does NOT verify the address is an ATA, meaning users can deposit from any token account they control.

Withdraw Function - ATA Required:

```
// withdraw.rs lines 127-138
let expected_address = get_associated_token_address_with_program_id(
    signer.key,
    mint_acc.key,
    token_program_id.key,
);
if expected_address != *client_associated_token_acc.key {
    bail!(InvalidAssociatedTokenAddress {
        token_id: token_state.id,
        expected_address: expected_address,
        actual_address: *client_associated_token_acc.key,
    });
}
```

The `withdraw()` function explicitly requires the destination to be the signer's ATA address, rejecting any other token account address.

This works fine with Token-2022, as the owner of a token account is immutable. However, in the legacy token program, a user can transfer ownership of any token account. If a user who deposited funds no longer owns the associated token account, they will not be able to withdraw their funds using the withdraw function since we verify that the ATA owner must be the signer.

Here: https://github.com/solana-program/token/blob/main/program/src/processor.rs#L441

Impact: It can cause a loss of user funds. Here is the scenario:

1. The user had an ATA and transferred ownership of that ATA.
2. The user deposited using a different token account.
3. Since the user no longer owns the original ATA, and our logic checks that the ATA owner must match the user, any withdrawal attempt will fail. As a result, the user is unable to withdraw their funds.

Recommended Mitigation: Either:

- Make Both Functions Consistent (Require ATA/Allow Any Token Account)
- If the current asymmetry is intentional, clearly document why deposits allow flexibility and withdrawals require ATA.

Deriverse: Fixed in commit 1e3d88.

Cyfrin: Verified.

7.2.10 Silent Error Handling in `clean_generic` Introduces Multiple Risks

Description: The `clean_generic()` function in `ClientPrimaryState` implements silent error handling for account parsing using `if let Ok(header)`, which introduces three potential issues that can lead to incorrect state updates, account desynchronization, and masking of underlying instruction problems.

The function expects accounts to be provided in pairs: a `maps_acc` account followed by a `client_infos_acc`. The loop counter increments by 2 for each successfully processed pair, assuming 2 accounts are consumed per iteration.

```
            if let Ok(header) = header { // Silent failure handling
                counter += 2;
                let client_infos_acc = next_account_info!(accounts_iter)?;

                SpotTradeAccountHeader::<SPOT_CLIENT_INFOS>::validate(
                    client_infos_acc,
                    program_id,
                    Some(header.instr_id),
                    version,
                )?;

                self.client_infos_acc = client_infos_acc;
                self.maps_acc = maps_acc;
                self.resolve_instr(client_infos_acc, false)?;
                self.finalize_spot()?;
            }
        }
        Ok(())
    }
```

- Issue 1: Silent Error Handling Masks Instruction-Level Problems

A single parsing failure may indicate that the entire instruction is malformed or that the account structure is fundamentally incorrect. By silently continuing execution, the function masks these critical errors and may introduce additional risks

- Issue 2: Incorrect Assumption About Account Pairing Structure

The silent error handling assumes a specific account ordering pattern: `[bad_maps_acc, good_maps_acc, good_client_infos_acc]`. However, this assumption is fragile and often incorrect. The actual account sequence may not follow this pattern. For example, if a `maps_acc` fails, the next account might be another `maps_acc` (as assumed), but it could also be a `client_infos_acc` from a previous pair

- Issue 3: Iterator-Counter Desynchronization

When `maps_acc` fails to parse, `next_account_info!(accounts_iter)?` has already advanced the iterator, consuming one account, but the counter does not increment when parsing fails. This creates a desynchronization:

- **Iterator position**: Advances by 1 account (`maps_acc consumed`)
- **Counter value**: Remains unchanged (no increment)
- **Expected behavior**: Counter should track consumed accounts

Given 6 accounts: `[map1(bad), map2(ok), client_info2(ok), map3(bad), map3(ok), client_info3(ok)]` and `length = 6`:

1. Iteration 1: Reads `map1(bad)` → parsing fails → counter = 1, iterator at position 1
2. Iteration 2: Reads `map2(ok)` → parsing succeeds → counter = 3, iterator at position 3 (after reading `client_info2`)
3. Iteration 3: Reads `map3(bad)` → parsing fails → counter = 3, iterator at position 4
4. Iteration 4: Reads `map3(ok)` → parsing succeeds → counter = 5, iterator at position 6 (after reading `client_info3`)
5. Loop condition `counter < length` (5 < 6) is still true, but iterator is exhausted

This desynchronization can cause:

- Reading accounts beyond the intended range
- Processing incorrect account pairs

- Potential panic if `next_account_info!` is called when the iterator is exhausted

Impact:

1. **Instruction Integrity Violation**: Silent error handling masks critical instruction-level problems, allowing malformed or malicious instructions to partially execute.
2. **Incorrect Account Pairing**: The assumption that failed accounts are followed by correct pairs is often violated.
3. **Iterator-Counter Desynchronization**: The iterator position and counter become misaligned when parsing fails, causing:
   - Subsequent iterations to read incorrect accounts
   - Processing accounts out of order or skipping required accounts
   - Potential panic when `next_account_info!` is called on an exhausted iterator

Recommended Mitigation: The function should fail fast on parsing errors rather than silently continuing. This addresses all three issues.

Deriverse: Fixed in commit 1c2f2a5.

Cyfrin: Verified.

7.2.11 `ClientCommunityState::update` function does not update the `rate` before calculating new `dividends_value`

Description: The `ClientCommunityState::update` function is responsible for distributing dividends based on the amount of DRVS tokens a user holds:

```
for (i, d) in self.data.iter_mut().enumerate() {
    let amount = (((community_state.base_crncy[i].rate - d.dividends_rate)
        * self.header.drvs_tokens as f64) as i64)
        .max(0);
    d.dividends_value += amount;
}
```

The value of `community_state.base_crncy[i].rate` is updated every hour through the dividend allocation function.

However, there is a possibility that more than one hour has passed since the last allocation and dividend allocation function is not called and our current implementation does not update the `rate` in such cases. As a result, the calculation may rely on stale `rate`, and `dividends_value` may be computed using a stale `community_state.base_crncy[i].rate`.

Scenario:

1. One hour has just passed, but the dividend allocation function is not called.
2. The user calls withdraw to withdraw DRVS tokens, which also does not update the rates.
3. As a result, the user receives a lower dividend value because the calculation is based on an outdated rate.

The scenario described above involves the withdraw functionality, but a similar issue can occur with deposit as well:

1. One hour has just passed, but the dividend allocation function is not called.
2. A user purchases DRVS tokens and calls deposit. However, we record the old rate as the user's `dividends_rate`.
3. The user can then call the dividend allocation function and receive dividends that they should not be entitled to, because they did not hold those DRVS tokens during the previous hour.

Impact: If the dividend allocation function is not called before update, and more than one hour has passed, it can result in some users unfairly incurring losses while others receive undeserved profits.

Recommended Mitigation: Update `community_state.base_crncy[i].rate` if one hour has passed before updating `dividends_value` and `dividends_rate`.

Deriverse: Fixed in commit ca593e.

Cyfrin: Verified.

7.2.12 Lack of slippage in `spot_lp` liquidity operations due to relying on the changing `header.crncy_tokens` and `header.asset_tokens`

Description: The `spot_lp` function lacks slippage protection when adding or removing liquidity.

In most cases, the function calculates the required asset and currency tokens based on the current pool state at execution time, **without allowing users to specify minimum output amounts or maximum acceptable slippage**. This exposes users to unfavorable execution prices due to pool state changes between transaction submission and execution.

In `src/program/processor/spot_lp.rs`, when adding liquidity (lines 189-194), the function directly calculates the required tokens based on the current pool state:

```
trade_crncy_tokens = ((instr_state.header.crncy_tokens + instr_state.header.pool_fees)
    as f64
    * amount as f64
    / instr_state.header.ps as f64) as i64;
trade_asset_tokens = (instr_state.header.asset_tokens as f64 * amount as f64
    / instr_state.header.ps as f64) as i64;
```

The pool's asset_tokens and crncy_tokens are modified during normal trading operations (as seen in `engine.rs` via `change_tokens` and `change_mints`).

```
                self.change_tokens(traded_qty, side)?;
                self.change_mints(traded_mints, side)?;
                self.log_amm(traded_qty, traded_mints, side);
```

However, the spot_lp function:

- Does not allow users to specify minimum acceptable output amounts (similar to Uniswap V2's `amountAMin` and `amountBMin`)
- Does not validate that the execution price is within an acceptable range

Impact: Users can suffer unexpected losses when adding/removing liquidity due to pool state changes

Recommended Mitigation: Add slippage protection parameters to SpotLpData structure and Implement validation checks after calculating `trade_asset_tokens` and `trade_crncy_tokens`

Deriverse: Fixed in commit 337383.

Cyfrin: Verified.

7.2.13 Referral Incentives Disabled for All Legitimate Users During Any Liquidation

Description: Once the engine detects any instrument that requires liquidation (`is_long_margin_call` or `is_short_margin_call`), the `margin_call` flag is set to true for every subsequent `new_perp_order`. This flag is passed unchanged into `match_{ask,bid}_orders`, which disables referral payouts while it is `true`. As a result, all users— even those submitting normal orders unrelated to the liquidation — stop receiving/producing referral rewards for as long as any liquidation candidate remains. This global switch was likely intended only for actual liquidation trades.

In `new_perp_order.rs` the code sets `margin_call = engine.is_long_margin_call() || engine.is_short_margin_call();`

```
let margin_call = engine.is_long_margin_call() || engine.is_short_margin_call();
if !margin_call {
    engine.state.header.perp_spot_price_for_withdrowal = engine.state.header.perp_underlying_px;
}
```

That boolean is forwarded to `PerpEngine::match_{ask,bid}_orders` via `MatchOrdersStaticArgs`

```
        if engine.cross(price, OrderSide::Ask) {
            (remaining_qty, _, ref_payment) = engine.match_ask_orders(
                Some(&mut client_community_state),
                &MatchOrdersStaticArgs {
                    price,
                    qty: data.amount,
                    ref_discount,
                    ref_ratio: header.ref_program_ratio,
                    ref_expiration: header.ref_program_expiration,
                    ref_client_id: header.ref_client_id,
                    trades_limit: 0,
                    margin_call,
                    client_id: client_state.temp_client_id,
                },
            )?;
        }
```

Referral rebates are conditioned on `!args.margin_call in perp_engine.rs`: when `margin_call` is `true`, ref_payment is forced to zero.

```
        let ref_payment = if self.time < args.ref_expiration && !args.margin_call {
            ((fees - rebates) as f64 * args.ref_ratio) as i64
        } else {
            0
        };
```

Liquidation routines (`check_long_margin_call, check_short_margin_call`) also pass `margin_call: true` explicitly, but there is no distinction between liquidation-triggered fills and ordinary orders.

```
    if buy {
        if engine.check_short_margin_call()? < MAX_MARGIN_CALL_TRADES {
            engine.check_long_margin_call()?;
        }
    } else if engine.check_long_margin_call()? < MAX_MARGIN_CALL_TRADES {
        engine.check_short_margin_call()?;
    }
```

Therefore, the presence of any liquidation candidate globally blocks referral rewards for all traders, regardless of who is being liquidated.

Impact: Legitimate users lose their expected referral incentives whenever any other account is under liquidation. Although not an immediate loss of funds, it represents a systemic incentive failure affecting all participants during stressed periods.

Recommended Mitigation: Restrict the `margin_call` flag to trades that are actually part of liquidation flows.

Deriverse: Fixed in commit bc9bd6.

Cyfrin: Verified.

7.2.14 Slippage Guard Could be Too Loose For Leveraged Perp Markets

Description: Perpetual "market" orders fall back to the same hard-coded `±12.5%` price cap that is used for spot orders. While that cap is arguably acceptable for spot, it is dangerously loose for leveraged perp trading: a user

can be filled up to 12.5 % away from the reference price in one shot, magnifying losses by the user's leverage factor. There is no ability to tighten this slippage or enforce a stricter cap when leverage is high.

In `new_perp_order.rs`, non-limit buys default to `px + (px >> 3)` and sells to `px - (px >> 3)`, i.e. ±12.5 % of the current underlying price.

```
    let (price, min_tokens) = PerpParams::get_settings(
        if data.order_type == OrderType::Limit as u8 {
            data.price
        } else if buy {
            px + (px >> 3)
        } else {
            px - (px >> 3)
        },
        data.amount,
        if buy { OrderSide::Bid } else { OrderSide::Ask },
        px,
        data.ioc,
        engine.dc,
    )?;
```

Spot orders reuse the same ±12.5 % window, which is acceptable because spot positions are unleveraged.

Perp orders, however, can be levered up to the protocol maximum (`MAX_PERP_LEVERAGE`). Filling a leveraged market order at `12.5% (or +12.5%)` immediately consumes a large portion of the user's margin and can nearly trigger unintended liquidation during extreme market conditions, even when the user was intended to trade near the mark price.

Also, users have no way to configure a tighter cap unless they avoid market orders altogether (use `limit+IOC`), which is unrealistic—many traders still expect market orders to have reasonable slippage protection.

Impact: Under volatile or thin-liquidity conditions, leveraged traders who rely on market orders can be executed at very unfavorable prices (up to `12.5%` away), leading to outsized losses or incoming instant liquidations.

Recommended Mitigation: Ultimately the team should decide how strict to be, but the current 12.5 % blanket cap is out of line with leveraged-market risk management and should be tightened.

If possible, implement leverage-aware slippage caps (e.g., shrink tolerance as leverage increases) or allow users to specify a custom slippage limit, with a protocol-defined maximum.

Another recommendation is to benchmark major CEX/DEX perp products to choose a safer default.

Deriverse: Fixed in commit b2ff47aa.

Cyfrin: Verified.

7.2.15 Referral discount is not applied when `fees_prepayment` is zero in `PerpEngine::fill`

Description: In `PerpEngine::fill`, when `fees_prepayment` is zero we do not apply the referral discount even though we still send `ref_payment` to `ref_address`.

```
                if *fees_prepayment == 0 {
                    let fee = (traded_crncy as f64 * self.fee_rate) as i64;
                    taker_info.sub_funds(fee).map_err(|err| drv_err!(err))?;
                    fee
                } else {
```

`self.fee_rate` is equal to `perp_fee_rate`.

Impact: The user is required to pay additional funds due to the referral discount not being applied, causing a loss to the user.

Recommended Mitigation: Instead of charging `self.fee_rate`, we should charge the fee rate after applying the referral discount, `(1.0 - args.ref_discount) * self.fee_rate`.

Deriverse: Fixed in commit 91bffc.

Cyfrin: Verified.

7.2.16 Operator precedence Issue during points program expiration validation in `change_points_program_expiration`

Description: In `change_points_program_expiration`, the validation for `new_expiration_time` uses `!data.new_expiration_time < root_state.points_program_expiration` instead of `!(data.new_expiration_time < root_state.points_program_expiration)`, causes unintended behavior.

```
if !data.new_expiration_time < root_state.points_program_expiration {
    bail!(InvalidNewExpirationTime {
        program_name: "Points Program".to_string(),
        new_time: data.new_expiration_time,
        old_time: root_state.points_program_expiration
    });
}
```

This occurs because the bitwise NOT operator has higher precedence than the less-than comparison operator(<). This causes the expression to be evaluated as:

```
(!data.new_expiration_time) < root_state.points_program_expiration
```

Instead of the intended:

```
!(data.new_expiration_time < root_state.points_program_expiration)
```

Impact: Using the `change_points_program_expiration` function, we cannot decrease the points program expiration because applying `!` to a `u32` performs a bitwise NOT operation. This produces a value close to `u32::MAX`, which is almost always greater than any reasonable expiration time when attempting to reduce `points_program_expiration`.

Recommended Mitigation: Use `!(data.new_expiration_time < root_state.points_program_expiration)` during validation, here's the fix:

```
    if !(data.new_expiration_time < root_state.points_program_expiration){
        bail!(InvalidNewExpirationTime {
            program_name: "Points Program".to_string(),
            new_time: data.new_expiration_time,
            old_time: root_state.points_program_expiration
        });
    }
```

Deriverse: Fixed in commit eae149.

Cyfrin: Verified.

7.2.17 Attacker can extract value by buying and selling the seat

Description: Whenever a user buys a seat, they must pay `seat_price` in the currency token. The `seat_price` depends on the `perp_clients_count` at the time of purchase. If the `perp_clients_count` is higher, the user will pay more for the seat; if it is lower, the seat will cost less.

```
    pub fn get_place_buy_price(supply: u32, dec_factor: u32) -> Result<i64, DeriverseError> {
        let df = get_dec_factor(dec_factor);
        Ok(get_reserve(supply + 1, df)? - get_reserve(supply, df)?)
    }
```

The attacker can exploit this behavior and extract profit by creating a scenario where the user ends up paying a much higher seat price. For example, assuming a currency X with 6 decimals:

1. Initially, the `perp_clients_count` is 199,999.
2. The attacker buys 1,000 seats to profit later.
3. During user transaction, the `perp_clients_count` is 200,999, and the user pays a seat price of 26,030,289.
4. The attacker then sells the 1,000 seats and extracts a profit of 1,030,788.
5. If the attacker had not performed this attack, the user would have only needed to pay 24,999,501 for their seat.

No front-running is required. An attacker can pre-purchase seats and later sell them to extract the funds without using the instrument.

Impact: The attacker can extract user funds through this behavior, causing the user to pay more than expected.

Recommended Mitigation: Recommendation is to store the seat price the user originally paid and later when the user sells the seat return the exact same amount they paid at the time of purchase.

Deriverse: Fixed in commit a80b0e.

Cyfrin: Verified.

7.2.18 Fee Prepayment Locked Due to Asset Record Cleanup after `fees_deposit`

Description: When a user deposits fee prepayment via `fees_deposit`, if the deposit reduces crncy_tokens to zero(right after or later afterwards), the asset record is cleared by `clean_and_check_token_records()`. Later, when the user attempts to withdraw via `fees_withdraw`, the `resolve()` call with `alloc=false` fails to find the asset record, preventing withdrawal of the prepaid fees even though the prepayment amount is still stored in `client_community_state`.

The issue occurs in the following sequence(below is an example):

- In `fees_deposit`: `client_state.sub_crncy_tokens(data.amount)` reduces the currency token balance

```
client_community_state.data[crncy_index].fees_prepayment += data.amount;
client_state.sub_crncy_tokens(data.amount)?;
```

- In `fees_deposit`: `client_state.clean_and_check_token_records()` is called, which clears asset records when `value == 0`:

```
client_state.clean_and_check_token_records()?;
```

```
pub fn clean_and_check_token_records(&mut self) -> DeriverseResult {
    for r in self.assets.iter_mut() {
        if r.asset_id != 0
            && (r.asset_id & 0xFF000000) != AssetType::SpotOrders as u32
            && (r.asset_id & 0xFF000000) != AssetType::Perp as u32
        {
            match r.value.cmp(&0) {
                std::cmp::Ordering::Equal => r.asset_id = 0,
                std::cmp::Ordering::Less => bail!(InsufficientFunds),
                std::cmp::Ordering::Greater => {}
            }
        }
    }
    Ok(())
}
```

- In `fees_withdraw`: `client_state.resolve(AssetType::Token, data.token_id, TokenType::Crncy, false)` is called with `alloc=false`

```
client_community_state.update(&mut client_state, &mut community_state, None)?;
client_state.resolve(AssetType::Token, data.token_id, TokenType::Crncy, false)?;
let dec_factor = get_dec_factor(community_state.base_crncy[crncy_index].decs_count) as f64;
```

```
let amount = (data.amount as f64 / dec_factor) as i64;
```

- In `find_or_alloc_asset`: When `alloc=false` and the asset record is not found, it returns `AssetNotFound` error:

```
        if asset_index == NULL_INDEX {
            realloc = true;
            if !alloc {
                bail!(AssetNotFound {
                    asset_type: asset,
                    id,
                });
            }
```

- The withdrawal fails, even though `fees_prepayment` is still stored in `client_community_state.data[crncy_index].fees_prepayment`

The root cause is that `fees_withdraw` requires `resolve()` to succeed to set the `crncy_tokens` pointer before calling `add_crncy_tokens()`. However, `resolve()` with `alloc=false` cannot recreate the asset record that was cleared after deposit.

Impact: Users who deposit fee prepayment that reduces their crncy_tokens to zero(or maybe afterwards) will be unable to withdraw their prepaid fees. They need to deposit again in order to create the `asset`. This requires manual addtional manual operation.

Recommended Mitigation: Change `fees_withdraw` to use `alloc=true` when resolving the currency token asset:

```
client_state.resolve(AssetType::Token, data.token_id, TokenType::Crncy, true)?;
```

Deriverse: Fixed in commit 6662b16.

Cyfrin: Verified.

7.2.19 Decimal Mismatch in Fee Prepayment Accounting Causes Incorrect Balance Tracking

Description: There is a critical decimal mismatch between how fee prepayment is stored and withdrawn. In `fees_deposit`, the `fees_prepayment` field stores the raw `data.amount` value (in token's native decimal units), but in `fees_withdraw`, it subtracts `data.amount / dec_factor` (in human-readable units).

This causes severe accounting errors where the stored prepayment balance becomes incorrect after withdrawals, and also leads to incorrect off-chain event logging.

The issue occurs due to inconsistent unit handling:

In `fee_deposit`:

```
let prepayment = data.amount as f64 / dec_factor;
...
client_community_state.data[crncy_index].fees_prepayment += data.amount;  // Stores raw value
...
client_state.sub_crncy_tokens(data.amount)?;
```

However, in `fee_withdraw`:

```
let amount = (data.amount as f64 / dec_factor) as i64;  // Divides by dec_factor
client_community_state.data[crncy_index].fees_prepayment -= amount;  // Subtracts divided value
```

Example with 6 decimals (dec_factor = 1,000,000):

- User deposits 1 token: `fees_prepayment += 1,000,000 (stored as 1,000,000)`
- User withdraws 1 token: `amount = 1,000,000 / 1,000,000 = 1, fees_prepayment -= 1`
- Result: `fees_prepayment = 999,999` instead of `0`

Additional Issues:

- Event logging mismatch: The log events record `data.amount` (raw value), but the actual withdrawal amount is `data.amount / dec_factor`, causing incorrect off-chain accounting

```
solana_program::log::sol_log_data(&[bytemuck::bytes_of::<FeesWithdrawReport>(
    &FeesWithdrawReport {
        tag: log_type::FEES_WITHDRAW,
        client_id: client_state.id,
        token_id: data.token_id,
        amount: data.amount,
        time: clock.unix_timestamp as u32,
        ..FeesWithdrawReport::zeroed()
    },
)]);
```

Workaround: The current implementation has a workaround to multiply the original `data.amount` by `dec_factor`, but this must be validated against `SPOT_MAX_AMOUNT` limits.

Impact:

- Off-chain accounting errors: Event logs show incorrect amounts, causing off-chain systems to track wrong values.
- Original Withdrawal Don't Work: Original `fees_withdraw` doesn't work correctly unless a workaround is being applied.

Recommended Mitigation: Use `data.amount`consistently:

```
client_community_state.data[crncy_index].fees_prepayment -= data.amount;  // Use raw value
...
client_state.add_crncy_tokens(data.amount)?;  // Use raw value
```

Deriverse: Fixed in commit 1dcab9d.

Cyfrin: Verified.

7.2.20 Users can sell their market seat without paying loss coverage

Description: When users incur loss, its covered from available insurance funds and is stored and tracked inside `taker_info4.loss_coverage`, this is what user owes to protocol before he closes his position, because insurance funds have been used to cover up for these losses incurred by user. Inside `sell-market-seat` we are calling close_account, it does not check for users's loss coverage, it only checks

```
        if (*self.perp_info3).bids_entry != NULL_ORDER
            || (*self.perp_info3).asks_entry != NULL_ORDER
            || self.current_instr_index >= self.assets.len()
        {
            bail!(ClientDataDestruction);
        }
```

if user has incurred losses in past and insurance funds covered this loss, he must pay back these funds before closing the seat, the function `try_to_close_perp` checks this properly, if user has pending loss coverage we subtract it from users' funds, this way he pays back his losses and then closes the position.

```
        if (*self.perp_info4).loss_coverage > 0 {
            let delta = (*self.perp_info4)
                .loss_coverage
                .min((*self.perp_info).funds);
            if delta > 0 {
                engine.state.header.perp_insurance_fund += delta;
                (*self.perp_info4).loss_coverage -= delta;
                (*self.perp_info).funds -= delta;
```

```
            }
        }
```

Impact: Users can close their seats without paying for their incured losses

Recommended Mitigation: Call `try_to_close_perp` instead of `close_perp` inside `sell-market-seat`.

Deriverse: Fixed in commit: e5af702

Cyfrin: Verified.

7.2.21 Users are getting back their `soc-loss-funds` while selling their market seat

Description: Inside `sell-market-seat` when soc-loss-funds are negative, which means user has added and contributed to protocol's soc loss funds & beared losses on behalf of others of his side... in this case user is getting back those funds, and these funds are being deducted from insurance funds as well, general idea is to not give back what user has contributed towards soc loss, but here, user is getting it back

```
    let collactable_losses = info.funds().min(client_state.perp_info4()?.soc_loss_funds);
    engine.state.header.perp_insurance_fund += collactable_losses;

    client_state.add_crncy_tokens((info.funds() - collactable_losses).max(0))?;
```

for eg.

- A user incurred some losses, his soc loss are updated to +20
- in next call to check-soc-loss he pays back +30, user's net soc loss becomes -10, meaning user contributed towards protocol's soc losses as we can see this amount being added to `self.state.header.perp_soc_loss_funds` as well
- now user goes to sell his market seat with his `soc-loss-funds` set to -10..... he gets back this amount, which he should not.

Impact: Users get back what they paid towards soc loss.

Recommended Mitigation: In the cases when user's `soc-loss-funds` come out to be negative, avoid accounting for it.

Deriverse: Fixed in commit: e5af702

Cyfrin: Verified.

7.2.22 Users can provide old price feeds to trade in their favor

Description: The function set_underlying_px is being called from many places in code, its setting the underlying price from user provided oracle feed if oracle is set. `set-underlying-px` relays on user provided feed but never verifies the price publish time or confidence interval. A price that is hours old (or intentionally frozen) is accepted as fresh. It doesnt check whether the oracle data is fresh, whether the oracle slot/timestamp is recent, whether the oracle confidence is good, the only check in place is feed account address matches expected.

```
            let feed_id = next_account_info!(accounts_iter)?;
            if self.header.feed_id == *feed_id.key {
                let oracle_px =
                    i64::from_le_bytes(feed_id.data.borrow()[73..81].try_into().unwrap());
                let oracle_exponent =
                    9 + i32::from_le_bytes(feed_id.data.borrow()[89..93].try_into().unwrap());
                let mut dc: i64 = 1;
```

Even if the account is the correct feed ID, the data inside may be old, because we currently do not check any of the feed's fields like Pyth format has timestamp, slot, conf, etc., which tells us how fresh the feed is.

Impact: Trade may be computed from obsolete data, letting attackers over or under-pay.

Recommended Mitigation: Check the feed's timestamp to ensure its recent, we can allow upto certain minutes old prices and if the interval if more than allowed, dont accept it.

Deriverse We exclude oracle support.

Cyfrin: Verified.

7.2.23 Users incur losses when selling seats

Description: Whenever a user buys or sells a seat, we calculate the seat price at that moment using the difference between `get_reserve` from the new/current position and the old position.

```
    pub fn get_place_buy_price(supply: u32, dec_factor: u32) -> Result<i64, DeriverseError> {
        let df = get_dec_factor(dec_factor);
        Ok(get_reserve(supply + 1, df)? - get_reserve(supply, df)?)
    }

    pub fn get_place_sell_price(supply: u32, dec_factor: u32) -> Result<i64, DeriverseError> {
        let df = get_dec_factor(dec_factor);
        Ok(get_reserve(supply, df)? - get_reserve(supply - 1, df)?)
    }
```

The `get_reserve` function returns an i64, whose maximum value is 9223372036854775807. and, in `get_reserve` calculation is done in f64 and if the result in f64 exceeds this limit, casting it back to i64 will clamp the value to the maximum i64 value (9223372036854775807).

Example: The seat price from 249,995 to 250,000, the value becomes zero when `crncy_token_decs_count` is 9.

Impact: This leads to unexpected behavior, which can result in losses for regular users and create opportunities for an attacker to extract value if `perp_clients_count` is close 250000.

Recommended Mitigation: `get_reserve` should return f64. After computing the difference between the results of both `get_reserve` calls, we can then convert the final value back to i64 for more accurate result.

Deriverse: Fixed in commit c8d26d.

Cyfrin: Verified.

7.2.24 `perp_statistics_reset` can be used by users to skip `collactable-losses` while selling market seat

Description: Inside `perp-statistics-reset` we set soc loss funds to zero without checking if user incurred losses.

```
//inside perp-statistics-reset
    client_state.perp_info4()?.soc_loss_funds = 0;
```

if user would have incurred losses, these funds would be positive which means, this is the amount of funds protocol should collect and add to insurance funds at the time when user sells the market seat. we set those to 0 and later call `check-soc-loss` which in turn would make this funds -ve. If now user calls `sell-market-seat` he gets back this funds and insurance funds are also reduced by this amount which is incorrect.

```
//inside sell-market-seat
    let collactable_losses = info.funds().min(client_state.perp_info4()?.soc_loss_funds);
    engine.state.header.perp_insurance_fund += collactable_losses;

    client_state.add_crncy_tokens((info.funds() - collactable_losses).max(0))?;
```

if user has incurred losses earlier & his `soc-loss-funds` is positive, he should not be allowed to use `perp-statistics-reset`, he can simply use it to wipe "the amount he owns to protocol on exiting the position".

Impact: Users can missuse this to clean their soc loss records which store how much they owe to protocol.

Recommended Mitigation: Get user's soc losses, if they are positive, which means user owes the amount to protocol, avoid statistics resetting and revert.

Deriverse: Fixed in commit: e5af702

Cyfrin: Verified.

7.2.25 Missing Array Synchronization in `dividends_claim` Prevents Users from Claiming Dividends for Newly Added Base Currencies

Description: The `dividends_claim` instruction does not synchronize `client_community_state.data` with `community_state.base_crncy` before processing dividends. If a new base currency is added to the community and a user hasn't performed any operations (like `deposit` or `trading`) that trigger synchronization, the user's `client_community_state.data` array will be shorter than community_state.base_crncy. This causes the zip iterator to stop early, preventing users from claiming dividends for newly added base currencies until they perform another operation that triggers synchronization.

In `src/state/client_community.rs`, the update function synchronizes arrays before processing:

```
pub fn update(
    &mut self,
    client_primary_state: &mut ClientPrimaryState<'a, 'info>,
    community_state: &mut CommunityState,
    payer: Option<&'a AccountInfo<'info>>,
) -> DeriverseResult {
    self.update_records(community_state, payer)?;  // Synchronizes arrays
    // ... rest of the logic
}
```

The update_records function ensures `client_community_state.data` matches `community_state.base_crncy`:

```
fn update_records(
    &mut self,
    community_state: &CommunityState,
    payer: Option<&'a AccountInfo<'info>>,
) -> DeriverseResult {
    // ... reallocates if needed ...
    self.header.count = community_state.header.count;
    // ... creates new data array ...
    for (d, b) in self.data.iter_mut().zip(community_state.base_crncy.iter()) {
        d.crncy_token_id = b.crncy_token_id;
        d.fees_ratio = 1.0;
    }
}
```

However, in `src/program/processor/dividends_claim.rs`, the instruction directly iterates without synchronization:

```
for (d, b) in client_community_state
    .data
    .iter_mut()
    .zip(community_state.base_crncy.iter_mut())
{
    // Process dividends...
}
```

- Other instructions like `deposit` call `client_community_state.update()`, which triggers `update_records`
- `dividends_claim` does not call `update_records` or update before processing
- If `client_community_state.data.len() < community_state.base_crncy.len()`, the zip iterator stops when the shorter array ends

- Users cannot claim dividends for newly added base currencies until they perform another operation

Impact: Users who haven't performed operations after new base currencies are added cannot claim dividends for those currencies. Users must perform an additional operation (e.g., deposit) to trigger synchronization before claiming dividends

Recommended Mitigation: Add synchronization before processing dividends in dividends_claim

Deriverse: Fixed in commit 5f5460.

Cyfrin: Verified.

7.2.26 `min_qty` Bypass via `IOC` limit order

Description: Spot orders compute the minimum order quantity `min_qty` from the user-supplied limit price. An attacker can set an arbitrarily large limit price to drive `min_qty` down to a tiny value while the actual execution price is still clamped near the mark price. This bypasses the intended minimum order size enforcement for IOC orders.

`SpotParams::get_settings` enforces a ±12.5 % clamp on IOC prices via the `ipx` variable, but the minimum tokens are calculated from the original `px` argument (the user provided price) even when `IOC` mode is active:

```
        if ioc != 0 {
            if px < last_px - max_diff {
                ipx = last_px - max_diff;
            } else if px > last_px + max_diff {
                ipx = last_px + max_diff;
            } else {
                ipx = px;
            }
        } else if px < last_px - max_diff || px > last_px + max_diff {
            bail!(DeriverseErrorKind::InvalidPrice {
                price: px,
                min_price: last_px - max_diff,
                max_price: last_px + max_diff,
            });
        } else {
            ipx = px;
        }
...
        let min_tokens = min_qty(px, dc);
        // TODO remove
        if qty != 0 && qty.abs() < min_tokens {
            bail!(DeriverseErrorKind::InvalidQuantity {
                value: qty,
                min_value: min_tokens,
                max_value: SPOT_MAX_AMOUNT,
            });
        }
```

Because `min_qty` uses the unbounded px, **for bid orders**, an attacker can send `ioc != 0` with `LIMIT order` and choose an enormous `data.price`. `min_qty` then pulls a very small threshold from `PRICE_TOKENS`, allowing the attacker to submit dust-sized orders. The engine later clamps the executable price to within ±12.5 % of the mark, so the trade still goes through at the normal market price, but with an amount far below the intended minimum.

For example, the current price is `500_000_000`, the attacker/user can make `px=100_000_000_000_000` so that he can pay as little as `20_000`, making dust order.

Impact: Minimum order size limits for `IOC` orders can be bypassed. Attackers can generate large numbers of dust trades by supplying a large `data.price`.

Recommended Mitigation: Consider computing `min_qty` using the current market price.

Deriverse: Fixed in commit 134db8b.

Cyfrin: Verified.

7.2.27 Wrong accounting of fee in perp engine

Description: In the perp fee calculation logic within perp_engine.rs, when a user has partial fee prepayment that doesn't fully cover the trade fee, the function incorrectly returns discount_sum + extra_fee instead of `fees_prepayment + extra_fee`

The bug occurs because discount_sum represents the`trade value` covered by prepayment (calculated as [fees_prepayment / fee_rate]), not the fee amount itself. This results in a massively inflated fee return value.

```
                    } else {
                        let discount_sum = (*fees_prepayment as f64 / args.fee_rate) as i64;
                        let extra_fee = ((traded_crncy - discount_sum) as f64
                            * (1.0 - args.ref_discount)
                            * self.fee_rate) as i64;
                        let fee = discount_sum + extra_fee; // here, discount_sum is trade value, not
                        ↪  fee
                        taker_info
                            .sub_funds(extra_fee)
                            .map_err(|err| drv_err!(err))?;
                        *fees_prepayment = 0;
                        fee
                    }
```

Consider this scenario: INPUTS:

- traded_crncy = $10,000 (trade value)
- self.fee_rate = 0.001 (0.1% base fee)
- ref_discount = 0.20 (20% referral discount)
- args.fee_rate = 0.001 * 0.80 = 0.0008 (discounted rate)
- fees_prepayment = $5

STEP 1: Calculate `discount_fee` discount_fee = 0.0008 * $10,000 = $8 Is $8 <= $5? NO means we enter partial coverage branch

STEP 2: Calculate `discount_sum` discount_sum = $5 / 0.0008 = $6,250 (This is trade value covered from prepayment)

STEP 3: Calculate `extra_fee` remaining_trade = $10,000 - $6,250 = $3,750 extra_fee = $3,750 * 0.80 * 0.001 = $3

IT RETURNS (worng): fee = discount_sum + extra_fee = $6,250 + $3 = $6,253 : this is wrong, it's inflated

CORRECT RETURN: fee = fees_prepayment + extra_fee = $5 + $3 = $8 ← CORRECT

WHAT USER ACTUALLY PAYS:

- Prepayment consumed: $5
- extra_fee from funds: $3
- Total paid: $8 (correct amount deducted)

BUT PROTOCOL RECORDS: $6,253 as fee which is way too much inflated and totally incorrect

Impact: Protocol records inflated fee which will later on corrupt rest of accounting which depend on it.

Recommended Mitigation: In the fill() function, change:

```
} else {
    let discount_sum = (*fees_prepayment as f64 / args.fee_rate) as i64;
    let extra_fee = ((traded_crncy - discount_sum) as f64
        * (1.0 - args.ref_discount)
        * self.fee_rate) as i64;
    // FIX: Use `fees_prepayment` (actual fee amount), not discount_sum (trade value)
```

```
    let fee = (*fees_prepayment) + extra_fee;
    taker_info
        .sub_funds(extra_fee)
        .map_err(|err| drv_err!(err))?;
    *fees_prepayment = 0;
    fee
}
```

Deriverse Fixed in commit: 94da2f1

Cyfrin: Verified.

7.2.28 User's chosen leverage is overwritten to `max-leverage` on every perp operation

Description: In `client_primary.rs]`, the new_for_perp function unconditionally sets the user's leverage on every call, regardless of whether the `alloc` parameter is true or false.

```
    let mut client_state = ClientPrimaryState::new_for_perp(
        program_id,
        client_primary_acc,
        &ctx,
        signer,
        system_program,
        0, //always passed as zero
        false,
        true,
    )?;
```

When leverage = 0 is passed (which happens in most perp operations like `perp_withdraw`, `perp_deposit`, `perp_-mass_cancel`, `perp_order_cancel`, `perp_quotes_replace`, `perp_statistics_reset`, `buy_market_seat`, `sell_-market_seat`), the code sets the user's leverage to `max_leverage`.

```
if leverage > 0 {
    //clear and set
    unsafe {
        (*state.perp_info2).mask &= 0xFFFFFF00;
        (*state.perp_info2).mask |= (leverage as u32).min(header.max_leverage as u32);
    }
} else {
    // When leverage == 0, sets to max_leverage (always by default)
    unsafe {
        (*state.perp_info2).mask &= 0xFFFFFF00;
        (*state.perp_info2).mask |= header.max_leverage as u32;
    }
}
Ok(state)
```

consider this scenario:

1. User calls perp_change_leverage(5) to set their leverage to 5x
    - leverage stored as 5
2. User calls `perp_deposit()` to add more margin
    - `new_for_perp()` called with leverage=0
    - leverage RESET to `max_leverage` (e.g., 15x)
3. User now has 15x leverage instead of 5x

Impact: Users who carefully set their leverage to a conservative value (e.g., 2x or 3x) will have it silently reset to `max_leverage` after performing any perp operation. This significantly increases their liquidation risk without their knowledge.

Recommended Mitigation: Only update leverage when explicitly requested (i.e., when `leverage > 0`). Remove the else branch that overwrites leverage with `max_leverage`:

```
if leverage > 0 {
    // Only update leverage when explicitly provided
    unsafe {
        (*state.perp_info2).mask &= 0xFFFFFF00;
        (*state.perp_info2).mask |= (leverage as u32).min(header.max_leverage as u32);
    }
}
// When leverage == 0, keep existing leverage unchanged
```

Deriverse: Fixed in commit: https://github.com/deriverse/protocol-v1/commit/e15ad9372bf10322c6d05c234189576b8aad3690

Cyfrin: Verified.

7.2.29 Redundant State Updates in `fill` Function Cause Issues

Description: The `fill` function contains **redundant state updates** to `last_asset_tokens` and `last_crncy_tokens` that are already properly handled by `write_last_tokens`. This redundancy causes data loss, incorrect accumulation, and state inconsistency when multiple orders are filled within the same slot.

- The Redundant Update In the `fill` function, `last_asset_tokens` and `last_crncy_tokens` are redundantly updated:

```
self.state.header.last_asset_tokens = traded_qty;
self.state.header.last_crncy_tokens = traded_crncy;
```

After `match_orders` completes, `write_last_tokens` is already called with the accumulated values:

```
engine.write_last_tokens(traded_qty, traded_sum, trades, px)?;
```

The `write_last_tokens` function properly handles these state updates with:

- Slot boundary checking
- Accumulation logic when slots match
- Reset logic when slots differ

```
if self.state.header.slot == self.slot {
    self.state.header.last_crncy_tokens = self
        .state
        .header
        .last_crncy_tokens
        .checked_add(traded_crncy_tokens)
        .ok_or(drv_err!(DeriverseErrorKind::ArithmeticOverflow))?;
    self.state.header.last_asset_tokens = self
        .state
        .header
        .last_asset_tokens
        .checked_add(traded_asset_tokens)
        .ok_or(drv_err!(DeriverseErrorKind::ArithmeticOverflow))?;
} else {
    self.state.header.slot = self.slot;
    self.state.header.last_crncy_tokens = traded_crncy_tokens;
    self.state.header.last_asset_tokens = traded_asset_tokens;
}
```

Problems Caused by the Redundancy

- Data Loss in fill Loop: Within the `fill` function's loop, each iteration overwrites the previous values, only preserving the last order's values:

```
while !order.is_null() && *remaining_qty > 0 {
    // ... process order ...
    self.state.header.last_asset_tokens = traded_qty;  // Overwrites previous value!
    self.state.header.last_crncy_tokens = traded_crncy; // Overwrites previous value!
}
```

- Multiple fill Calls: The `match_orders` function can call `fill` multiple times, each overwriting the state with only partial data.
- Incorrect Accumulation: When `write_last_tokens` is subsequently called. If the slot matches, it adds the total accumulated values to the incorrectly set values from `fill`. This causes double counting or incorrect totals

Example: Before, the `last_asset_tokens = 200`, If `fill` sets `last_asset_tokens = 100` (last order only), then `write_last_tokens` adds the total `traded_qty = 500`, resulting in `600` instead of `200 + 500`.

Impact: Incorrect State: When `write_last_tokens` accumulates values, it adds to incorrectly set values, leading to wrong totals

Recommended Mitigation: Remove the redundant state updates from the `fill` function.

Deriverse: Fixed in commit 0be264f1.

Cyfrin: Verified.

7.2.30 Missing Trade Count Updates in `reversed_swap` Function for Buy Orders

Description: The `reversed_swap` function used for `buy` orders in `swap` operations fails to properly track and return `trade` counts, leading to incorrect trade statistics being written to the state. This inconsistency with `match_orders` causes buy order trades to be recorded as 0, resulting in inaccurate trading volume and trade count metrics.

In the `swap` flow, buy orders use `reversed_swap` while sell orders use `match_orders`.

```
if buy {
    if price > px || engine.cross(price, OrderSide::Ask) {
        let total_fees;
        let remaining_sum;
        let input_sum = (data.amount as f64 / (1.0 + fee_rate)) as i64;

        (remaining_sum, traded_qty, total_fees) = engine.reversed_swap(price, input_sum)?;
```

There are 2 issues:

Issue 1: Missing `trades` Return Value

The `reversed_swap` function tracks trades internally but does not return this value:

```
            (remaining_sum, traded_qty, total_fees) = engine.reversed_swap(price, input_sum)?;
```

It returns `(remaining_sum, qty, total_fees)` but not `trades`, even though it maintains a trades variable internally.

Thus, the `trades` variable remains `0`, and this incorrect value is passed to `write_last_tokens`:

```
if traded_qty > 0 && traded_sum > 0 {
    let candles = uninit_candles.init(&engine)?;
    engine.write_candles(candles, traded_qty, traded_sum)?;
    engine.write_last_tokens(traded_qty, traded_sum, trades, px)?;
} else {
    bail!(DeriverseErrorKind::FailedToSwap {
        price,
```

```
            side: if buy { OrderSide::Ask } else { OrderSide::Bid }
        });
    }
```

Issue 2: Missing Trade Count Increment for AMM Trades Unlike match_orders, reversed_swap does not increment `trades` after AMM transactions. In `match_orders`, every AMM trade increments the counter:

```
self.change_tokens(traded_qty, side)?;
self.change_mints(traded_mints, side)?;
self.log_amm(traded_qty, traded_mints, side);
self.set_px();
trades += 1;
```

However, in `reversed_swap`, AMM trades occur without incrementing `trades`:

```
self.change_tokens(traded_qty, side)?;
self.change_mints(traded_mints, side)?;
self.log_amm(traded_qty, traded_mints, side);
self.set_px();
total_fees = total_fees
    .checked_add((traded_mints as f64 * self.fee_rate) as i64)
    .ok_or(drv_err!(DeriverseErrorKind::ArithmeticOverflow))?;
break;
```

Similar omissions occur where AMM trades are executed without incrementing `trades`.

Impact:

- Incorrect trade statistics: Buy order trades are recorded as 0.
- Data inconsistency: Buy and sell orders are handled differently, causing asymmetric reporting.

Recommended Mitigation: Modify `reversed_swap` to return `trades` and increase `trades` after each AMM trade to be consistent with `match_orders`.

Deriverse: Cyfrin:

7.2.31 Missing `change_funding_rate` Call After Price Update in `perp_mass_cancel` and `perp_order_cancel`

Description: In `perp_order_cancel` and `perp_mass_cancel functions`, the code updates the underlying price (`perp_underlying_px`) via `set_underlying_px` but fails to call `change_funding_rate` before invoking `check_rebalancing`(and potentially `mass_cancel`), which internally calls `check_funding_rate`. This results in funding rate calculations being performed with stale global funding rate values that haven't been updated to reflect the new underlying price(even though the price is correctly updated), potentially leading to incorrect funding fee calculations for users.

The funding rate mechanism works as follows:

- `change_funding_rate` updates the global funding rate state

```
    pub fn change_funding_rate(&mut self) {
        let time_delta = self.time - self.state.header.perp_funding_rate_time;
        if time_delta > 0 && self.state.header.perp_price_delta != 0.0 {
            self.state.header.perp_funding_rate +=
                ((time_delta as f64) / DAY as f64) * self.state.header.perp_price_delta;
        }
        self.state.header.perp_price_delta =
            (self.market_px() - self.state.header.perp_underlying_px) as f64 * self.rdf;
        self.state.header.perp_funding_rate_time = self.time;
    }
    /*
```

- `check_funding_rate` applies the global funding rate to individual clients:

```rust
    pub fn check_funding_rate(&mut self, temp_client_id: ClientId) -> Result<bool, DeriverseError> {
        let info = unsafe { &mut *(self.client_infos.offset(*temp_client_id as isize)) };
        let info5 = unsafe { &mut *(self.client_infos5.offset(*temp_client_id as isize)) };
        let perps = info.total_perps();
        let mut change = false;
        if perps != 0 {
            if self.state.header.perp_funding_rate != info5.last_funding_rate {
                let funding_funds = -(perps as f64
                    * (self.state.header.perp_funding_rate - info5.last_funding_rate))
                    .round() as i64;
```

Thus the `perp_funding_rate` should be refreshed each time before `check_funding_rate` is called.

The issue is that in the `perp_order_cancel` and `perp_mass_cancel`:

```rust
// perp_order_cancel
engine.state.set_underlying_px(accounts_iter)?;
// ... order cancellation logic ...
engine.check_rebalancing()?;  // Calls check_funding_rate() internally

// perp_mass_cancel
engine.state.set_underlying_px(accounts_iter)?;
engine.mass_cancel(client_state.temp_client_id)?;  // Calls check_funding_rate() at line 1888
// ... margin call checks ...
engine.check_rebalancing()?;  // Also calls check_funding_rate() at line 2363
```

When `perp_underlying_px` is updated but `change_funding_rate` is not called, the global `perp_funding_rate` may not reflect the latest price changes.

Impact: Users may be charged incorrect funding fees when canceling orders, as the funding rate calculations use stale global funding rate values that don't reflect the updated underlying price since the `change_funding_rate` is not called.

Recommended Mitigation: Add `change_funding_rate` calls immediately after `set_underlying_px` in both vulnerable functions.

Deriverse: Fixed in commit 74f9650.

Cyfrin: Verified.

7.2.32 Inefficient rebalancing can cause the loss of users

Description: We can only call rebalance once every 5 minutes. This is enforced through the `check_rebalancing` function, which allows only 25 rebalancing calls each time it runs. This limitation can create significant issues.

For example, if there are 200,000 open positions and `check_rebalancing` is triggered whenever a user performs a perp-related action, we would need roughly 8,000 user-triggered calls every 5 minutes to rebalance all positions. If the number of user actions is lower, many positions will not be rebalanced in time.

This can result in positions being rebalanced too late, which may lead to improper liquidations(early or late liquidation) and potentially increase socialized losses for other users.

Impact: This limitation can lead to positions not being rebalanced until much later if perp-related transactions are very low, which can result in improper liquidations.

Recommended Mitigation: Implement a function that can be called to rebalance user positions every 5 minutes.

Deriverse: Fixed in commit 7873db, 76dbbe.

Cyfrin: Verified.

7.2.33 Incorrect Margin Call State Detection After Liquidation in `perp_withdraw`

Description: In perp_withdraw, the margin call state is incorrectly determined after executing liquidation checks. The function calls `check_long_margin_call()` and `check_short_margin_call()` first which may successfully liquidate positions, but then checks the `margin_call` state using `is_long_margin_call()` and `is_short_margin_call()`. Since these check functions examine the current highest-risk node in the tree (which may have changed after liquidation), they can return `false` even when liquidation has just occurred, leading to incorrect withdrawal amount calculations.

The issue occurs in `src/program/processor/perp_withdraw.rs`:

```
engine.check_long_margin_call()?;
engine.check_short_margin_call()?;
let margin_call = engine.is_long_margin_call() || engine.is_long_margin_call();
```

The Problem:

- Liquidation Execution: `check_long_margin_call` iterates through the `long_px` tree, finds high-risk positions(`max_node`), and liquidates them. When a position is successfully liquidated, it calls `change_edge_px(temp_client_id)`, which removes or updates the node in the tree.

```
                let (_, t, _) = self.match_bid_orders(
                    None,
                    &MatchOrdersStaticArgs {
                        price: margin_call_px,
                        qty: info.perps,
                        ref_discount: 0f64,
                        ref_ratio: 0f64,
                        ref_expiration: 0,
                        ref_client_id: ClientId(0),
                        client_id: temp_client_id,
                        trades_limit: MAX_MARGIN_CALL_TRADES - trades,
                        margin_call: true,
                    },
                )?;
                self.check_funding_rate(temp_client_id)?;
                self.change_edge_px(temp_client_id); // <= here
```

```
    if info2.px_node != NULL_NODE {
        if info2.mask & 0x80000000 == 0 {
            let node: NodePtr<i128> =
                unsafe { NodePtr::get(self.long_px.entry, info2.px_node) };
            self.long_px.delete(node);
        } else {
            let node: NodePtr<i128> =
                unsafe { NodePtr::get(self.short_px.entry, info2.px_node) };
            self.short_px.delete(node);
        }
    }
```

- State Check After Liquidation: After liquidation, `is_long_margin_call()` checks the current `max_node()` in the `long_px` tree. However, this node may be different from the one that was just liquidated. And could have different liquidation status.

```
pub fn is_long_margin_call(&self) -> bool {
    let root = self.long_px.get_root::<i128>();
    if root.is_null() {
        false
    } else {
        ((self.state.header.perp_underlying_px as f64
            / (1.0 + self.state.header.liquidation_threshold)) as i64)
            < (root.max_node().key() >> 64) as i64
    }
```

```
    }
```

False Negative Scenario:

- Node A is the highest-risk node (`px > margin_call_px`)
- `check_long_margin_call` successfully liquidates Node A
- Node A is removed/updated via `change_edge_px`
- The loop continues and checks Node B
- If Node B's `px <= margin_call_px`, the loop breaks
- `is_long_margin_call` checks Node B and returns false But liquidation has already occurred(for Node A), so the system should be considered in a margin call state

If liquidation occurred but `margin_call` is incorrectly set to `false`, the withdrawal calculation uses the normal path instead of the margin call path, causing incorrect `amount` calculation.

```
    let amount = if margin_call {
        let margin_call_funds =
            funds.min(engine.get_avail_funds(client_state.temp_client_id, true)?);
        if margin_call_funds <= 0 {
            bail!(ImpossibleToWithdrawFundsDuringMarginCall);
        }
        if data.amount == 0 {
            margin_call_funds
        } else {
            if margin_call_funds < data.amount {
                if funds >= data.amount {
                    bail!(ImpossibleToWithdrawFundsDuringMarginCall);
                }
                bail!(InsufficientFunds);
            }
            data.amount
        }
    } else if data.amount == 0 {
        funds
    } else {
        if funds < data.amount {
            bail!(InsufficientFunds);
        }
        data.amount
    };
```

Also, `check_rebalancing` is unnecessarily called.

```
    if !margin_call {
        engine.check_rebalancing()?;
    }
```

Impact: If liquidation occurred but `margin_call` is incorrectly set to false, the function would behave different according to `margin_call`.

Recommended Mitigation: Check Return Values: `check_long_margin_call` and `check_short_margin_call` return the number of trades executed. Use these return values to determine if liquidation occurred.

Deriverse: Fixed in commit 74f9650.

Cyfrin: Verified.

7.2.34 Stale edge price in liquidation tree after perp withdraw call

Description: In `perp_withdraw`, after withdrawing funds from a perpetual position, the function does not call `change_edge_px` to update the liquidation price(edge price) in the liquidation tree.

When funds are withdrawn, `info.funds` decreases, which affects `total_funds` and consequently the edge price. However, the liquidation trees(`long_px` and `short_px`) continue to store the outdated edge price because `change_edge_px` is never invoked.

Impact: After a withdrawal, the liquidation trees retain stale edge prices because `change_edge_px` is never called. This leads to incorrect margin call detection and liquidation priority.

Recommended Mitigation: `change_edge_px` should be invoked for the user withdrawing funds at the end of the function.

Deriverse: Fixed in commit 4f7bc8.

Cyfrin: Verified.

7.2.35 Missing Update of `perp_spot_price_for_withdrowal` in `perp_withdraw` Function

Description: The `perp_withdraw` function fails to update the `perp_spot_price_for_withdrowal` field(including `perp_long_spot_price_for_withdrowal` and `perp_short_spot_price_for_withdrowal` in later commits) when there is no margin call, while other similar functions (`new_perp_order` and `perp_quotes_replace`) properly update this field. This inconsistency can lead to stale price data being used in subsequent margin call scenarios, potentially affecting withdrawal calculations.

The `perp_spot_price_for_withdrowal` field is used by the `get_avail_funds` function when calculating available funds during margin call situations. The field should be updated to the current `perp_underlying_px` when there is no margin call to ensure accurate calculations in future operations.

We can see that in `new_perp_order` and `perp_quotes_replace`:

```
    if !long_margin_call {
        engine.state.perp_long_spot_price_for_withdrowal = engine.state.perp_underlying_px;
    }
    if !short_margin_call {
        engine.state.perp_short_spot_price_for_withdrowal = engine.state.perp_underlying_px;
    }
```

Now, in `perp_withdraw`, we are having:

```
let margin_call = engine.is_long_margin_call() || engine.is_long_margin_call();
if !margin_call {
    engine.check_rebalancing()?;
}
```

Impact: The `perp_spot_price_for_withdrowal` field(including `perp_long_spot_price_for_withdrowal` and `perp_short_spot_price_for_withdrowal` in later commits) may retain stale values if `perp_withdraw` is called without a margin call, while other functions update it. If a margin call occurs after a `perp_withdraw` operation (in the same transaction or subsequent transactions), the `get_avail_funds` function may use an outdated price when `margin_call=true`, leading to incorrect available funds calculations.

Recommended Mitigation: Add the missing update to maintain consistency and price up-to-date.

Deriverse: Fixed in commit 66c878.

Cyfrin: Verified.

7.2.36 Margin call uses stale edge price

Description: The `check_long_margin_call` and `check_short_margin_call` functions determine whether positions should be liquidated based on edge prices retrieved from the price trees before applying funding rate updates

and rebalancing. This causes positions to be liquidated using stale edge prices that don't reflect the current state after funding rate changes, leading to unfair liquidations.

`check_funding_rate` and `check_soc_loss` are called after the liquidation decision is made, but these functions can modify the user's funds, which directly affects whether liquidation should occur.

```
pub fn check_long_margin_call(&mut self) -> Result<i64, DeriverseError> {
    let mut trades = 0;
    let margin_call_px = (self.state.header.perp_underlying_px as f64
        * (1.0 - self.state.header.liquidation_threshold)) as i64;

    loop {
        let root = self.long_px.get_root::<i128>();
        if root.is_null() || trades >= MAX_MARGIN_CALL_TRADES {
            break;
        }
        let node = root.max_node();
        let px = (node.key() >> 64) as i64;  // Gets stale edge price

        if px > margin_call_px {  // Decision made with stale price
            let temp_client_id = ClientId(node.link());

            // Funding rate checked AFTER liquidation decision
            self.check_funding_rate(temp_client_id)?;
            self.check_soc_loss(temp_client_id)?;

            // ... liquidation logic ...

            // Edge price updated AFTER liquidation
            self.change_edge_px(temp_client_id);
        }
    }
}
```

Impact: Users may be liquidated when they should not be, resulting in unnecessary loss of funds.

Recommended Mitigation: Consider using the updated edge price, after applying the `check_funding_rate` and `check_soc_loss` to determine whether the position should be liquidated.

Deriverse: Fixed in commit 1efef6.

Cyfrin: Verified.

7.2.37 Insurance fund decrease when `margin_call_penalty_rate` is less than `rebates_rate`

Description: When `margin_call_penalty_rate` is less than `rebates_rate`(`fee_rate * REBATES_RATIO`), the insurance fund decreases instead of increasing during margin calls.

During a margin call, the insurance fund is updated as: `perp_insurance_fund += (new_fees - rebates)`, where:

- `new_fees = traded_crncy * margin_call_penalty_rate`
- `rebates = traded_crncy * fee_rate * REBATES_RATIO`

If `margin_call_penalty_rate < fee_rate * REBATES_RATIO`, then `(new_fees - rebates)` becomes negative, causing the insurance fund to decrease when it should increase.

Impact: The insurance fund can be drained during margin calls instead of being replenished.

Recommended Mitigation: Ensure that `margin_call_penalty_rate >= fee_rate * REBATES_RATIO` to guarantee that margin call penalties always contribute positively to the insurance fund.

Deriverse: Fixed in commit 1ef948.

Cyfrin: Verified.

7.3 Low Risk

7.3.1 Casting from `u64` to `i64` causes genuine deposit requests to fail in `deposit` function

Description: The `deposit` function in `deposit.rs` casts the amount value(the amount of tokens trader is willing to deposit) which is of type `u64` to `i64` when `deposit_all` is set to `true`. When token amounts exceed `i64::MAX` (9,223,372,036,854,775,807), the cast wraps around to negative values due to Rust's default overflow behavior in release builds. The amount is later cast back to u64 for SPL token transfers, however we have this check before casting it back up to `u64`

```
    if !(1..=SPOT_MAX_AMOUNT).contains(&amount) {
        bail!(InvalidQuantity {
            value: amount,
            min_value: 1,
            max_value: SPOT_MAX_AMOUNT,
        });
    }
```

here the goal is to put lower and upper bound on the amount between 1 & 36028797018963967, since we have casted the amount from `u64` to `i64`, if the amount was big, It might have turned to a negative number and this negative number does not lie in the desired range, so the transaction does not go through.

Impact: Genuine transactions especially for large token decimal mints may get reverted.

Proof of Concept:

```
fn main() {
    let a: i64;

    let b: u64 = 15_000_000_000_000_000_000;

    a = b as i64;  // Casting

    println!("a = {}", a);  //
}
Output: a = -3446744073709551616
```

Recommended Mitigation: Don't convert input amount to `i64` instead we can do this

```
const SPOT_MAX_AMOUNT_U64: u64 = SPOT_MAX_AMOUNT as u64;
```

Deriverse: Fixed in commit 801209

Cyfrin: Verified.

7.3.2 Expired Private Client Cannot Be Re-added to Queue

Description: The `new_private_client()` function incorrectly rejects re-adding a wallet whose previous entry has expired. The function checks for duplicate wallet addresses before validating expiration status, preventing expired records from being treated as vacant slots.

In the record insertion logic (lines 131-133), the code checks if a wallet already exists:

```
if record.wallet == *wallet.key && record.creation_time != 0 {
    return Err(AlreadyExists(index));
}
```

However, this check occurs before the expiration validation. According to the `PrivateClient::is_vacant()` method (defined in `src/state/private_client.rs`), a record should be considered vacant if either:

1. `creation_time == 0` (uninitialized), or
2. `current_time > expiration_time` (expired)

The problem is that when iterating through records to find an insertion position, the duplicate check at line 131 returns an error immediately when a matching wallet is found, regardless of expiration status. This prevents the code from reaching the `is_vacant()` check at line 136, which would correctly identify expired records as reusable slots.

Impact: Queue Slot Exhaustion: Expired private client cannot be renewed.

Recommended Mitigation: Modify the duplicate wallet check to validate expiration status before returning an error. Only return `AlreadyExists` if the wallet matches and the record is still valid (not expired).

Deriverse: Fixed in commit a626a26.

Cyfrin: Verified.

7.3.3 Account Count Validation Mismatch in `new_base_crncy` Instruction

Description: The `new_base_crncy` instruction has an inconsistency between the declared minimum account count and the actual number of accounts required.

The function comment and implementation both indicate that `9` accounts are needed, but `NewBaseCrncyInstruction::MIN_ACCOUNTS` is set to `8`.

```
pub fn new_base_crncy(program_id: &Pubkey, accounts: &[AccountInfo], _: &[u8]) -> DeriverseResult {
    // New Base Currency Instruction
    // 1 - Admin (Signer)
    // 2 - Root Account (Read Only)
    // 3 - Token Account
    // 4 - Program Token Account (Signer when creating a new token)
    // 5 - Deriverse Authority Account (Read Only)
    // 6 - Token Program ID (Read Only)
    // 7 - Mint Address (Read Only, when wSOL can not be old native mint)
    // 8 - System Program (Read Only)
    // 9 - Community Account
```

The function first reads 8 accounts, then later reads the 9th account:

```
    let community_acc = next_account_info!(accounts_iter)?;
```

However, the validation check only requires 8 accounts:

```
    if accounts.len() < NewBaseCrncyInstruction::MIN_ACCOUNTS {
        bail!(InvalidAccountsNumber {
            expected: NewBaseCrncyInstruction::MIN_ACCOUNTS,
            actual: accounts.len(),
        });
    }
```

Where `MIN_ACCOUNTS` is defined as:

```
    pub struct NewBaseCrncyInstruction;
    impl DrvInstruction for NewBaseCrncyInstruction {
        const INSTRUCTION_NUMBER: u8 = 4;
        const MIN_ACCOUNTS: usize = 8;
    }
```

Impact:

1. **Incorrect Validation**: The instruction accepts `8` accounts when it actually requires `9`, creating a mismatch between the validation and the actual account requirements.
2. **Late Failure**: Transactions with only `8` accounts will pass the initial check but fail later during execution.

Recommended Mitigation: Update `MIN_ACCOUNTS` to `9` to match the actual account requirements:

```rust
pub struct NewBaseCrncyInstruction;
impl DrvInstruction for NewBaseCrncyInstruction {
    const INSTRUCTION_NUMBER: u8 = 4;
    const MIN_ACCOUNTS: usize = 9;  // Changed from 8 to 9
}
```

Deriverse: Fixed in commit f54117b0.

Cyfrin: Verified.

7.3.4 Inflexible Voting System Prevents Rapid Parameter Adjustments

Description: The governance voting system uses a rigid rotation mechanism where which parameter can be modified is determined by `voting_counter % 6`.

This creates a fixed 6-parameter rotation cycle where each parameter can only be voted on once every 6 voting periods. Combined with the 14-day voting period duration, this means a specific parameter can only be modified again after approximately 84 days (6 periods × 14 days), severely limiting the protocol's ability to respond to urgent situations or make consecutive adjustments to the same parameter.

The parameter selection is determined in `finalize_voting()`:

```rust
let tag = community_account_header.voting_counter % 6;
match tag {
    0 => spot_fee_rate,
    1 => perp_fee_rate,
    2 => spot_pool_ratio,
    3 => margin_call_penalty_rate,
    4 => fees_prepayment_for_max_discount,
    _ => max_discount,
}
```

The `voting_counter` can only increment by 1 in `finalize_voting()`:

```rust
community_account_header.voting_counter += 1;
```

And in `next_voting()`, the counter can only be incremented to 1 if it's 0, or finalized (which increments by 1):

```rust
if community_state.header.voting_counter == 0 {
    community_state.header.upgrade()?.voting_counter += 1;
}
community_state.finalize_voting(clock.unix_timestamp as u32, clock.slot as u32)?;
```

The Problem:

1. There is no mechanism to skip voting rounds or target a specific parameter directly
2. The `voting_counter` can only increment sequentially, never skip ahead
3. If a parameter needs urgent adjustment or consecutive modifications, the protocol must wait through the entire 6-parameter cycle
4. No emergency mechanism exists for operator or admin to override the rotation schedule

Example Scenario:

1. Voting period 1 (`voting_counter` = 1): Community votes to decrease `perp_fee_rate` (tag `1`)
2. After 14 days, the change is applied, `voting_counter` becomes 2
3. Market conditions change, requiring another immediate adjustment to `perp_fee_rate`
4. The protocol must wait for voting_counter = `7, 13, 19`, etc. (every 6th period)

5. This means waiting approximately `70 days` (5 more periods × 14 days) before `perp_fee_rate` can be voted on again

```
#[cfg(not(feature = "test-sbf"))]
pub fn voting_end(time: u32) -> u32 {
    let days = (time - SETTLEMENT) / DAY;
    days * DAY + 14 * DAY + SETTLEMENT
}
```

Impact:

- **Delayed Response to Market Conditions:** The protocol cannot quickly respond to urgent situations requiring consecutive parameter adjustments
- **Inefficient Governance:** If a parameter needs multiple adjustments to reach an optimal value, the process takes months(5*14 = 70 days) instead of weeks

Recommended Mitigation: Consider adding an optional mechanism to allow the operator to specify the next `voting_counter` value in extreme circumstances, while maintaining the default sequential increment for normal operations.

Deriverse:: Fixed in commit bb853ad.

Cyfrin: Verified.

7.3.5 Dividend Calculation Uses Stale Token Balance in Subsequent `update()` Calls After `fees_deposit` with DRVS

Description: The `fees_deposit()` function calls `client_community_state.update()` before `client_state.sub_crncy_tokens()`, causing `self.header.drvs_tokens` to be updated with a stale value that doesn't reflect the actual token balance after the reduction. When `update()` is called again in subsequent instructions (e.g., `deposit`, `spot_quotes_replace`), it calculates dividends using this stale stored value instead of the actual current balance.

Prerequisites: This issue only occurs when `fees_deposit` is called with `token_id = 0` (DRVS token)

The issue occurs because:

1. **Order of operations**: In `fees_deposit()`, `update()` is called before `sub_crncy_tokens()`
2. **Stale balance storage**: `update()` reads the current balance (before reduction) and stores it in `self.header.drvs_tokens`, but `sub_crncy_tokens()` reduces the actual balance afterward
3. **Stale dividend calculation**: When `update()` is called again in a subsequent instruction, it calculates dividends using `self.header.drvs_tokens` (the stale stored value from before the reduction) instead of the actual current balance

```
// src/program/processor/fees_deposit.rs
client_community_state.update(&mut client_state, &mut community_state)?;
client_state.resolve(AssetType::Token, data.token_id, TokenType::Crncy, false)?;
// ... fee calculation logic ...
client_state.sub_crncy_tokens(data.amount)?;
```

```
// src/state/client_community.rs
        if available_tokens != self.header.drvs_tokens {
            for (i, d) in self.data.iter_mut().enumerate() {
                let amount = (((community_state.base_crncy[i].rate - d.dividends_rate)
                    * self.header.drvs_tokens as f64) as i64)
                    .max(0);
                d.dividends_value += amount;
            }
```

Example scenario demonstrating the issue:

Initial state: User has 100 DRVS tokens, `self.header.drvs_tokens = 100`

1. **First** `fees_deposit(token_id=0, amount=10)` **call:**
   - `update()` is called:
     - `resolve(AssetType::Token, 0, ...)` resolves DRVS token (token_id = 0)
     - `available_tokens = 100` (current balance before reduction)
     - `self.header.drvs_tokens = 100` (stored value)
     - Since they're equal, no dividend calculation occurs
     - `self.header.drvs_tokens` is updated to `100`
   - `sub_crncy_tokens(10)` is called, actual balance becomes `90`
   - **Result**: `self.header.drvs_tokens = 100` (stale), but `actual balance = 90`
2. **Next instruction that calls** `update()` **(e.g.,** `deposit`**,** `spot_quotes_replace`**):**
   - `update()` is called again:
     - `available_tokens = 90` (actual current balance, or could be others)
     - `self.header.drvs_tokens = 100` (stale stored value from step 1)
     - Since they differ, dividend calculation occurs using `self.header.drvs_tokens = 100`
     - **Problem**: Dividends are calculated based on `100` tokens, but the user only has `90` tokens during this period
   - **Result**: User receives dividends calculated on `100` tokens instead of `90`, receiving more than they should

The root cause is that in `fees_deposit()`, `update()` stores the balance before `sub_crncy_tokens()` reduces it, creating a discrepancy between the stored value and the actual balance. When `update()` is called again later, it uses this stale stored value for dividend calculations.

Impact: Overpayment of Dividends: When `update()` is called in subsequent instructions after a `fees_deposit(token_id=0)`, users receive dividends calculated on a higher token balance than they actually hold, leading to financial loss for the protocol.

Recommended Mitigation: The order should be re-arranged to ensure that the `drvs_token` is always update to date

Deriverse: Fixed in commit 4df80d.

Cyfrin: Verified.

7.3.6 Griefing Attack: Malicious Takers Can Force Order Cancellation by Partial Filling Below Minimum Quantity

Description: A malicious taker can exploit the automatic order cancellation mechanism to force makers' orders to be cancelled by intentionally partially filling orders such that the remaining quantity falls below the `min_qty` threshold.

When this occurs, the system automatically cancels the order and refunds the remaining locked funds to the maker. However, since the refunded amount is below `min_qty`, the maker cannot place a new order with these funds, effectively creating a griefing attack that disrupts normal trading operations.

The vulnerability exists in the `fill` function of the spot trading engine . When an order is partially filled (last = true), the system will automatically cancels the order via `erase_client_order(order, node, true, side)`.

```
// src/program/spot/engine.rs:1265-1279
if last {
    order.decr_qty(traded_qty).map_err(|err| drv_err!(err))?;
```

```
    order.decr_sum(traded_crncy).map_err(|err| drv_err!(err))?;
    order.set_time(self.time);
    if order.qty() < min_qty {  //  check
        let node = self.get_node_ptr(order.link(), fill_static_args.side);
        self.erase_client_order(order, node, true, fill_static_args.side)?;
        // ... order is cancelled and funds refunded
    }
}
```

Attack Scenario:

- Maker places an order with quantity 100 tokens, where `min_qty = 2`
- Malicious taker intentionally matches 99 tokens, leaving `1` token remaining
- Since `1 < min_qty (2)`, the system automatically cancels the order
- The remaining `1` token is refunded to the maker via `erase_client_order`
- The maker cannot place a new order with the refunded 1 token because it's below `min_qty`

Impact:

- Griefing Attack: Attackers can systematically target orders and force their cancellation by leaving dust amounts below `min_qty`.
- **Refunded amounts below `min_qty` cannot be used to place new orders, effectively locking small amounts of funds**. I think this is not restricted to the order cancellation process, but is throughout the whole repo as it's uncertain how to deal with the dust amount

Recommended Mitigation: For this, I have two possible suggestions:

- Prevent Automatic Cancellation on Partial Fill
- Add a way for the user to sell dust amount directly on AMM

Deriverse: Fixed in commit 134db8b.

Cyfrin: Verified. User's can use market order to deal with tiny amounts

7.3.7 Missing `last_time` Update in `spot_lp` Causes Incorrect Daily Trade Statistics

Description: The `spot_lp` function uses `instr_state.header.last_time` to determine if a day has passed, but never updates this field.

In contrast, other trading functions (`swap, new_spot_order, spot_quotes_replace`) update `last_time` via `engine.write_last_tokens()`. This inconsistency causes incorrect daily LP trade statistics when regular trading operations haven't occurred recently.

In `src/program/processor/spot_lp.rs`, the function checks if a day has passed using:

```
if instr_state.header.last_time < fixing_time {
    instr_state.header.lp_prev_day_trades = instr_state.header.lp_day_trades;
    instr_state.header.lp_day_trades = 1;
} else {
    instr_state.header.lp_day_trades += 1;
}
```

However, `instr_state.header.last_time` is never updated in the `spot_lp` function, while other trading functions update it.

Consider the following case:

- Last regular trade (swap/new_spot_order) occurred 1 day ago, setting last_time to that timestamp
- Multiple spot_lp operations occur today

- For each spot_lp:
  - `last_time` remains from 1 day ago
  - `fixing_time` is today's settlement time
  - `last_time < fixing_time` is always true
  - `lp_day_trades` is reset to 1 instead of incrementing

Impact: In such case(last regular trade occurred 1 day ago), all LP trades on the same day are counted as the first trade of the day, losing accurate daily statistics.

Recommended Mitigation: Update `last_time` in the `spot_lp` function after the day-crossing check.

```
instr_state.header.lp_day_trades += 1;
}
instr_state.header.last_time = time;  // Add this line
```

Deriverse: Fixed in commit 29281c5.

Cyfrin: Verified.

7.3.8 Inconsistent Price Reference Used for Trade Execution Logic in `new_spot_order` and `swap`

Description: The `new_spot_order` and `swap` functions use different methods to obtain the reference price (`px`) for determining whether orders should execute, which creates inconsistency in the codebase.

In `src/program/processor/new_spot_order.rs`:

```
let px = engine.market_px();
```

In `src/program/processor/swap.rs`:

```
let px = engine.state.header.last_px;
```

In `src/program/processor/spot_quotes_replace.rs`:

```
let px = engine.state.header.last_px;
```

Difference:

The `market_px()` function (lines 1898-1906 in `engine.rs`) returns:

- `best_ask` if `best_ask < last_px`
- `best_bid` if `best_bid > last_px`
- `last_px` otherwise

This means `market_px()` may return an order book price (`best_ask` or `best_bid`) rather than the actual last traded price (`last_px`).

Impact: While both functions use `|| engine.cross(...)` as a fallback, the different `px` values may not directly cause security vulnerabilities, it may lead to subtle behavioral differences since:

- The `px` value is passed to `drv_update`, which uses it for fixing price calculations.
- Also, the `px` value is passed to `write_last_tokens`, which uses it to set `last_close` and `day_low` when crossing day boundaries.

Recommended Mitigation: Standardize the price reference across all spot trading functions.

Deriverse: Fixed in commit f7e57dc.

Cyfrin: Verified

7.3.9 Missing Signer and New Account Validation for `asset_token_program_acc` in `new_instrument`

Description: The `new_instrument` instruction lacks validation checks for `asset_token_program_acc` when creating a new asset token account. Unlike `new_base_crncy` which explicitly validates that the program token account is a signer and a new account, `new_instrument` omits these checks, leading to inconsistent error handling.

In new_instrument.rs, when `is_new_account(asset_token_acc)` is true, the code directly calls `TokenState::create_token` which internally calls `create_programs_token_account`. This function requires `asset_token_program_acc` to be a signer because it performs system operations:

```
    if is_new_account(asset_token_acc) {
        let decimals = *asset_mint
            .data
            .borrow()
            .get(MINT_DECIMALS_OFFSET)
            .ok_or_else(|| drv_err!(DeriverseErrorKind::InvalidClientDataFormat))?
            as u32;

        if !(MIN_DECS_COUNT..=MAX_DECS_COUNT).contains(&decimals) {
            bail!(DeriverseErrorKind::InvalidDecsCount {
                decs_count: decimals,
                min: MIN_DECS_COUNT,
                max: MAX_DECS_COUNT,
                token_address: *asset_mint.key,
            });
        }

        #[cfg(feature = "native_mint_2022")]
        if *asset_mint.key == spl_token::native_mint::ID {
            bail!(LegacyNativeMintNotSupported);
        }
        #[cfg(not(feature = "native_mint_2022"))]
        if *asset_mint.key == spl_token_2022::native_mint::ID {
            bail!(Token2022NativeMintNotSupported);
        }

        TokenState::create_token(
            root_state,
            asset_mint,
            asset_token_acc,
            drvs_auth_acc,
            &drvs_auth,
            bump_seed,
            program_id,
            asset_token_program_acc,
            token_program,
            signer,
            decimals,
        )?;
```

However, new_instrument does not validate:

- Whether `asset_token_program_acc.is_signer` is true
- Whether `asset_token_program_acc` is a new account (`is_new_account(asset_token_program_acc)`)

This is required, however, in the comment, indicating that when creating a new token, this account should be a signer.

```
///
/// [*Incorrect Price Validation When Creating `NewInstrumentData` Struct during
↪  `NewInstrumentInstruction` instruction*](#incorrect-price-validation-when-creating-newinstrumentdat
↪  a-struct-during-newinstrumentinstruction-instruction) - Asset Tokens Program Account `[SPL, if
↪  new_token signer]` - Spl token account
```

```
///
```

In contrast, `new_base_crncy.rs` explicitly performs these validations:

```
if is_new_account(token_acc) {
    if !is_new_account(program_acc) {
        bail!(InvalidNewAccount { ... });
    }
    if !program_acc.is_signer {
        return Err(drv_err!(MustBeSigner { ... }));
    }
    // ...
}
```

Note: similar works for `new_root_account`, do we also need to check the signer for that?

```
///
/// [*typo error in variables*](#typo-error-in-variables) - Deriverse Program Account `[SPL]` - Spl
↪  token account
///
```

Impact:

- Inconsistent error handling: different validation patterns across similar instructions, causing failures occur during CPI calls rather than early validation

Recommended Mitigation: Add explicit validation checks in new_instrument.rs when `is_new_account(asset_token_acc)` is true, matching the pattern in `new_base_crncy.rs`

Deriverse: Fixed in commit 96f4e923.

Cyfrin: Verified.

7.3.10 Referrer cannot be set after account creation

Description: The referral system has a critical limitation, users can only set a referrer during their **first deposit call** when creating a new account. There is no function to set or update a referrer after account creation, which creates several problems:

1. **First user cannot refer anyone**: The first user who creates an account cannot refer anyone because:
    - They can create referral links via `new_ref_link`, but
    - They cannot set a referrer during their own first deposit because there is no other user
    - There is no function to set a referrer after the first deposit
2. **Users who missed setting a referrer cannot set one later**: Users who did not provide a `ref_id` during their first deposit cannot set a referrer in subsequent deposits because the referrer setting logic only executes during account creation.

Impact:

1. **First User Problem**: The very first user to create an account cannot refer anyone, even though they can create referral links. This breaks the referral program for early adopter.
2. **Permanent Limitation**: Users who forgot to include a `ref_id` during their first deposit are permanently locked out of the referral program.

Recommended Mitigation: Create a new instruction `set_referrer` that allows users to set a referrer after account creation (with appropriate restrictions, e.g., only if `ref_address` is currently unset).

Deriverse: Fixed in commit 993cf7.

Cyfrin: Verified.

7.3.11 Inconsistent Price Calculation for Fee in Spot LP Trading

Description: The fee calculation in the `spot_lp` instruction uses `last_px` directly without applying the same price constraints (`best_bid, best_ask`) and unit conversion (`RDF`) that are used when adding liquidity. This inconsistency can lead to incorrect fee calculations when `last_px` falls outside the valid market price range.

In `src/program/processor/spot_lp.rs`, there are two different price calculation approaches:

1. When adding liquidity:

```
let px_f64 = instr_state
    .header
    .last_px
    .max(instr_state.header.best_bid)
    .min(instr_state.header.best_ask) as f64
    * RDF;
```

This calculation constrains the price to the range `[best_bid, best_ask]`

2. When calculating fees

```
fees = 1
    + ((instr_state.header.last_px as f64
        / get_dec_factor(instr_state.header.asset_token_decs_count) as f64)
        as i64)
        .max(1);
```

Uses `last_px` directly without price range constraints.

Impact: If `last_px` is outside `[best_bid, best_ask]` (e.g., due to market volatility or order book changes), fees may be calculated using an invalid price, leading to overcharging or undercharging users.

Recommended Mitigation: Update the fee calculation to use the same price logic as liquidity addition:

Deriverse: Fixed in commit 40e36f70.

Cyfrin: Verified.

7.3.12 `get_place_buy_price` function does not currently support the maximum `MAX_SUPPLY` value.

Description: Whenever a user buys or sells a seat, we calculate the seat price at that moment using the difference between get_reserve from the new/current position and the old position.

```
    pub fn get_place_buy_price(supply: u32, dec_factor: u32) -> Result<i64, DeriverseError> {
        let df = get_dec_factor(dec_factor);
        Ok(get_reserve(supply + 1, df)? - get_reserve(supply, df)?)
    }

    pub fn get_place_sell_price(supply: u32, dec_factor: u32) -> Result<i64, DeriverseError> {
        let df = get_dec_factor(dec_factor);
        Ok(get_reserve(supply, df)? - get_reserve(supply - 1, df)?)
    }
```

In `get_reserve`, we calculate `difference_to_max`, which can become zero when the user is at the 250,000th position. As a result, the `reserve` becomes infinite, causing `get_reserve` to return i64::MAX.

```
pub fn get_reserve(supply: u32, dec_factor: i64) -> Result<i64, DeriverseError> {
    let difference_to_max = MAX_SUPPLY - supply as i64;
    if difference_to_max < 0 {
        bail!(InvalidSupply {
            supply,
            supply_difference: difference_to_max as u32
        });
    }
```

```
    let reserve = MAX_SUPPLY as f64 * INIT_SEAT_PRICE * supply as f64 / difference_to_max as f64;

    return Ok((reserve * dec_factor as f64) as i64);
}
```

Impact: A user buying at the 250,000th position will end up paying significantly more than a user buying at the 249,999th position, because `get_reserve` for supply + 1 returns i64::MAX.

Recommended Mitigation: When calculating `difference_to_max` for the last user, we should add +1 to ensure that the reserve does not become infinite.

Deriverse: Fixed in commit c8d26d.

Cyfrin: Verified.

7.3.13 `get_current_leverage` calculates leverage incorrectly for long positions

Description: When calculating the margin, we use `(-(info.funds.min(perp_value as i64))).max(0)`, In a long position, the user's funds become negative, while in a short position, the `perp_value` becomes negative.

However, when computing leverage, we use `position_size / value`. For long positions, the user's funds effectively equal the initial funds used to open the position minus the position size, which leads to an incorrect leverage calculation.

Scenerio: A user with 10,000 USDC in funds who wants to open a 10× long position on Bitcoin, priced at 100,000 USDC. The user uses the 10,000 USDC as margin and enters a long position of 1 BTC. After the purchase, the user's funds become –90,000 USDC, and their perpetual position equals 1 BTC.

However, when calculating the current leverage, the function returns 9× instead of 10× because it computes the leverage as follows: `max(-min(-90000, 100000), 0) / 10,000`

Impact: This does not have any practical impact because the old and new leverage values are compared in `check_client_leverage_shift`. The logic still holds, as an increase in leverage always results in a higher new leverage, so no issues arise currently.

Recommended Mitigation: We should use `perp_value` when calculating the current leverage.

Deriverse: Fixed in commit e4c0a6.

Cyfrin: Verified.

7.3.14 Margin call limit bypass

Description: The limit check for margin calls can be bypassed, allowing the total number of margin calls to exceed `MAX_MARGIN_CALL_TRADES`.

The issue occurs because after the first `check_{short,long}_margin_call`, the trade counter is reset in next `check_{short,long}_margin_call` tha allow to execute additional trades beyond the intended limit.

Example: `MAX_MARGIN_CALL_TRADES` is set to 10. `check_short_margin_call` returns 9 (meaning 9 margin calls were executed). Since 9 < 10, `check_long_margin_call` is then called. `check_long_margin_call` executes an additional 10 margin calls.

As a result, a total of 9 + 10 = 19 margin calls are executed, which exceeds the intended limit of 10.

Impact: The `MAX_MARGIN_CALL_TRADES` limit can be exceeded, which may also lead to more execution costs.

Recommended Mitigation: Track the cumulative margin-call count across both functions and compare it to MAX_-MARGIN_CALL_TRADES before executing any additional margin calls.

Deriverse: Fixed in commit 4f7bc8.

Cyfrin: Verified.

7.3.15 Missing Fixing Window Data Accumulation After Daily Reset in `drv_update`

Description: When `drv_update` resets fixing data for a new day, the current `last_asset_tokens` and `last_crncy_tokens` are discarded instead of being accumulated into the new day's fixing data, leading to incomplete fixing price calculations.

In `src/state/instrument.rs`, the `drv_update` function handles daily reset logic:

```
if current_date > last_fixing_date {
    // Calculate new fixing price based on previous day's data
    let new_fixing_px: i64 = if self.header.fixing_crncy_tokens
        > get_dec_factor(self.header.crncy_token_decs_count)
    {
        (self.header.fixing_crncy_tokens as f64 * self.header.dec_factor as f64
            / self.header.fixing_asset_tokens as f64) as i64
    } else {
        prev_px
    };
    self.header.fixing_asset_tokens = 0;  // Reset
    self.header.fixing_crncy_tokens = 0;  // Reset
    self.header.fixing_time = time;
    // ... update variance and fixing_px
} else {
    // Only accumulate if within fixing window
    let sec = time % DAY;
    if last_asset_tokens > 0 && (SETTLEMENT - FIXING_DURATION..SETTLEMENT).contains(&sec) {
        self.header.fixing_asset_tokens += last_asset_tokens;
        self.header.fixing_crncy_tokens += last_crncy_tokens;
    }
}
```

When entering a new day (`current_date > last_fixing_date`), the code resets `fixing_asset_tokens` and `fixing_crncy_tokens` to `0` However, if the current time time is still within the fixing window (`SETTLEMENT - FIXING_DURATION..SETTLEMENT`), the current `last_asset_tokens` and `last_crncy_tokens` should be accumulated into the new day's fixing data. **Currently, these values are discarded after reset, and the code does not check if the current time is within the fixing window**

Impact: Trading volume that occurs immediately after the daily reset but within the fixing window is not included in the fixing price calculation.

Recommended Mitigation: After resetting the fixing data, check if the current time is within the fixing window and accumulate the current data if applicable.

Deriverse: Fixed in commit d17502c5.

Cyfrin: Verified.

7.3.16 `get_by_tag` tries to access out of bound index

Description: The get_by_tag contains an off-by-one error in its loop condition. The function iterates using while i <= continer.candles.len() instead of the correct while i < continer.candles.len(), causing an out-of-bounds array access when iterating past the last valid index.

```
pub const fn get_by_tag<const TAG: u32>(
    continer: CandleRegister,
) -> Result<CandleParams, DeriverseError> {
    let mut i = 0;

    while i <= continer.candles.len() {  // BUG: should be `<` not `<=`
        if continer.candles[i].tag == TAG {  // Out-of-bounds when i == len
            return Ok(continer.candles[i]);
        }
        i += 1;
```

```
    }
    // ...
}
```

This function is called throughout the codebase in different candle-related operations & all paths are affected. The bug may not manifest during normal operations if the TAG is found early in the array, It will panic when i == len() due to out-of-bounds indexing rather than returning normal error

```
    Err(DeriverseError {
        error: DeriverseErrorKind::CandleWasNotFound { tag: TAG },
        location: ErrorLocation {
            file: file!(),
            line: line!(),
        },
    })
```

Impact: When the requested TAG is not found before the loop exhausts, or when i reaches continer.candles.len(), the code attempts to access memory beyond the array bounds and hence will cause program to panic rather than throwing proper error.

Recommended Mitigation:

```
pub const fn get_by_tag<const TAG: u32>(
    continer: CandleRegister,
) -> Result<CandleParams, DeriverseError> {
    let mut i = 0;

    while i < continer.candles.len() {  // Fixed: use `<` instead of `<=`
        if continer.candles[i].tag == TAG {
            return Ok(continer.candles[i]);
        }
        i += 1;
    }

    Err(DeriverseError {
        error: DeriverseErrorKind::CandleWasNotFound { tag: TAG },
        location: ErrorLocation {
            file: file!(),
            line: line!(),
        },
    })
}
```

Deriverse: Fixed in commit: 4e88698

Cyfrin: Verified.

7.3.17 Incorrect Amount Logged in `PerpWithdrawReport`

Description: The `perp_withdraw` function logs the requested amount (`data.amount`) instead of the actual withdrawn amount (`amount`) in the `PerpWithdrawReport` event. This creates a discrepancy between the logged amount and the actual funds transferred, particularly when `data.amount == 0` (withdraw all available funds) or when margin call restrictions apply.

In the `perp_withdraw` function, the actual withdrawal amount is calculated based on various conditions:

```
let amount = if margin_call {
    let margin_call_funds =
        funds.min(engine.get_avail_funds(client_state.temp_client_id, true)?);
    if margin_call_funds <= 0 {
        bail!(ImpossibleToWithdrawFundsDuringMarginCall);
    }
```

```
    if data.amount == 0 {
        margin_call_funds  // Actual amount may differ from data.amount
    } else {
        // ... validation logic ...
        data.amount
    }
} else if data.amount == 0 {
    funds  // Actual amount may differ from data.amount
} else {
    // ... validation logic ...
    data.amount
};
```

The actual funds are transferred using the calculated `amount` variable :

```
client_state.add_crncy_tokens(amount)?;
client_state
    .perp_info()?
    .sub_funds(amount)
    .map_err(|err| drv_err!(err))?;
```

However, the log entry records `data.amount` instead of the actual `amount`:

```
solana_program::log::sol_log_data(&[bytemuck::bytes_of::<PerpWithdrawReport>(
    &PerpWithdrawReport {
        tag: log_type::PERP_WITHDRAW,
        client_id: client_state.id,
        instr_id: data.instr_id,
        amount: data.amount,  //  Should be: amount
        time: ctx.time,
        ..PerpWithdrawReport::zeroed()
    },
)]);
```

Impact: Logs do not reflect actual withdrawals, complicating accounting.

Recommended Mitigation: Update the log entry to record the actual withdrawn `amount`.

Deriverse: Fixed in commit 2a0fe33.

Cyfrin: Verified.

7.3.18 Silent Failure in `voting_reset`

Description: The `voting_reset` function silently ignores failures when attempting to upgrade the community account header for writing. If the `community_acc` account is not marked as writable, the `upgrade()` call fails, but the function still returns `Ok(())`, giving the caller a false impression that the voting parameters were successfully reset when they were not.

The voting_reset function uses if let `Ok(header) = community_state.header.upgrade()` to conditionally update the community account header. However, this pattern silently returns error that occurs when the account is not marked as writable.

```
    if let Ok(header) = community_state.header.upgrade() {
        header.spot_fee_rate = START_SPOT_FEE_RATE;
        header.perp_fee_rate = START_PERP_FEE_RATE;
        header.max_discount = START_MAX_DISCOUNT;
        header.margin_call_penalty_rate = START_MARGIN_CALL_PENALTY_RATE;
        header.fees_prepayment_for_max_discount = START_FEES_PREPAYMENT_FOR_MAX_DISCOUNT;
        header.spot_pool_ratio = START_SPOT_POOL_RATIO;
    }
```

Throughout the codebase, all other functions that use `upgrade` properly propagate errors using the `?` operator:

```
    if let Some(ref mut header) = client_state.header {
        header.upgrade()?.points = 0;
        header.upgrade()?.mask &= 0xFFFFFFFFFFFFFF;
    }
```

```
        if community_state.header.voting_counter == 0 {
            community_state.header.upgrade()?.voting_counter += 1;
        }
```

Impact: If the `community_acc` account is incorrectly not marked as writable (due to a bug in the caller or a configuration error), the function will appear to succeed but no state updates will occur.

Recommended Mitigation: Change the code to properly propagate the error when `upgrade()` fails.

Deriverse: Fixed in commit 9ef2d7.

Cyfrin: Verified.

7.4 Informational

7.4.1 Incorrect Price Validation When Creating `NewInstrumentData` Struct during `NewInstrumentInstruction` instruction

Description: The current validation logic when creating `NewInstrumentData` using `new` method during `NewInstrumentInstruction` instruction uses an exclusive upper bound when checking the price field. As a result, a price equal to MAX_PRICE is incorrectly rejected even though it should be considered valid.

```
    fn new(instruction_data: &[u8], tokens_count: u32) -> Result<&Self, DeriverseError> {
        let data = bytemuck::try_from_bytes::<Self>(instruction_data)
            .map_err(|_| drv_err!(InvalidClientDataFormat))?;

        if data.crncy_token_id >= tokens_count {
            bail!(InvalidTokenId {
                id: data.crncy_token_id,
            })
        }

        if !(MIN_PRICE..MAX_PRICE).contains(&data.price) {                  //<- HERE
            bail!(InvalidPrice {
                price: data.price,
                min_price: MIN_PRICE,
                max_price: MAX_PRICE,
            })
        }

        return Ok(data);
    }
```

In Rust, the syntax a..b defines a range that excludes the upper bound(b), whereas a..=b defines an inclusive range that allows b as a valid value.

Impact: This bug prevents legitimate instruments with a price equal to MAX_PRICE from being created.

Recommended Mitigation: To fix this issue, the validation should use an inclusive range (a..=b) so that prices equal to MAX_PRICE pass the check.

```
    fn new(instruction_data: &[u8], tokens_count: u32) -> Result<&Self, DeriverseError> {
        let data = bytemuck::try_from_bytes::<Self>(instruction_data)
            .map_err(|_| drv_err!(InvalidClientDataFormat))?;

        if data.crncy_token_id >= tokens_count {
            bail!(InvalidTokenId {
                id: data.crncy_token_id,
            })
        }

        if !(MIN_PRICE..=MAX_PRICE).contains(&data.price) {                  //<- HERE
            bail!(InvalidPrice {
                price: data.price,
                min_price: MIN_PRICE,
                max_price: MAX_PRICE,
            })
        }

        return Ok(data);
    }
```

Deriverse: Fixed in commit aa6136.

Cyfrin: Verified.

7.4.2 typo error in variables

Description: The variable `mint_token_prgoram_version` contains a typo: "prgoram" should be "program". This typo is repeated in the error message.

```
    let data = NewInstrumentData::new(instruction_data, root_state.tokens_count)?;
    let mint_token_prgoram_version = TokenProgram::new(asset_mint.owner)?;

    let token_program_version = TokenProgram::new(token_program.key)?;

    if token_program_version != mint_token_prgoram_version {
        bail!(DeriverseErrorKind::InvalidMintProgramId {
            expected: token_program_version,
            actual: mint_token_prgoram_version,
            mint_address: *asset_mint.key,
        });
    }
```

Impact: This does not affect functionality, it reduces code readability and consistency.

Recommended Mitigation: Rename the variable to mint_token_program_version:

```
-    let mint_token_prgoram_version = TokenProgram::new(asset_mint.owner)?;
+    let mint_token_program_version = TokenProgram::new(asset_mint.owner)?;

    let token_program_version = TokenProgram::new(token_program.key)?;

-    if token_program_version != mint_token_prgoram_version {
+    if token_program_version != mint_token_program_version {
        bail!(DeriverseErrorKind::InvalidMintProgramId {
            expected: token_program_version,
-            actual: mint_token_prgoram_version,
+            actual: mint_token_program_version,
            mint_address: *asset_mint.key,
        });
    }
```

Deriverse: Fixed in commit a3c75ae8.

Cyfrin: Verified.

7.4.3 Transfers are noop when `lamports_diff` is zero

Description: The codebase contains multiple instances where lamports transfers are performed for rellocating space and size without checking if the transfer amount `lamports_diff` is greater than zero. To evaluate `lamports_diff` we are using `saturating_sub` which limits subtraction in cases when there are chances of underflow.

```
// here if the account already had enough lamports to cover up its rent, the `lamports_diff` would come
↪  out as 0.
let lamports_diff = new_minimum_balance.saturating_sub(account.lamports());
invoke(
    &system_instruction::transfer(signer.key, account.key, lamports_diff),
    &[signer.clone(), account.clone(), system_program.clone()],
)?;
```

When `lamports_diff` is zero, the code still makes unnecessary Cpi calls to the System Program's transfer instruction, which wastes computational units and is no op.

Here are the files where its present: `perp_engine.rs`, `new_base_crncy.rs`, `new_operator.rs`, `engine.rs`, `candles.rs`, `client_community.rs`, `client_primary.rs`

Impact: Unnecessary cu lost in cases when `lamports_diff==0`.

Recommended Mitigation: Perform transfer only if `lamports_diff` > 0.

```
let lamports_diff = new_minimum_balance.saturating_sub(account.lamports());
if lamports_diff > 0 {
    invoke(
        &system_instruction::transfer(signer.key, account.key, lamports_diff),
        &[signer.clone(), account.clone(), system_program.clone()],
    )?;
}
```

Deriverse: Fixed in commit: 7091f4

Cyfrin: Verified.

7.4.4 Entrypoint panics on empty `instruction_data`

Description: The issue originates from a missing bounds check for the `instruction_data` array before accessing its first element during opcode dispatch.

```
entrypoint!(process_instruction);

pub fn process_instruction(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    instruction_data: &[u8],
) -> ProgramResult {
    match instruction_data[0] {
        NewHolderInstruction::INSTRUCTION_NUMBER => new_holder_account(program_id, accounts)?,
        NewOperatorInstruction::INSTRUCTION_NUMBER => {
            new_operator(program_id, accounts, instruction_data)?
        }
```

Impact: This can result in an out-of-bounds read and program panic when `instruction_data` is empty instead of exiting gracefully with correct error message.

Recommended Mitigation: To mitigate this vulnerability, the program should validate the length of `instruction_data` before accessing entries, and handle the error gracefully instead of resorting to a crash.

Deriverse: Fixed in commit 8a2bd16.

Cyfrin: Verified.

7.4.5 Unnecessary Lamports Transfer Without Checking Existing Balance

Description: The `new_private_client()` function transfers the full `minimum_balance` amount to the `private_clients_acc` account without first checking if the account already has sufficient lamports. This results in unnecessary transfers even when the account already contains enough funds to cover the rent requirement.

```
// src/program/processor/new_private_client.rs:163-174
let rent = &Rent::default();
let lamports = rent.minimum_balance(std::mem::size_of::<PrivateClient>());

invoke(
    &system_instruction::transfer(admin.key, private_clients_acc.key, lamports),
    &[
        admin.clone(),
        private_clients_acc.clone(),
        system_program.clone(),
    ],
)
.map_err(|err| drv_err!(err.into()))?;
```

This approach differs from the pattern used consistently throughout the codebase in similar scenarios, which calculate and transfer only the difference needed:

Impact: Potential Over-payment: If the account already contains more than the minimum required balance, the function still attempts to transfer the full minimum balance amount.

Recommended Mitigation: Calculate and transfer only the difference needed, matching the pattern used in other functions like `new_operator()`.

Deriverse: Fixed in commit 7091f4.

Cyfrin: Verified.

7.4.6 Incomplete Balance Check Missing Transaction Fee and Reserve Balance

Description: The `create_client_account()` function checks if the wallet has sufficient balance to create two accounts, but it does not account for transaction fees or reserve balance for the wallet itself. This can lead to transaction failures or leave the wallet account in an unusable state after the operation.

When creating client accounts, the code only verifies that the wallet balance covers the rent-exempt minimums for the two new accounts:

```
    if balance < client_primary_lamports + client_community_lamports {
        bail!(InsufficientFunds);
    }
```

After the two `create_account` instructions execute, the wallet transfers:

- `client_primary_lamports` to the primary account
- `client_community_lamports` to the community account

If the wallet's initial balance was exactly `client_primary_lamports + client_community_lamports`, the wallet would be left with:

- 0 lamports (or very close to 0)
- Unable to pay transaction fees
- Potentially unable to perform subsequent operations

Impact: If a wallet has exactly `client_primary_lamports + client_community_lamports` balance, the transaction may fail during execution.

Recommended Mitigation: Reserve a small, predefined amount (e.g., a few thousand lamports) on top of the required rent-exempt minimums.

```
const WALLET_RESERVE_LAMPORTS: u64 = 5_000_000; // example reserve
if balance < client_primary_lamports + client_community_lamports + WALLET_RESERVE_LAMPORTS {
    bail!(InsufficientFunds);
}
```

Deriverse: Fixed in commit 548040f.

Cyfrin: Verified.

7.4.7 Eligible intruments may not be propagated

Description: The function `dividends_allocation` currently throws error when encountering an instrument whose `distrib_time` has not yet passed the 1-hour threshold:

```
        } else {
            //@audit instead of returning we should have continued to another instr
            bail!(TooEarlyToDistribFunds {
                limit_time: instr_state.header.distrib_time,
                current_time: time,
```

```
            });
        }
```

This immediately terminates the entire dividends allocation process, even if subsequent instruments in the same transaction are eligible for distribution. As a result, valid instruments are skipped.

Impact: Eligible instruments in the same batch are not processed.

Recommended Mitigation: Instead of bailing, we should continue to the next instrument in the loop:

```
if time > instr_state.header.distrib_time + HOUR {
    // distribute funds
} else {
    // skip this instrument, continue to next
    continue;
}
```

Deriverse Fixed in commit ca593e2.

Cyfrin: Verified.

7.4.8 Insufficient Self-Referral Protection Still Allows Multiple Account Self-Referral

Description: The referral validation in the `deposit()` function only prevents self-referral by checking if the referral account address matches the client's account address. However, this allows users to create multiple accounts (with different wallets they control) and self-refer through different accounts.

```
if *ref_acc.key == *client_primary_acc.key {
    bail!(InvalidRefAddress {
        expected_address: *client_primary_acc.key,
        actual_address: *ref_acc.key,
    });
}
```

Scenario:

1. User controls `wallet1` and `wallet2`
2. User creates `account A` with `wallet1` and generates referral link
3. User creates `account B` with `wallet2`
4. User uses account A's referral link when depositing with account B (wallet2)
5. User successfully self-refers through different wallets

Impact: The referral program's intended purpose of user acquisition is undermined.

Recommended Mitigation: Preventing multi-wallet self-referral (where users control multiple wallets) is a fundamental blockchain limitation that may require off-chain solutions like KYC or centralized activities which may not be applicable here.

Deriverse: Acknowledged.

7.4.9 Attacker can exploit Dividend allocation by depositing large mounts of DRVS Tokens

Description: Dividends can be allocated once every hour, and the allocation process can be triggered by anyone. The distribution of fees is determined based on the amount of DRVS tokens each user has deposited at the time of allocation.

However, since the allocation timing is predictable, an attacker can exploit this mechanism by depositing a large amount of DRVS tokens just before the allocation occurs. This allows the attacker to unfairly receive a portion of the dividends that should rightfully belong to long-term depositors. Immediately after receiving their share of the

dividends, the attacker can claim and then withdraw their DRVS tokens, effectively capturing profits without making any meaningful contribution to the system.

This behavior can be further optimized. For example, an attacker could develop a smart contract that performs all the steps deposit, trigger the allocation, and withdraw within a single transaction.

An attacker doesn't need to hold DRVS at all in wallet — they can simply use a flash loan to temporarily acquire the dividend, claim the dividend. Similarly, they can borrow DRVS, claim the rewards, and return the borrowed amount, or just swap another asset into DRVS to claim the dividend and immediately swap back.

Impact: This allows repeated exploitation of the allocation mechanism, letting attackers drain dividends from honest users and reducing the fairness of the distribution process.

Recommended Mitigation: Mitigations are:

1. Introduce a withdrawal delay — users should only be allowed to withdraw their DRVS tokens after a certain period has passed since their last deposit.
2. Reduce the allocation interval — allow allocations to occur more frequently(e.g., every few seconds) to make it harder to exploit.

Deriverse: Acknowledged; we think that extracting profit from this approach is unlikely and very expensive.

7.4.10 Forced Oldest-Order Eviction Enables Griefing

Description: This issue is theoretically possible, but the exploitation conditions are quite strict, so I marked it as INFO.

When a side of the spot order book exceeds `MAX_ORDERS`, the matching engine unconditionally cancels the globally oldest order on that side. Because `MAX_ORDERS` is shared by all traders, a malicious participant can deliberately fill the book with tiny orders, repeatedly trigger the eviction path, and force legitimate users' resting/large orders to be cancelled.

`add_order` grows the order book and, after inserting the new order, checks whether the per-side total exceeds MAX_ORDERS. If so, it removes the oldest order, without regard to ownership and the order size.

```
        } else if self.orders_count(side) > MAX_ORDERS {
            let oldest_node = self.find_oldest_order_node(side);
            let oldest_order = self.get_order_ptr(oldest_node.link(), side);
            self.erase_client_order(oldest_order, oldest_node, true, side)?;
            ...
```

`MAX_ORDERS` is a global limit (14,334) shared by all users of the instrument.

```
pub mod spot {
    pub const MAX_LINES: usize = 2048;
    pub const MAX_ORDERS: u32 = (4 * 64 * 64 - MAX_LINES) as u32 - 2;
```

An attacker can create many minimum-quantity orders (funds are returned when their orders are eventually evicted), drive `orders_count(side)` above `MAX_ORDERS`, and delete the oldest resting order. The cost is limited to transaction fees plus temporarily locking collateral for the attacker's own active orders.

Impact: Other traders' limit orders can be griefed off the book at will regardless of the order size, degrading market integrity, denying service, and allowing the attacker to control displayed liquidity.

Recommended Mitigation: Prevent a single participant from consuming the entire order quota. Options include:

- Enforce a per-client cap on orders numbers.

Deriverse: Fixed in commit ed3b97ec.

Cyfrin: Verified.

7.4.11 User cannot claim dividends after withdrawing their DRVS tokens

Description: In `dividends_claim`, the function checks that `client_community_state.header.drvs_tokens` is greater than zero for the user to be eligible to claim their dividend tokens.

```
    if client_community_state.header.drvs_tokens > 0 {
        let clock = Clock::get().map_err(|err| drv_err!(err.into()))?;
        let slot = clock.slot as u32;
        for (d, b) in client_community_state
            .data
            .iter_mut()
            .zip(community_state.base_crncy.iter_mut())
        {
            client_state.resolve(AssetType::Token, b.crncy_token_id, TokenType::Asset, true)?;
            let amount = ((((b.rate - d.dividends_rate)
                * client_community_state.header.drvs_tokens as f64)
                as i64)
                + d.dividends_value)
                .min(b.funds)
                .max(0);
            client_state.add_asset_tokens(amount)?;
            b.funds -= amount;
            d.dividends_rate = b.rate;
            d.dividends_value = 0;
            solana_program::log::sol_log_data(&[bytemuck::bytes_of::<EarningsReport>(
                &EarningsReport {
                    tag: log_type::EARNINGS,
                    client_id: client_state.id,
                    amount,
                    token_id: b.crncy_token_id,
                    time: clock.unix_timestamp as u32,
                    ..EarningsReport::zeroed()
                },
            )]);
        }
        client_state.header.try_upgrade()?.slot = slot;
    }
```

Our current design allows a user to withdraw their DRVS tokens without claiming dividends. This means that if the user sells their tokens after withdrawing, they will not be able to claim dividends until they deposit some DRVS tokens again.

Impact: User will not be able to claim their `dividends_value` even if it is greater than zero, unless their DRVS deposit is also greater than zero.

Recommended Mitigation: Allow claiming the dividend value even when `client_community_state.header.drvs_tokens` is zero.

Deriverse: Fixed in commit 4dba9d.

Cyfrin: Verified.

7.4.12 Missing Validation for `order_type` in NewSpotOrderData

Description: The `NewSpotOrderData::new` validation function does not verify that `order_type` is a valid value for spot orders. While spot orders should only accept `Limit (0)` and `Market (1)`, the validation allows `MarginCall (2)` and `ForcedClose (3)` to pass through. These invalid values are then incorrectly processed as Market orders, leading to data inconsistency in logs and potential confusion.

In `src/program/instruction_data.rs`, the `NewSpotOrderData::new` function validates:

- ‘instr_id range
- price validity (only when `order_type == 0`)

- `amount` range However, it does not validate that `order_type` is within the allowed range for spot orders (0 or 1).

```
    fn new(instruction_data: &[u8], instr_count: u32) -> Result<&Self, DeriverseError> {
        let data = bytemuck::try_from_bytes::<Self>(instruction_data)
            .map_err(|_| drv_err!(InvalidClientDataFormat))?;

        if data.instr_id >= instr_count {
            bail!(InvalidInstrId { id: *data.instr_id })
        }
        if data.order_type == 0 && !(MIN_PRICE..=MAX_PRICE).contains(&data.price) {
            bail!(InvalidPrice {
                price: data.price,
                min_price: MIN_PRICE,
                max_price: MAX_PRICE,
            })
        }
        if !(1..=SPOT_MAX_AMOUNT).contains(&data.amount) {
            bail!(InvalidQuantity {
                value: data.amount,
                min_value: 1,
                max_value: SPOT_MAX_AMOUNT,
            })
        }

        return Ok(data);
    }
```

The `OrderType` enum defines:

```
pub enum OrderType {
    Limit = 0,
    Market = 1,
    MarginCall = 2,     // Only for perp orders
    ForcedClose = 3,    // Not used in codebase
}
```

In `src/program/processor/new_spot_order.rs`, the code only explicitly handles Limit:

```
if data.order_type == OrderType::Limit as u8 {
    data.price
} else if buy {
    mark_px + (mark_px >> 3)  // All other values treated as Market
} else {
    mark_px - (mark_px >> 3)
}
```

Then the `type` is being emitted in logs:

```
    solana_program::log::sol_log_data(&[bytemuck::bytes_of::<SpotPlaceOrderReport>(
        &SpotPlaceOrderReport {
            tag: log_type::SPOT_PLACE_ORDER,
            order_type: data.order_type,
            side: if buy { 0 } else { 1 },
            ioc: data.ioc,
            client_id: client_state.id,
            order_id: engine.state.header.counter,
            instr_id: data.instr_id,
            qty: data.amount,
            price,
            time: ctx.time,
        },
```

```
    )]);
```

Impact:

- Data Inconsistency: Logs will contain incorrect order_type values that don't match the actual order behavior
- Input Validation Gap: Missing validation allows invalid enum values to be accepted

Recommended Mitigation: Add validation in `NewSpotOrderData::new` to ensure `order_type` is only 0 (Limit) or 1 (Market) for spot orders:

Example:

```
        // Add this validation
        if data.order_type > OrderType::Market as u8 {
            bail!(InvalidOrderType {
                order_type: data.order_type,
                allowed_types: vec![OrderType::Limit as u8, OrderType::Market as u8],
            })
        }
```

Deriverse: Fixed in commit 53000a3.

Cyfrin: Verified.

7.4.13 Rounding Error Accumulation in Partial Order Fills Leads to Unfair Cost Distribution

Note: This finding was submitted after careful consideration. The impact is negligible in practice, as it only manifests when an order is partially filled multiple times, causing tiny rounding errors to accumulate. However, it is still worth documenting, as it represents a systematic bias that could potentially be addressed in documentation or future improvements.

Description: When an order is partially filled multiple times, rounding errors from the `trade_sum` function accumulate in the order's remaining `sum` field. The final fill receives the accumulated rounding errors, causing the last trader to pay more than they should based on the actual traded quantity.

The vulnerability exists in the `fill` function of the spot trading engine. When an order is created, the total currency value is calculated using `trade_sum(price, qty)` and stored in `order.sum`:

```
// Line 643: Order creation
let order_sum = self.trade_sum(price, qty)?;
// ...
sum: order_sum,  // Stored in order
```

The `trade_sum` function performs floating-point multiplication with `rdf` (rounding division factor) and truncates to `i64`:

```
// Lines 157-158: rdf calculation
let df = state.header.dec_factor as f64;
let rdf = 1f64 / df;  // rdf is typically a decimal (e.g., 0.1, 0.01, 0.001)

// Lines 277-284: trade_sum function
fn trade_sum(&self, a: i64, b: i64) -> Result<i64, DeriverseError> {
    let sum = (a as f64 * b as f64) * self.rdf;  // Floating-point multiplication with rdf
    // ...
    Ok(sum as i64)  // Truncation from f64 to i64 causes rounding errors
}
```

Root Cause of Rounding Errors: The rounding errors occur because:

1. `rdf = 1.0 / dec_factor`, where `dec_factor = 10^n` (a power of 10 based on token decimal differences)
2. `rdf` could be a decimal fraction (e.g., 0.1, 0.01, 0.001), requiring floating-point arithmetic

3. The multiplication `(a * b) * rdf` in floating-point can produce results that are not exactly representable as integers
4. The conversion from `f64` to `i64` truncates the fractional part, causing precision loss
5. Without `rdf` (i.e., if `rdf = 1.0`), the calculation would be exact integer arithmetic with no rounding errors

When an order is partially filled (line 1187), the code recalculates the currency value using `trade_sum`:

```
// Lines 1178-1188
let (traded_qty, traded_crncy, last) = if order_qty <= *remaining_qty {
    (order_qty, order.sum(), false)  // Full fill uses stored sum
} else {
    (*remaining_qty, self.trade_sum(*remaining_qty, px)?, true)  // Partial fill recalculates
};
```

Then, when `last = true` (partial fill), the code decrements the order's sum:

```
// Lines 1265-1267
if last {
    order.decr_qty(traded_qty).map_err(|err| drv_err!(err))?;
    order.decr_sum(traded_crncy).map_err(|err| drv_err!(err))?;  // Subtracts recalculated value
    // ...
}
```

The Problem:

1. The presence of `rdf` (a decimal fraction) in `trade_sum` causes floating-point precision issues when converting to `i64`.
2. Each partial fill recalculates `trade_sum(remaining_qty, px)`, which introduces rounding errors due to `f64→i64` truncation in the presence of `rdf`.
3. The recalculated value is subtracted from `order.sum`, causing rounding errors to accumulate in the remaining `sum`.
4. When the order is fully filled, the remaining `order.sum` may not equal `trade_sum(remaining_qty, px)` but instead contains all accumulated rounding errors from previous partial fills.

Example Scenario: Assume `rdf = 0.1` (i.e., `dec_factor = 10`) and price = 99:

- Order: qty=100, price=99, rdf=0.1
- Original sum: `trade_sum(100, 99) = (100 * 99) * 0.1 = 990.0` → 990 (exact)
- First partial fill (33 qty): `traded_crncy = trade_sum(33, 99) = (33 * 99) * 0.1 = 326.7` → 326 (truncated, loses 0.7)
- Remaining sum: `990 - 326 = 664` (should be 663.3, but stored as integer)
- Second partial fill (33 qty): `traded_crncy = trade_sum(33, 99) = 326.7` → 326 (truncated, loses 0.7)
- Remaining sum: `664 - 326 = 338` (should be 337.3, but stored as integer)
- Final fill (34 qty):
  - Expected: `trade_sum(34, 99) = (34 * 99) * 0.1 = 336.6` → 336
  - Actual received: `order.sum = 338` (contains accumulated rounding errors: 0.7 + 0.7 = 1.4)
  - The final trader receives 338 instead of 336, paying 2 more than they should

Impact: Unfair Cost Distribution: The last trader to fill a partially-filled order bears the cost of all accumulated rounding errors from previous partial fills.

Recommended Mitigation: This may be a design choice, and leaving it as-is is acceptable. However, it should be clearly documented that rounding errors are unavoidable when using floating-point arithmetic with `rdf`, and the current implementation accumulates these errors to the final partial fill, Given the negligible impact and the fact that

rounding errors are unavoidable (the question is only who bears them), **the current design choice is acceptable as long as it is properly documented.**

Deriverse: Fixed in commit 058c856.

Cyfrin: Verified. Added documents.

7.4.14 Returning true when the current time has reached the `expiration_time` in `is_vacant`

Description: `is_vacant` is used in `deposit` and `new_private_client`. In deposit function, we check whether the user is a private client and `is_vacant` is false, meaning the record has not expired. In `new_private_client`, we check whether `is_vacant` returns true, meaning the record is vacant or has expired.

```
    pub fn is_vacant(&self, current_time: u32) -> bool {
        self.creation_time == 0 || current_time > self.expiration_time
    }
```

In `is_vacant`, we check whether the current time is greater than the expiration time. If the current time is equal to the expiration time, the function still returns false.

Impact: `is_vacant` allows an expired private client to deposit, and in `new_private_client` we do not update the record at the index even when the current time has reached the expiration.

Recommended Mitigation: `is_vacant` should return true when the current time is equal to the expiration time:

```
    pub fn is_vacant(&self, current_time: u32) -> bool {
        self.creation_time == 0 || current_time >= self.expiration_time
    }
```

Deriverse: Fixed in commit 408cd2.

Cyfrin: Verified.

7.4.15 Missing System Program Check in Instructions like `new_operator`(Inconsistency)

Description: Unlike many other handlers, `new_operator` does not verify that the `system_program` account equals `solana_program::system_program::ID` before using it in `realloc_account_data`. The call would eventually fail if a wrong account is supplied, but the error surfaces only after a CPI panic rather than as a clean, descriptive failure.

The instruction expects `system_program (accounts[3])` to be the Solana system program and later passes it into `realloc_account_data`, which performs a CPI to the system program.

```
    let system_program = next_account_info!(accounts_iter)?;
    check_holder_admin(admin)?;
    let mut holder_state = HolderState::new(holder_acc, program_id)?;

    let begin = std::mem::size_of::<HolderAccountHeader>()
        + (holder_state.header.operators_count as usize) * std::mem::size_of::<Operator>();

    let new_size = begin + std::mem::size_of::<Operator>();

    if holder_acc.data_len() < new_size {
        realloc_account_data(admin, holder_acc, system_program, new_size, None, true)
            .map_err(|err| drv_err!(err.into()))?;
    }
```

If a client provides some other executable account, the CPI will trap because the runtime detects the program-id mismatch. This isn't exploitable, but it differs from other processors (e.g., `new_instrument`) that proactively check `system_program.key == system_program::ID` and return a clear error before the CPI.

```
    if !system_program::check_id(system_program.key) {
        bail!(InvalidSystemProgramId {
            actual_address: *system_program.key,
```

```
        });
    }
```

Impact: Program will panic instead of returning a well-typed error.

Recommended Mitigation: Add an explicit check in `new_operator` (and any similar handlers) that `system_program.key == &solana_program::system_program::ID`

Deriverse: Fixed in commit d7b852b4.

Cyfrin: Verified.

7.4.16 Inefficient `free_index` Selection in Assets Array Causes Performance Degradation

Description: The `find_or_alloc_asset` function in `client_primary` always selects the last vacant slot (`asset_id == 0`) instead of the first available slot when allocating new assets. This causes unnecessary `O(n)` iterations, space fragmentation, and potential compute budget exhaustion as the assets array grows.

In `client_primary`, the loop logic is:

```
for (i, a) in self.assets.iter().enumerate() {
    let current_id = a.asset_id;
    if current_id == asset_id {
        asset_index = i;
        break;
    } else if current_id == 0 {
        free_index = i;  // Always overwrites with the last vacant slot
    }
}
```

`free_index` is continuously overwritten whenever a vacant slot (`asset_id == 0`) is found, meaning it will always contain the index of the last vacant slot in the array, not the first.

Example scenario:

- Array state: `[asset1, 0, asset2, 0, asset3]`
- When searching for a non-existent `asset4`
- The loop traverses all 5 elements
- `free_index` is set to index `1` (first 0), then overwritten to index `3` (last 0)
- The first vacant slot at index `1` remains unused

Impact: Later in the search, every lookup could requires full O(n) traversal even when vacant slots exist early in the array

Recommended Mitigation: `free_index` could only be overwritten when it is `NULL_INDEX`, thus we are always returning and using the first vacant slot.

Deriverse: Fixed in commit 30e062e.

Cyfrin: Verified.

7.4.17 Withdraw Instruction Trying to Create New Asset Records for Non-Existent Tokens Instead of Failing

Description: The `withdraw` instruction uses `alloc=true` when resolving the token asset, which tries the creation of a new asset record with zero balance if the token doesn't exist, instead of immediately failing with an appropriate error. While the transaction eventually fails, this behavior wastes account space and violates the semantic expectation that withdrawals should only operate on existing assets.

In `withdraw.rs`, the resolve function is called with `alloc=true`:

```
client_state.resolve(AssetType::Token, data.token_id, TokenType::Asset, true)?;
```

When `find_or_alloc_asset` is called with `alloc=true` and the asset doesn't exist in `client_primary`, it:

- Creates a new asset record with asset_id set to the token ID
- Initializes the balance to 0
- Potentially reallocates the account if no free slot is available

Subsequently, `sub_asset_tokens(data.amount)` is called, which subtracts the withdrawal amount from 0, resulting in a negative balance. The check then fails with InsufficientFunds.

```
    client_state.sub_asset_tokens(data.amount)?;
    if client_state.asset_tokens()? < 0 {
        bail!(InsufficientFunds);
    }
```

This is semantically incorrect because:

- Withdrawals should only operate on existing assets with positive balances

Note:

In some cases, the `finalize_spot` will be called in `clean_generic`.

```
    if accounts.len() > 11 + extra_accounts {
        client_state.clean_generic(accounts_iter, accounts.len() - 11 - extra_accounts)?;
    }
```

But the asset record will be created anyway in `clean_generic`:

```
            self.resolve_instr(client_infos_acc, false)?;
```

So, we don't ever need to put `alloc=true`:

```
    if accounts.len() > 11 + extra_accounts {
        client_state.clean_generic(accounts_iter, accounts.len() - 11 - extra_accounts)?;
    }
    client_state.resolve(AssetType::Token, data.token_id, TokenType::Asset, true)?;
```

Impact: The instruction should fail immediately if the token doesn't exist, not create a record first.

Recommended Mitigation: Change the `alloce` to `false`

```
client_state.resolve(AssetType::Token, data.token_id, TokenType::Asset, false)?;
```

Deriverse: Fixed in commit 45ee168.

Cyfrin: Verified.

7.4.18 Users Cannot Change Their Vote Once Cast in a Voting Period

Description: The voting system prevents users from changing or revoking their votes once they have voted in a voting round. While this maybe an intentional design to prevent duplicate voting, it may impact user experience if users make mistakes or want to change their vote before the voting period ends.

In `voting`, the system checks if a user has already voted:

```
if client_community_state.header.last_voting_counter
    >= client_community_state.header.current_voting_counter
{
    bail!(AlreadyVoted);
```

```
}
```

After a successful `vote`, `last_voting_counter` is set to the current `voting_counter`, which prevents the user from voting again in the same round, even if they want to:

- Correct a mistaken vote
- Change their vote choice (e.g., from INCREMENT to DECREMENT)
- Revoke their vote

Impact: Users who make mistakes cannot correct them or they cannot change their vote if they change their mind during the voting period.

Recommended Mitigation: Consider allow vote changes with proper accounting.

Deriverse: Fixed in commit 1689196.

Cyfrin: Verified.

7.4.19 Using a stale `perp_price_delta` to calculate the `perp_funding_rate`

Description: The funding rate is calculated using `change_funding_rate`, but this function computes `perp_funding_rate` based on the previous `perp_price_delta`, which is outdated.

```
    pub fn change_funding_rate(&mut self) {
        let time_delta = self.time - self.state.header.perp_funding_rate_time;
        if time_delta > 0 && self.state.header.perp_price_delta != 0.0 {
            self.state.header.perp_funding_rate +=
                ((time_delta as f64) / DAY as f64) * self.state.header.perp_price_delta;
        }
        self.state.header.perp_price_delta =
            (self.market_px() - self.state.header.perp_underlying_px) as f64 * self.rdf;
        self.state.header.perp_funding_rate_time = self.time;
    }
```

This can result in losses for some users and unintended profits for others, depending on whether the market price is below or above the underlying price.

Scenario:

1. The market price and the underlying price of Bitcoin are both 100k.
2. Alice opens a 1× long position for 1 BTC at 100k, and Bob opens a 1 BTC short position at the same price.
3. After one day, the underlying price of Bitcoin increases from 100k to 110k, but the market price remains unchanged.
4. When Alice and Bob attempt to close their positions after one day, the funding rate is calculated using the old delta.
5. As a result, Alice receives no funding payment even though the difference between the market price and underlying price is now 10k, and Bob does not pay any funding despite being on the paying side.

The funding rate is intended to ensure that the market price tracks the underlying price. Calculating the funding rate using an old delta fails to create this corrective force.

Impact: The funding rate is being calculated using a stale delta, users may end up receiving or paying funding later than they should. This can lead to unintended losses or profits depending on when a user enters or exits a position, as shown in the scenario above.

Recommended Mitigation: To mitigate this, `change_funding_rate` should first calculate the updated delta and then compute the funding rate based on that value.

```
    pub fn change_funding_rate(&mut self) {
        let time_delta = self.time - self.state.header.perp_funding_rate_time;
```

```
        self.state.header.perp_price_delta =
            (self.market_px() - self.state.header.perp_underlying_px) as f64 * self.rdf;
        if time_delta > 0 && self.state.header.perp_price_delta != 0.0 {
            self.state.header.perp_funding_rate +=
                ((time_delta as f64) / DAY as f64) * self.state.header.perp_price_delta;
        }
        self.state.header.perp_price_delta =
            (self.market_px() - self.state.header.perp_underlying_px) as f64 * self.rdf;
        self.state.header.perp_funding_rate_time = self.time;
    }
```

Deriverse: Acknowledged; we think our approach is better as in a scenario where the price delta was unchanged most of the time between perp transactions our approach is more relevant.

7.4.20 Incorrect Token Program Error Message in Airdrop Flow

Description: The airdrop instruction correctly enforces that DRVS tokens must use the `Token-2022` program, but the error message suggests the opposite. This inconsistency can mislead operators when diagnosing configuration mistakes.

```
if check_spl_token(...) ? {
    let token_program_version = TokenProgram::new(token_program.key)?;
    bail!(InvalidTokenProgramId {
        expected: TokenProgram::Original,
        actual: token_program_version,
    });
}
```

The bailout is correct: the code later calls `spl_token_2022::instruction::transfer_checked`, so Token-2022 is required. However, the error message reports `expected: TokenProgram::Original`, which is the *rejected* option.

Impact: Operators who misconfigure the mint/program pair will see an error stating that "Original" was expected even though the instruction actually expects Token-2022.

Recommended Mitigation: Update the `InvalidTokenProgramId` message to reflect the real expectation.

Deriverse: Fixed in commit 40043ae.

Cyfrin: Verified.

7.4.21 Incorrect Boundary Condition in `check_pool_fees` Function Excludes Valid Minimum Token Transactions

Description: The `check_pool_fees` function uses a strict greater-than (>) comparison when checking if `tokens_qty` meets the `min_tokens` requirement. This incorrectly excludes the valid boundary case where `tokens_qty` equals `min_tokens`, preventing legitimate pool fee conversions when the calculated quantity exactly matches the minimum allowed value.

In `src/program/spot/engine.rs`, the `check_pool_fees` function contains the following condition:

```
pub fn check_pool_fees(
    &mut self,
    client_state: &mut ClientPrimaryState,
    rest_of_order: &mut i64,
    traded_sum: &mut i64,
    price: i64,
    min_tokens: i64,
) -> DeriverseResult {
    let mints_qty = self.state.header.pool_fees >> 1;
    let tokens_qty = (self.state.header.asset_tokens as f64 * mints_qty as f64
        / self.state.header.crncy_tokens as f64) as i64;
    if tokens_qty > min_tokens && *rest_of_order > min_tokens + tokens_qty {
        // ... execute pool fee conversion logic
```

```
    }
    Ok(())
}
```

This violates with other parts of the code, like:

```
    if !(data.order_type == OrderType::Market as u8 || data.ioc != 0) && data.amount < min_tokens {
        bail!(InvalidQuantity {
            value: data.amount,
            min_value: min_tokens,
            max_value: SPOT_MAX_AMOUNT
        })
    }
```

Impact: Valid pool fee conversions are incorrectly rejected when the calculated tokens_qty exactly equals min_tokens, reducing the efficiency of pool fee utilization

Recommended Mitigation: Change the comparison operator from > to >= to include the valid boundary case.

Deriverse: Fixed in commit 3a24de8.

Cyfrin: Verified.

7.4.22 wrong error emitted from `spot-order-cancel`

Description: Inside spot-order-cancel, wrong error is emitted when accounts length is more than expected length, which is wrong and inconsistent with rest of the places.

```
    if accounts.len() < SpotOrderCancelInstruction::MIN_ACCOUNTS {
        //@audit wrong, should have been InvalidAccountsNumber
        bail!(InvalidDataLength {
            expected: SpotOrderCancelInstruction::MIN_ACCOUNTS,
            actual: accounts.len(),
        });
    }
```

Impact: wrong errors cause confusions and give hard time debugging underlying issue.

Recommended Mitigation: Replace it with

```
        bail!(InvalidAccountsNumber {
            expected: SpotMassCancelInstruction::MIN_ACCOUNTS,
            actual: accounts.len(),
        });
```

Deriverse: Fixed in commit : 4b2e123

Cyfrin: verified.

7.4.23 Redundant Flag `READY_TO_DRV_UPGRADE` Appears Unused

Description: The `READY_TO_DRV_UPGRADE` flag appears to be redundant and potentially dead code. It is always set together with `READY_TO_PERP_UPGRADE`, but there is no separate DRV upgrade functionality in the codebase. The flag is never independently checked or used, suggesting it may be leftover code from a planned feature that was never implemented or was removed.

7.5 Vulnerability Details

The codebase defines two flags for instrument upgrade:

1. `READY_TO_PERP_UPGRADE` - indicates readiness for perpetual upgrade

2. `READY_TO_DRV_UPGRADE` - indicates readiness for DRV upgrade (purpose unclear since it's never shown in the code)

Analysis:

1. **No separate DRV upgrade functionality exists:**
   - In the codebase, there's no `upgrade_to_drv` function or `UpgradeToDrvInstruction`
   - The only upgrade function is `upgrade_to_perp`
2. **Flags are always set together:**
   - In `set_instr_ready_for_perp_upgrade`, both flags are set:

```
instr_header.mask |= instr_mask::READY_TO_PERP_UPGRADE;
instr_header.mask |= instr_mask::READY_TO_DRV_UPGRADE;
```

   - In `update_variance`, both flags are set together:

```
self.mask |= instr_mask::READY_TO_DRV_UPGRADE;
self.mask |= instr_mask::READY_TO_PERP_UPGRADE;
```

3. However, `READY_TO_DRV_UPGRADE` **is never independently checked:**
   - In `upgrade_to_perp` (line 148-150), only `READY_TO_PERP_UPGRADE` is checked:

```
if instr_header.mask & instr_mask::READY_TO_PERP_UPGRADE == 0
    || instr_header.mask & instr_mask::PERP != 0
```

   - `READY_TO_DRV_UPGRADE` is **never checked** in any upgrade function
4. **Only used in** `update_variance` **with AND logic:**
   - In `update_variance` (line 51-52), both flags are checked together:

```
if self.mask & instr_mask::READY_TO_DRV_UPGRADE == 0
    && self.mask & instr_mask::READY_TO_PERP_UPGRADE == 0
```

   - Since they're always set together, checking both is redundant

Impact: Dead Code / Redundancy: `READY_TO_DRV_UPGRADE` appears to serve no purpose since:

- There is no DRV upgrade functionality
- It's always set together with `READY_TO_PERP_UPGRADE`
- It's never independently checked or used

Recommended Mitigation: Remove the flag entirely if DRV upgrade is not planned.

Deriverse: Fixed in commit 75da0c.

Cyfrin: Verified.

7.5.1 Incorrect mask check in `check_points` function

Description: The `check_points` function uses an incorrect bitmask(0x7FF) to determine whether milestone points should be checked. This mask covers bits 0–10, but it should check for bits 0, 1, 2, 9, 10, 11, 12, and 13. As a result, the function incorrectly evaluates the condition even when the user has already received all eight points.

```
if self.mask & 0x7FF != 0x7FF {
```

Impact: The mask value 0x7FF was intended to optimize the function by skipping checks when all milestones are achieved, but it does not work correctly.

Recommended Mitigation: We should check against 0x3E07. If (mask & 0x3E07) != 0x3E07 is false, then we should skip the point checks.

```
            if self.mask & 0x3E07 != 0x3E07 {
```

Deriverse: Fixed in commit 3ffcaa.

Cyfrin: Verified.