Synthetix Security Review

Pashov Audit Group

Conducted by: Koolex, FrankCastle, ZanyBonzy, zhaojio

November 5th 2024 - December 13th 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **SynthetixSolana/synthetix-solana** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Synthetix on Solana

Synthetix provides a flexible framework for creating derivatives and managing liquidity across blockchain networks. It enables users to stake collateral, provide liquidity, and trade derivatives. The V3 design introduces modular features like collateral vaults, liquidity pools, and advanced trading tools for smooth operation. This audit was focused on Synthetix on Solana, written on Rust.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - d8a4172b1432b712ab0158f7365c6862323c8c86

fixes review commit hash - d8a4172b1432b712ab0158f7365c6862323c8c86

Scope

The following smart contracts were in scope of the audit:

```
- accept
- account
- add
- add_config
- assign_debt_to_account
- associated_markets
- burn
- burn_usd_for_market
- claim
- claim_liquidated_collateral
- close
- collateral_disabled_by_default
- collateral_mint_definition
- commit
- complete
- configure
- consolidate_debt
- core_config
- core_market_reporting
- create
- create_lock
- create_market
- create_user
- custody
- delegate_collateral
- delegation_cooldown_duration
- deposit
- deposit_collateral
- deposit_usd
- disable
- distribute
- distribute_debt
- distribute_debt_to_accounts
- distribute_debt_to_vaults
- enabled
- error
```

```
- feature_flag
- flag_position
- funding_recomputed
- global_config
- initialize_program
- keeper
- keeper_liquidation_management
- keeper_margin_management
- lib
- liquidate_flagged_position_margin
- liquidate_margin
- liquidate_position
- liquidation
- manage_collateral
- margin
- margin_collateral_configured
- margin_liquidated
- market_collateral_access
- market_computation_access
- market_connections
- market_created
- market_debt
- market_size_updated
- market_usd_config
- max
- max_delegation_cooldown_duration
- min_liquidity_ratio
- minimum_credit
- mint
- mint_usd_for_market
- mod
- name
- nominate
- oracle_node_definition
- oracle_price
- order
- order_canceled
- order_committed
- order_settled
- pay_debt
- perp_market
- pool_owner_only
- pools
- position
- position_flagged
- position_liquidated
- preferred_pool
- prepare
- prepare_claim
- process_price
- program_owner_only
- pyth
- rebalance_markets
- recalculate_vault_collateral
```

```
- register
- register_market
- remove
- renounce_nomination
- renounce_ownership
- settle
- synthetix_perps_account
- synthetix_usd
- total_debt
- trader
- transfer_fees
- update
- update_depositing_enabled
- update_market_config
- update_one_pool_only
- update_pause
- user
- user_account_owner_only
- user_collateral
- utilization_recomputed
- vault
- withdraw
- withdraw_collateral
- withdraw_usd
```

7. Executive Summary

Over the course of the security review, Koolex, FrankCastle, ZanyBonzy, zhaojio engaged with Synthetix to review Synthetix on Solana. In this period of time a total of **16** issues were uncovered.

Protocol Summary

Protocol Name	Synthetix on Solana
Repository	https://github.com/SynthetixSolana/synthetix-solana
Date	November 5th 2024 - December 13th 2024
Protocol Type	Derivative trading

Findings Count

Severity	**Amount**
High	1
Medium	4
Low	11
Total Findings	**16**

Summary of Findings

ID	Title	Severity	Status
[H-01]	Rewards loss due to share updates in claim calculation	High	Resolved
[M-01]	UserCollateral.vaults array has no length limit	Medium	Resolved
[M-02]	reward_distribution.schedule needs to be updated when slot_duration == 0	Medium	Resolved
[M-03]	Permanent DoS on liquidation process	Medium	Resolved
[M-04]	Permanent DoS during the price update	Medium	Resolved
[L-01]	Preventing account closure by sending small token amount	Low	Acknowledged
[L-02]	Overflow in try_update_market_size_on_trade	Low	Resolved
[L-03]	Missing validation for minimum publish time	Low	Resolved
[L-04]	Risky token extensions	Low	Acknowledged
[L-05]	Not clearing debt when amount_delta_usd == 0	Low	Resolved
[L-06]	Function is missing	Low	Resolved
[L-07]	Improper validation of susd_mint	Low	Resolved
[L-08]	create_collateral_lock only allows increasing the lock amount when the previous lock expires	Low	Resolved
[L-09]	Missing update to	Low	Resolved

	last_interaction_slot		
[L-10]	Keepers can frontrun the trader order cancellation	Low	Acknowledged
[L-11]	Missing confidence interval validation	Low	Acknowledged

8. Findings

8.1. High Findings

[H-01] Rewards loss due to share updates in claim calculation

Severity

Impact: High

Likelihood: Medium

Description

The reward calculation in `prepare_reward_claim` is based on the current share balance when calculating unclaimed rewards:

```
let user_shares = i128::try_from(connection.holdings.shares)?;
.
.
let new_reward = reward_distribution
    .reward_per_share
    .saturating_sub(reward_claim.last_reward_per_share)
    .checked_mul_decimal(user_shares)
```

However, any modification to shares through `delegate_vault_collateral` before calling `prepare_reward_claim` will affect this calculation. This leads to several issues:

- If a user reduces their delegation before claiming:
  - The rewards will be calculated based on the new (lower) share amount
  - They lose rewards they should have earned on their previous higher share amount
- If a user increases their delegation before claiming:
  - The rewards will be calculated based on the new (higher) share amount
  - They receive more rewards than they should have earned which is unfair to other users
- Special case - Full withdrawal (new_amount = 0):

```
if new_amount == 0 {
    user_collateral.vaults.swap_remove(connection_index);
}
```

- The vault connection is completely removed
- When trying to claim rewards, the operation fails with `UserToVaultConnectionNotFound`
- This makes it impossible to ever claim rewards after a full withdrawal
- This is particularly severe as it guarantees complete loss of earned rewards

This creates both potential loss of earned rewards and potential unfair distribution of rewards, depending on the timing of operations, with full withdrawals being a guaranteed loss scenario.

Risk Assessment

While instruction of `prepare_reward_claim` can be added by the user before delegating (in the same TX), this solve only the case if the shares are being reduced or connection is removed. This puts the full responsibility on the user which is not ideal, or on the developer/integrator which might not be aware of this.

However, in case the shares are being increased, the user has the incentive **not** to do so, causing unfair distrubtion of rewards comparing to other users.

Recommendations

There are many ways to resolve this, one is:

```
// create prepare_reward_claim_internal function and
// In delegate_vault_collateral, before modifying shares:
if old_shares > 0 {
    self.prepare_reward_claim_internal(
        vault,
        reward_distribution,
        old_shares,
        current_slot
    )?;
}
```

Please take into account that, `prepare_reward_claim_internal` might fail which could block delegation. So, for this, I suggest that there is a flag passed as a param (bool) which gives the user the option to skip rewards or revert if there are existing unprepared rewards.

8.2. Medium Findings

[M-01] `UserCollateral.vaults` array has no length limit

Severity

Impact: Medium

Likelihood: Medium

Description

UserCollateral.vaults is an array that can be found by `pool_id` and `vault_epoch`:

```
pub struct UserCollateral {
    pub account_id: u64,
    pub mint: Pubkey,

    pub signer_bump: u8,
    pub custody_bump: u8,

    pub lock: UserCollateralLock,

    /// NOTE: For InitSpace, we only consider the length of the vector. Whenever
    // this vector
    /// changes, we will need to perform an account realloc.
    #[max_len(0)]
    pub vaults: Vec<UserToVaultConnection>,
}

    pub fn find_vault_connection(
        &self,
        pool_id: u64,
        vault_epoch: u32,
    ) -> Option<(usize, &UserToVaultConnection)> {
        self.vaults
            .iter()
            .position(|connection| {

                                connection.pool_id == pool_id && connection.vault_epo
            })
            .map(|index| (index, &self.vaults[index]))
    }
```

The user can add any number of elements to the array using a different pool_id(Anyone can create a pool).

`liquidate_account` instruction When liquidating a user, it traverses the vaults array. If there are too many elements in the array, the execution will fail due to insufficient gas, so that the malicious user can prevent liquidation.

```
pub fn liquidate_account<'info> {
    ......
    let (vault_index, _) = user_collateral
        .find_vault_connection(pool.id, vault.epoch)
        .ok_or_else(|| error!
          (ProgramErrorCode::UserToVaultConnectionNotFound))?;
   ......
}
```

In addition, when the vault is liquidated, a new epoch is set, Since `user_collateral.vault` is composed of `pool.id` and `vault.epoch`, when vault.epoch is updated, the old elements will remain in `user_collateral.vaults` without being removed.

```
let new_epoch = vault.epoch.checked_add(1).ok_or_else(|| error!
              (CommonErrorCode::NumericOverflow))?;

            // All of this data persists.
            let vault_base = vault.base;
            let vault_config = vault.config;

            // Reset the existing vault data and uptick its epoch. We use
            // set_inner to ensure we cover all fields.
            vault.set_inner(Vault {
                base: vault_base,
@>          epoch: new_epoch,
                config: vault_config,
                //
                ..Default::default()
            });
```

Recommendations

Describe your recommendation here `UserCollateral.vaults` Increases length limits and allows users or administrators to delete invalid data.

[M-02] `reward_distribution.schedule` needs to be updated when slot_duration == 0

Severity

Impact: Medium

Likelihood: Medium

Description

The reward distribution will determine how to distribute the reward based on time. If slot_duration == 0, `checked_distribution_per_share` function will return `value/total_shares` directly.

```
fn checked_update_reward_per_share(
    reward_distribution: &mut RewardDistribution,
    debt_distribution: &AccountsDebtDistribution,
    RewardSchedule {
        value,
        start_slot,
        slot_duration,
    }: RewardSchedule,
    current_slot: u64,
) -> Option<()> {
    // Unlocks the entry's distributed amount into its value per share.
    //    let debt_distribution = &vault.users.share_distribution;
    // distributed = value / total_shares
    let mut distributed = reward_distribution.checked_distribution_per_share
      (debt_distribution, current_slot)?;

    // If the current slot is past the end of the entry's duration, update any
    // rewards which
    // may have accrued since last run.
    if current_slot >= start_slot.saturating_add(slot_duration.into()) {
@>   reward_distribution.schedule = Default::default();
        reward_distribution.last_update_slot = Default::default();

        let total_shares = debt_distribution.try_total_shares().ok()?;

        //distributed = value / total_shares + distributed
        // distributed = 2 * (value / total_shares)
        distributed = value.checked_div_decimal(total_shares)?.checked_add
          (distributed)?;
    }
    // Otherwise, schedule the amount to distribute.
    else {
@>       reward_distribution.schedule = RewardSchedule {
            value,
            start_slot,
            slot_duration,
        };

        // The amount is actually the amount distributed already *plus* whatever
        // has been specified now.
        reward_distribution.last_update_slot = Default::default();

        distributed = reward_distribution
            .checked_distribution_per_share(debt_distribution, current_slot)?
            .checked_add(distributed)?;
    }



    Some(())
}
```

```
pub fn checked_distribution_per_share(
        &mut self,
        accounts_debt_distribution: &AccountsDebtDistribution,  let
        // debt_distribution = &vault.users.share_distribution;
        current_slot: u64,
    ) -> Option<i128> {
        let RewardSchedule {
            value,
            start_slot,
            slot_duration,
        } = self.schedule;

        // We will need this for various checks in this method.
        let end_slot = start_slot.saturating_add(slot_duration.into());

        if value == 0
            || accounts_debt_distribution.total_shares == 0
            // No balance if current time is before start slot.
            || current_slot < start_slot
        {
            Some(Default::default())
        }
        // If the entry's duration is zero, consider the entry to be an instant
        // distribution.
@>      else if slot_duration == 0 {
            let total_shares = accounts_debt_distribution.try_total_shares().ok
              ()?;
 @>      value.checked_div_decimal(total_shares)
        }
```

The problem is that if `reward_distribution.schedule` is not reset, `checked_distribution_per_share` will always return value/total_shares,

`prepare_reward_claim` will re-call `checked_distribution_per_share`, so the reward will be double-counted.

```
pub fn prepare_reward_claim(
    PrepareRewardClaim {
        fee_payer: _,
        user_admin,
        vault,
        user_collateral,
        reward_distribution,
        reward_claim,
        system_program: _,
    }: &mut PrepareRewardClaim,
) -> Result<()> {
    // Unlocks the entry's distributed amount into its value per share.
    let distributed = reward_distribution
@>      .checked_distribution_per_share
  (&vault.users.share_distribution, Clock::get().unwrap().slot)
        .ok_or_else(|| error!(ProgramErrorCode::FailedToDistributeRewards))?;
     ......
```

It depends on how the distributor sets the parameters, This happens if `current_slot < start_slot-slot_duration` and `slot_duration= 0`, `reward_distribution.schedule` will not be reset when `current_slot <`

`start_slot`, When `slot_duration= 0`, the `checked_distribution_per_share` function returns `value/total_shares`.

So the reward will double count.

Recommendations

if slot_duration == 0 reward_distribution.schedule = Default::default();

[M-03] Permanent DoS on liquidation process

Severity

Impact: High

Likelihood: Low

Description

The `add_market_collateral_config` function allows an unlimited number of collateral types to be added to the system without imposing a maximum limit. This creates a risk of permanent denial of service (DoS) for the liquidation process when the number of supported collateral types becomes too large.

The problematic code snippet is as follows:

```
market.deposited_collaterals.push(CoreMarketCollateral {
    is_enabled: true,
    custody_bump: bumps.market_custody_token,
    collateral: DepositedCollateral::try_new(collateral_mint, Default::default
      ()).unwrap(),
    max_deposit_allowed,
});
```

Problem Analysis:

1. **Unlimited Collateral Types:**
   The margin.`deposited_collaterals` array grows upto the margin supported collateral length as users deposit small amounts of various collateral types.

2. **Price Update Requirements:**
   The liquidation process requires up-to-date prices for each collateral type. Specifically, two price update instructions are needed per collateral type within the same slot.
   `process_oracle_node_price` and `process_collateral_definition_price`

3. **Scalability Issue:**
   If the system supports 50 collateral types, it would require 100 price update instructions. Given Solana's constraints, it becomes highly improbable to execute all required updates within a single slot.

4. **Blocked Liquidation:**
   The liquidation process fails if the prices are not up-to-date for all collateral types, resulting in permanent blocking of the liquidation mechanism.

Impact:

- **Permanent DoS:** The liquidation process becomes inoperable if the number of supported collateral types exceeds the system's capacity to handle price updates in the same slot.

Recommendation

1. **Set a Maximum Limit for Supported Collaterals:**
   Define and enforce a maximum size for the `deposited_collaterals` vector. For example:

```
const MAX_COLLATERALS: usize = 20;
if market.deposited_collaterals.len() >= MAX_COLLATERALS {
    return Err(error!(ProgramErrorCode::MaxCollateralsExceeded));
}
```

[M-04] Permanent DoS during the price update

Severity

Impact: High

Likelihood: Low

Description

The function `process_collateral_definition_price` updates the collateral price using the `node_definition` without providing a valid `comparison_node`. This results in a permanent denial of service (DoS) to the function due to the behavior of the `price_deviation_circuit_breaker` mechanism.

The circuit breaker enforces slippage protection by validating price deviations between the primary node and a comparison node. If no valid comparison node is provided, the circuit breaker will revert with an error, causing the function to fail. The current implementation passes `None` as the `comparison_node` in the following line:

```
node_definition.try_update_cached_price
  (price_feed, None, current_slot, current_timestamp)?;
```

Issues:

1. **Reversion of Updates:**
   The `try_update_cached_price` call always reverts when no valid `comparison_node` is provided and the circuit breaker is defined.

2. **Unprotected Collateral Price Updates:**
   If the circuit breaker is set to `None`, the collateral price updates without slippage protection, exposing the protocol to potential manipulation.

3. **Complete Protocol DoS:**
   Since the function fails under valid configurations (when the circuit breaker is enabled), this issue can prevent collateral prices from being updated entirely, leading to a permanent DoS.

Impact

- **Permanent DoS:** The `process_collateral_definition_price` function fails to update collateral prices, halting protocol operations dependent on up-to-date price data.

Recommendations

1. Remove the Invalid Function Call

Remove the call to `try_update_cached_price` since the subsequent `update` function already ensures the latest price is used:

```
// Remove this line:
node_definition.try_update_cached_price
  (price_feed, None, current_slot, current_timestamp)?;
```

2. Add Explicit Circuit Breaker Handling

If additional functionality is required, handle the circuit breaker and comparison logic explicitly to avoid unexpected reversion:

```
match &node_definition.price_deviation_circuit_breaker {
    Some(_) => {
        require!(
            comparison_node_definition.is_some(),
            ProgramErrorCode::MissingComparisonNodeDefinition
        );
    },
    None => {} // Proceed without circuit breaker checks.
}
```

8.3. Low Findings

[L-01] Preventing account closure by sending small token amount

In the `withdraw_account_collateral` function, when a user attempts to withdraw their full balance, the function checks if the withdrawal amount equals the total balance to determine if the accounts should be closed:

```
if amount == total_collateral && user_collateral.vaults.is_empty() {
    token_interface::close_account(...)?;
    user_collateral.close(...)?;
}
```

A malicious actor can front-run the withdrawal transaction and send a small amount (e.g., 0.000000001 tokens) to the custody account. This would cause:

- The `total_collateral` captured at the start of the transaction will be slightly higher than the user's withdrawal amount
- The condition `amount == total_collateral` will be false
- The accounts will not be closed, forcing the user to maintain rent payments for both:
  - The user_collateral account
  - The custody token account

For tokens with 9 decimals, an attacker only needs to send 0.000000001 tokens (costing almost nothing) to prevent closure of accounts whose rent cost is much higher (given the value of Sol).

While this doesn't result in direct token loss, it creates a denial of service where users are forced to keep paying rent for accounts they intended to close. The attack is particularly cheap for the attacker as they only need to send a minimal amount of tokens to force continued rent payments from the victim.

Consider adding a full withdrawal flag, if true, it means the user intends to withdraw the full amount, therefore, withdraw it fully and close the accounts. For partial withdrawal, this is not effected.

[L-02] Overflow in `try_update_market_size_on_trade`

The `try_update_market_size_on_trade` function updates the market size and skew by accounting for the trader's position changes. It calculates the total market size using the absolute values of the `new_trader_pos` (new position size) and `trader_pos` (old position size). However, the function first adds the absolute value of `new_trader_pos` to the market size before subtracting the absolute value of `trader_pos`. This sequence introduces a potential for overflow in the intermediate calculation, even if the net difference (`delta_size`) is valid and does not exceed the `u128` maximum value (`u128::MAX`). This intermediate overflow results in valid trades being reverted, causing unexpected behavior and a denial of service for valid order settlements.

```
pub(crate) fn try_update_market_size_on_trade(
    &mut self,
    new_trader_pos: i128,
    trader_pos: i128,
) -> Result<()> {
    let data = &mut self.data;

    data.size = data
        .size
        .checked_add(u128::try_from(new_trader_pos.abs())?)
        .ok_or_else(|| error!(CommonErrorCode::NumericOverflow))?
        .saturating_sub(u128::try_from(trader_pos.abs())?);

    data.skew = data
        .skew
        .checked_add(new_trader_pos)
        .ok_or_else(|| error!(CommonErrorCode::NumericOverflow))?
        .saturating_sub(trader_pos);

    Ok(())
}
```

Proof of Concept (POC)

Scenario 1: Increasing Position Causes Overflow

This test demonstrates a valid increase in position size being incorrectly reverted due to an intermediate overflow.

```rust
#[test]
fn test_update_market_size_on_trade() {
    let mut market = perp_market_for_test();
    let trader_pos = 4_000_000_000;
    let new_trader_pos = 5_000_000_000;
    // The size needed for this position to be stored is 1_000_000_000
    // The empty size is 2_000_000_001
    // The trade should be marked valid since it is within u128::MAX
    let size_before = u128::MAX.checked_sub(2_000_000_000u128).unwrap();
    market.data.size = size_before;

    market
        .try_update_market_size_on_trade(new_trader_pos, trader_pos)
        .unwrap();
}
```

Scenario 2: Decreasing Position Causes Overflow

This test demonstrates a valid decrease in position size being incorrectly reverted due to an intermediate overflow, even though the market size is ultimately reduced.

```rust
#[test]
fn test_update_market_size_on_trade() {
    let mut market = perp_market_for_test();
    let trader_pos = 4_000_000_000;
    let new_trader_pos = 3_000_000_000;
    // The empty size in the market is 2_000_000_001
    // We are reducing the size so it should not overflow; new_pos < old_pos
    // The trade should be marked valid since it is within u128::MAX
    let size_before = u128::MAX.checked_sub(2_000_000_000u128).unwrap();
    market.data.size = size_before;

    market
        .try_update_market_size_on_trade(new_trader_pos, trader_pos)
        .unwrap();
}
```

Impact: DoS on Order Settlement: Valid trades are reverted due to overflow errors, leading to disrupted order settlement processes. **Loss of Funds:** The protocol and keepers lose potential fees associated with the reverted trades.

Fix: Reverse the order of operations in `try_update_market_size_on_trade`. Subtract the absolute value of `trader_pos` from the market size before adding the absolute value of `new_trader_pos`. This ensures intermediate calculations avoid overflow.
Fixed Function:

```rust
pub(crate) fn try_update_market_size_on_trade(
    &mut self,
    new_trader_pos: i128,
    trader_pos: i128,
) -> Result<()> {
    let data = &mut self.data;

    data.size = data
        .size
        .checked_sub(u128::try_from(trader_pos.abs())?)
        .ok_or_else(|| error!(CommonErrorCode::NumericOverflow))?
        .checked_add(u128::try_from(new_trader_pos.abs())?)
        .ok_or_else(|| error!(CommonErrorCode::NumericOverflow))?;

  ......

    Ok(())
}
```

[L-03] Missing validation for minimum publish time

The `try_price` function retrieves the most recent price from the cached oracle price. However, it does not validate whether the price's `publishTime` meets the specified minimum limit (`pyth_publish_time_min`). This omission allows prices that are too recent to bypass validation, potentially resulting in incorrect calculations when fetching the oracle price to determine the collateral value.

`pyth_publish_time_min` and `pyth_publish_time_max`:

```rust
/// Minimum acceptable publishTime from Pyth WH VAA price update data.
pub pyth_publish_time_min: i64,

/// Max acceptable publishTime from Pyth.
pub pyth_publish_time_max: i64,
```

`try_price` Function:

```rust
pub fn try_price(&self, current_slot: u64) -> Result<i128> {
    require_eq!(
        self.cached_price.last_updated_slot,
        current_slot,
        ProgramErrorCode::StalePrice
    );

    Ok(self.cached_price.to_inner())
}
```

Here, `try_price` validates that the cached price is not stale but does not enforce a check against `pyth_publish_time_min`.

The absence of a minimum `publishTime` check allows overly recent oracle prices to influence collateral value calculations. This could lead to incorrect collateral requirements or liquidation thresholds, potentially causing financial loss or instability in the protocol.

- Update the `try_price` function to include validation against `pyth_publish_time_min`.
- Example fix:

```
pub fn try_price(&self, current_slot: u64) -> Result<i128> {
    require_eq!(
        self.cached_price.last_updated_slot,
        current_slot,
        ProgramErrorCode::StalePrice
    );

    let publish_time = self.cached_price.publish_time;
    require!(
        publish_time >= self.pyth_publish_time_min,
        ProgramErrorCode::RecentPriceNotAllowed
    );

    Ok(self.cached_price.to_inner())
}
```

[L-04] Risky token extensions

The function `deposit_market_collateral` allows users to deposit collateral into the protocol. However, it does not filter or validate the token extensions of the collateral being deposited. Certain token extensions, such as the `TransferFee` extension, can cause discrepancies between the transferred and received token amounts. For instance, with the `TransferFee` extension, the protocol may trust the user-specified transfer amount, but the actual amount received may be less due to fees. This discrepancy can lead to financial damage to the protocol. Below is the relevant snippet of the function:

```
pub fn deposit_market_collateral(
    DepositMarketCollateral {
        update_collateral:
            UpdateMarketCollateral {
                access:
                    MarketCollateralAccess {
                        market,
                        feature_authority: _,
                        feature_flag: _,
                    },
                mint: collateral_mint,
            },
        source_token,
        market_custody_token,
        token_program,
        event_authority: _,
        program: _,
    }: &mut DepositMarketCollateral,
    amount: u64,
) -> Result<MarketCollateralDeposited> {
    // Update deposited collateral amount.
    {
        let (index, _) = market
            .find_deposited_collateral(&collateral_mint.key())
            .unwrap();
        let deposited_collateral = &mut market.deposited_collaterals[index];
        require!(
            deposited_collateral.is_enabled,
            ProgramErrorCode::MarketCollateralDisabled
        );

        market.deposited_collaterals[index].try_add_amount
          (collateral_mint, amount)?;
    }

    // And transfer amount into the system.
    token_interface::transfer_checked(
        CpiContext::new_with_signer(
            token_program.to_account_info(),
            token_interface::TransferChecked {
                from: source_token.to_account_info(),
                mint: collateral_mint.to_account_info(),
                to: market_custody_token.to_account_info(),
                authority: market.to_account_info(),
            },
            &[&[
                CoreMarket::SEED_PREFIX,
                &market.id.to_seed(),
                &[market.signer_bump],
            ]],
        ),
        amount,
        collateral_mint.decimals,
    )?;

    Ok(MarketCollateralDeposited {
        market_id: market.id,
        collateral_type: collateral_mint.key(),
        token_amount: amount,
        program_id: market.program_id,
    })
}
```

Since the function does not validate or filter tokens based on their extensions, it exposes the protocol to potential financial losses. For example, a token with

the `TransferFee` extension will reduce the number of tokens received in the `market_custody_token` account, breaking assumptions about the amount of collateral deposited.

To mitigate this vulnerability:

1. **Filter Unsupported Token Extensions:** Use a function to validate supported token extensions and reject risky ones such as `TransferFee` or `CloseMint`. Update the protocol's documentation to reflect these restrictions.

```
pub fn is_supported_mint(mint_account: &InterfaceAccount<Mint>) -> bool {
    let mint_info = mint_account.to_account_info();
    let mint_data = mint_info.data.borrow();
    let mint = StateWithExtensions::<spl_token_2022::state::Mint>::unpack
      (&mint_data).unwrap();
    let extensions = mint.get_extension_types().unwrap();
    for e in extensions {
        match e {
            // Allow only safe extensions
            ExtensionType::TransferFee => return false,
            ExtensionType::CloseMint => return false,
            _ => {},
        }
    }
    true
}
```

2. **Update the Function Logic:** Incorporate the filtering logic into `deposit_market_collateral` to reject unsupported tokens before processing the deposit.

3. **Adjust Code to Support Extensions:** If supporting extensions like `TransferFee` is necessary, calculate the post-fee amount to determine the actual tokens received by the protocol. Here is a sample function to calculate the post-fee amount:

```rust
pub fn get_post_fee_amount_ld
  (token_mint: &InterfaceAccount<Mint>, amount: u64) -> Result<u64> {
    let token_mint_info = token_mint.to_account_info();
    let token_mint_data = token_mint_info.try_borrow_data()?;
    let token_mint_ext = StateWithExtensions::<MintState>::unpack
      (&token_mint_data)?;
    let post_amount = if let Ok
      (transfer_fee_config) = token_mint_ext.get_extension::<TransferFeeConfig>() {
        transfer_fee_config
            .get_epoch_fee(Clock::get()?.epoch)
            .calculate_post_fee_amount(amount)
            .ok_or(ProgramError::InvalidArgument)?
    } else {
        amount
    };
    Ok(post_amount)
}
```

Implementing these measures will safeguard the protocol from financial losses due to risky token extensions while maintaining user trust.

[L-05] Not clearing debt when `amount_delta_usd` == 0

`try_realize_account_pnl_and_update` incorrectly returns when `amount_delta_usd` == 0. It doesn't clear the account's margin debt or update total market debt even though the order was settled.

```rust
pub(crate) fn try_realize_account_pnl_and_update(
        &mut self,
        market: &mut PerpMarket,
        amount_delta_usd: i128,
    ) -> Result<()> {
        if amount_delta_usd == 0 {
            Ok(())
        } else {
            let available_susd = self.deposited_susd;
            let previous_debt = self.debt_usd;
```

The correct implementation can be seen here

[L-06] Function is missing

In InitializeProgram struct, there's the sol_oracle declared. But, contrary to the intended design, there's no function to change it if it misbehaves or returns incorrect prices. The expected `set_sol_oracle_definition` function is not implemented and as such, sol_oracle cannot be changed if need be.

```
// We want to log the price found in this definition account so we can
    // double-check SOLUSD. If
    // it isn't correct, we will have to execute the set_sol_oracle_definition
    // instruction to change
    // it.
    sol_oracle: OraclePrice<'info>,
```

[L-07] Improper validation of `susd_mint`

The `susd_mint` account is not validated to ensure it matches the intended mint stored in the core global configuration. This opens the possibility for an attacker to specify a different mint with higher decimals, which would deflate the `keeper_fee_buffer_usd` amount. This manipulation would reduce the amount of fees sent to the keeper and adversely affect the `validate_trade` function.

The following snippet highlights the vulnerable code:

```
susd_mint: Box<InterfaceAccount<'info, token_interface::Mint>>,
```

Within the `commit_order` function, the user is expected to provide the list of accounts that this instruction will interact with. The code should validate these accounts, including `susd_mint`, as follows:

```rust
#[derive(Accounts)]
#[event_cpi]
pub struct CommitOrder<'info> {
    trader: User<'info>,

    #[account(
        constraint = {
            require_eq!(
                perp_market.config.oracle_node_id,
                perp_oracle.definition_key(),
                ProgramErrorCode::MismatchedOracleNodeId
            );
            true
        },
    )]
    perp_market: Account<'info, PerpMarket>,

    #[account(
        constraint = {
            require_eq!(
                core_market.program_id,
                crate::ID,
                ProgramErrorCode::InvalidMarket
            );

            require_eq!(
                core_market.id,
                perp_market.seeds.market_id,
                ProgramErrorCode::InvalidCoreMarket
            );

            true
        }
    )]
    core_market: Account<'info, synthetix_core::state::CoreMarket>,

    #[account(
        mut,
        constraint = margin.order.is_none() @ ProgramErrorCode::OrderExists,
        address = margin.try_known_address(
            trader.account_id(),
            perp_market.seeds.market_id
        )?,
    )]
    margin: Account<'info, Margin>,

    perp_oracle: OraclePrice<'info>,

    susd_mint: Box<InterfaceAccount<'info, token_interface::Mint>>,

    #[account(
        constraint = {
            require_eq!(
                sol_oracle.definition_key(),
                global_config.sol_oracle_definition,
                ProgramErrorCode::MismatchedOracleNodeId
            );
            true
        }
    )]
    sol_oracle: OraclePrice<'info>,

    global_config: GlobalConfig<'info>,
```

```
        synthetix_core_program: Program<'info, synthetix_core::program::SynthetixCore
}
```

The `susd_mint` should be validated against the corresponding public key stored in the `SynthetixCoreConfig`:

```
#[account]
#[derive(Debug, Default, PartialEq, Eq, InitSpace)]
pub struct SynthetixCoreConfig {
    pub next_system_account_id: NextSystemAccountId,
    pub next_market_id: u64,
    pub susd: Pubkey,
    pub market: SystemMarketConfig,
    pub pool: SystemPoolConfig,
}
```

Without this validation, an attacker can:

1. Use a different mint with higher decimals.
2. Manipulate the value of `keeper_fee_buffer_usd`, causing incorrect fees to be sent to the keeper.
3. Distort calculations in `validate_trade`, leading to incorrect margin and liquidation values.
4. Exploit order settlement with manipulated values, resulting in significant financial losses to the protocol.

For example:

```
let scaled_keeper_buffer =
    SystemAmount::try_from_mint_amount(susd_mint, keeper_fee_buffer_usd)?;

let keeper_fee =
    utils::keeper::checked_settlement_fee
      (global_config, scaled_keeper_buffer, sol_price)
        .ok_or_else(|| error!(ProgramErrorCode::InvalidKeeperFee))?;
```

The manipulated value will affect order settlement, potentially causing substantial losses to the protocol.

The improper validation of `susd_mint` can:

1. Cause financial losses for keepers by under-calculating their fees.
2. Allow malicious actors to manipulate order settlement values.

Additionally, this issue could arise unintentionally due to misconfigured inputs from users.

To mitigate this issue:

1. Validate the `susd_mint` public key against the `susd` value stored in `SynthetixCoreConfig`:

```
require_eq!(
    susd_mint.key(),
    global_config.susd,
    ProgramErrorCode::InvalidMint
);
```

[L-08] `create_collateral_lock` only allows increasing the lock amount when the previous lock expires

In `CreateCollateralLock`, the constraint below is defined.

```
let UserCollateralLock {
                amount,
                unlock_slot,
            } = lock;

            let current_lock = &user_collateral.lock;

            require!(amount != 0, ProgramErrorCode::InvalidAmount);
            require!(
>
                 amount >= current_lock.amount && amount <= user_collateral_custody.am
                ProgramErrorCode::InvalidLockAmount
            );

            require!(
>               unlock_slot > Clock::get().unwrap().slot
                    && unlock_slot >= current_lock.unlock_slot,
                ProgramErrorCode::InvalidLockUnlockSlot
            );
```

Under these constraints, creating a new lock can overridde the current or a previous lock, provided the new lock amount is more than current, unlock slot to set is in the future and current unlock slot may be at the same time or in the past relative to unlock slot to set.

The second condition, about the unlock slots allows that if a lock is expired, i.e current unlock slot is in the past compared to clock slot, a new lock can be created, without withdrawing to reset the lock. The only caveat is the first condition, which is that the new amount to be locked must be greater than the previous amount locked.

As a result, a user wanting to reduce locked amount after the lock expires is unable to.

Recommend loosening the constraint in that aspect. If a lock is fully expired, allow changing amount in any direction.

[L-09] Missing update to `last_interaction_slot`

The `last_interaction_slot` field in the `UserAccount` structure is designed to track the last slot during which a user interacted with the protocol. This field is crucial for maintaining an accurate record of user activity and ensuring correct state transitions.

```
pub struct UserAccount {
    pub id: u64,
    pub admin: Ownable,
    pub last_interaction_slot: u64,
}
```

However, in the Perp Market program, this field is not updated during deposit or withdrawal operations, which are key interactions that modify the state of a user's margin account. The absence of these updates could lead to inconsistencies or unexpected behavior when the protocol relies on this field for time-sensitive operations or validations.

Recommendations:

Update the `last_interaction_slot` field in the `UserAccount` structure during every user interaction that modifies their state, including but not limited to:

- Deposit operations.
- Withdrawal operations.

[L-10] Keepers can frontrun the trader order cancellation

The current design of the order cancellation functionality creates a scenario where the trader cannot effectively cancel their orders without being front-run by keepers. Keepers are incentivized to cancel orders before traders because

they receive cancellation fees. This behavior renders the trader's cancellation functionality insufficient and may result in financial losses for the trader. Specifically:

1. If a trader attempts to cancel their order to avoid keeper fees, a keeper can front-run the trader's transaction and cancel the order first.
2. This increases the trader's debt since they must pay additional fees to the keeper.

The following code snippets illustrate the issue:

Trader's Cancellation Function.

The trader can cancel an order through the following function:

```
pub fn cancel_order_trader(
    CancelOrderTrader {
        trader: _,
        perps_account: _,
        global_config,
        margin,
        market,
        perp_oracle,
        synthetix_core_program: _,
        event_authority: _,
        program: _,
    }: &mut CancelOrderTrader,
) -> Result<OrderCanceled> {
    let commitment_time = utils::order::validate_order_cancellation(
        global_config,
        margin,
        &market.perp,
        perp_oracle.price(),
    )?
}
```

Keeper's Cancellation Function with Additional Fee Charging. Keepers are incentivized to cancel orders through a separate function:

```rust
pub fn cancel_order_keeper<'info>(
    CancelOrderKeeper {
        perps_account: _,
        keeper_margin_management,
        sol_oracle,
        synthetix_core_program: _,
        event_authority: _,
        program: _,
    }: &mut CancelOrderKeeper<'info>,
    remaining_accounts: &'info [AccountInfo<'info>],
) -> Result<OrderCanceled> {

    let commitment_time = utils::order::validate_order_cancellation(
        global_config,
        margin,
        &keeper_margin_management.market_usd.market.perp,
        keeper_margin_management.market_usd.perp_oracle.price(),
    )?;

    let keeper_fee = utils::keeper::checked_cancellation_fee
      (global_config, sol_oracle.price())
        .ok_or_else(|| error!(ProgramErrorCode::InvalidKeeperFee))?;

    // Mint susd to the keeper.
    keeper_margin_management.market_usd.mint(
        remaining_accounts,
        keeper_fee.try_to_mint_amount
          (&keeper_margin_management.market_usd.susd_mint)?,
    )?;
}
```

Impact:

- **Financial Loss:** Traders may incur unnecessary fees if their cancellation attempts are front-run by keepers.
- **Reduced Utility:** The trader's cancellation functionality becomes effectively unusable.
- **Unfair Advantage to Keepers:** Keepers can exploit their role to generate fees at the trader's expense.

To address this issue:

1. **Introduce a Time Interval for Trader Cancellation:**
   Implement a specific time window during which only the trader can cancel their order. For example:

   ```rust
   let trader_only_window = global_config.trader_cancel_window;
   if current_time < commitment_time + trader_only_window {
       return Err(error!(ProgramErrorCode::TraderOnlyWindowActive));
   }
   ```

2. **Enhance Validation Logic:**
   Ensure that the `validate_order_cancellation` function prioritizes trader

cancellation within the defined window and rejects keepers' cancellation attempts during this period.

[L-11] Missing confidence interval validation

The protocol's oracle integration currently ignores the confidence interval provided by Pyth price feeds in `PriceFeedMessage`:

```
pub fn try_account_to_price(
    oracle_account: &AccountInfo,
    feed_id: &[u8; 32],
    use_ema: bool,
) -> Result<(i128, PublishTime)> {
    let msg = try_read_price_feed_message(oracle_account)?;
    // ...
    let feed_price = i128::from
      (if use_ema { msg.ema_price } else { msg.price });
```

The protocol reads from `PriceFeedMessage` which contains:

```
pub struct PriceFeedMessage {
    feed_id: [u8; 32],
    price: i64,
    conf: u64,         // Confidence interval being ignored
    ema_price: i64,
    ema_conf: u64,     // EMA confidence also ignored
    // ...
}
```

While the protocol correctly validates:

- Feed ID matching
- Price not being zero
- Price staleness

It does not validate the confidence values (`conf` and `ema_conf`) which could lead to:

- Using prices during high uncertainty periods
- Missing market anomaly signals
- Potential incorrect liquidations when market confidence is low (liquidations is just an example, this is applicable on all other parts where a valid price is crucial)

Simple understanding of confidence values

> "In scientific and engineering fields, an observation or measurement is almost always accompanied by a measurement uncertainty. Measurement uncertainty indicates how much precision or uncertainty there is in a single observation…the observer's best estimate of how far off from the 'true' value their measurement is likely to be."
>
> The distance between two points might be measured to be 10.12m ± 0.01m. The time it took for a car to travel that distance might be 1.23s ± 0.05s. In the context of the Pyth Network, a BTC/USD price feed might read $19,405.19 ± $9.907, indicating high conviction in the aggregate price with less than $10 of uncertainty ($9.907/$19,405.19 ≈ 0.05%).

For more on this topic and how Pyth differs from other Oracle networks:

Pyth Docs:confidence-intervals

Confidence Interval Case Studies

Add confidence validation to price reading, an example:

```rust
pub fn try_account_to_price(
    oracle_account: &AccountInfo,
    feed_id: &[u8; 32],
    use_ema: bool,
    max_confidence_pct: u64,  // Added parameter
) -> Result<(i128, PublishTime)> {
    let msg = try_read_price_feed_message(oracle_account)?;

    // Validate confidence based on which price we're using
    let (price, conf) = if use_ema {
        (msg.ema_price, msg.ema_conf)
    } else {
        (msg.price, msg.conf)
    };

    // Check if confidence exceeds maximum allowed percentage of price
    let confidence_pct = (conf as u128)
        .checked_mul(100)?
        .checked_div(price.abs() as u128)?;
    require!(
        confidence_pct <= max_confidence_pct as u128,
        ProgramErrorCode::PriceConfidenceTooHigh
    );

    // Continue with existing price scaling...
    let feed_price = i128::from(price);
    // ...
}
```

Consider using tighter bounds for critical operations like liquidations