BULLET

Bullet Exchange

Security Assessment Report

Version: 2.2

July, 2025

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Bullet Exchange. The review focused solely on the security aspects of the Rust implementation of components in scope, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the components in scope. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Bullet components contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Bullet components in scope.

Overview

Bullet is a Rust-native, Sovereign SDK-powered Layer 2 blockchain architected from the ground up for institutional-grade trading.

At the heart of the network sits Bullet Exchange, a fully integrated perpetuals trading and lending market designed to rival the performance and UX of the world's top CEXs.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the zetamarkets/sov-rollup repository.

The scope of this time-boxed review was strictly limited to files in the `crates/exchange/` directory (excluding `crates/exchange/src/api/`), assessed at commit 9ea6031.

The implementation of fixes for the identified issues was assessed at commit 0d5756c.

Additionally, pull requests #382, #374, #371 and #357 were also reviewed.

Note: third party libraries and dependencies were excluded from the scope of this assessment.

Approach

The security assessment covered components written in Rust.

The manual review focused on identifying issues associated with the business logic implementation of the components in scope. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Rust language.

Additionally, the manual review process focused on identifying vulnerabilities related to known Rust anti-patterns and attack vectors, such as unsafe code blocks, integer overflow, floating point underflow, deadlocking, error handling, memory and CPU exhaustion attacks, and various panic scenarios including index out of bounds, `panic!()`, `unwrap()`, and `unreachable!()` calls.

To support the Rust components of the review, the testing team also utilised the following automated testing tools:

- Clippy linting: `https://doc.rust-lang.org/stable/clippy/index.html`
- Cargo Audit: `https://github.com/RustSec/rustsec/tree/main/cargo-audit`
- Cargo Outdated: `https://github.com/kbknapp/cargo-outdated`
- Cargo Geiger: `https://github.com/rust-secure-code/cargo-geiger`
- Cargo Tarpaulin: `https://crates.io/crates/cargo-tarpaulin`

Output for these automated tools is available upon request.

Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 33 issues during this assessment. Categorised by their severity:

- Critical: 3 issues.
- High: 11 issues.
- Medium: 12 issues.
- Low: 4 issues.
- Informational: 3 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Bullet components in scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the components, including optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
BULL-01	Premature Access To Unsettled PnL During `USDC` Withdrawals	Critical	Resolved
BULL-02	Unrealised Profits Can Offset Losses In Conservative Risk Calculations	Critical	Resolved
BULL-03	Excessive Perp Liquidation Penalty Worsens Liquidatee Health	Critical	Closed
BULL-04	Denial Of Service (DoS) Issues Due To Expensive Vector Operations	High	Closed
BULL-05	Cancelled TPSLs Are Re-Queued Without Restoring Their State	High	Resolved
BULL-06	TPSL Order With Invalid Price Allows Griefing and Market Disruption	High	Resolved
BULL-07	Insufficient Order Cleanup On Perp Market Updates Breaks Liquidation Flow	High	Resolved
BULL-08	Liquidation Threshold Mismatch Enables Spot Liquidation Evasion	High	Closed
BULL-09	Precision Loss In `transfer()`	High	Resolved
BULL-10	Large State Objects May Cause Denial Of Service	High	Resolved
BULL-11	Overestimated Margin Due To Ignored `reduce_only` Flag In Loss Calculation	High	Resolved
BULL-12	Oracle Price Staleness Check Lacks Grace Period For Minor Update Delays	High	Resolved
BULL-13	Insufficient Liquidity In USDC Market Can Halt Core Protocol Functions	High	Closed
BULL-14	Missing Access Control in Order and TPSL Cancellation Functions	High	Resolved
BULL-15	Spot Liquidation Fails If Insurance Fund Cannot Cover Full Liquidator Reward	Medium	Closed
BULL-16	User Set Max Leverage Unnecessarily Restricts Withdrawals	Medium	Resolved
BULL-17	ADL May Overfill Positions By Using Outdated Position Size Snapshot	Medium	Resolved
BULL-18	Incorrect ADL Fill Price For User With Negative Equity	Medium	Resolved
BULL-19	Auto Deleverage Can Be Frontrun	Medium	Closed
BULL-20	Misleading Success In `handle_withdraw_msg()`	Medium	Resolved
BULL-21	Stale State for Counterparties in Auto Deleverage (ADL)	Medium	Resolved
BULL-22	Logic Issues In `find_risk_increasing_order_ids()`	Medium	Resolved
BULL-23	Incorrect Asset Sorting with Preferred Asset Priority	Medium	Resolved
BULL-24	Incorrect Impact Price Calculation Due To Strict Inequality	Medium	Resolved

BULL-25	Premature Termination Of Batch Mark Price Updates On Single Market Error	Medium	**Resolved**
BULL-26	PNL Pool Carries Unbacked Gains from Insolvent Liquidatees in Perp Liquidations	Medium	**Closed**
BULL-27	Incorrect Hourly Check In `update_funding_rate()` Allows Premature Updates	Low	**Closed**
BULL-28	`queue_tpsls_for_execution_for_market()` May Be Called With Stale Prices	Low	**Resolved**
BULL-29	Missing Functionality For Rotating Admins	Low	**Resolved**
BULL-30	Missing Price Validation	Low	**Resolved**
BULL-31	No Support For Tokens With Large Supply	Informational	**Closed**
BULL-32	Duplicate Liquidation Checks During Perp Liquidations	Informational	**Resolved**
BULL-33	Outdated Dependencies	Informational	**Closed**

BULL-01	Premature Access To Unsettled PnL During USDC Withdrawals		
Asset	protocol/handlers.rs		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Users can bypass PnL settlement and withdraw unbacked `USDC` profits, exposing the system to financial inconsistencies and potential fund imbalances.

The `handle_withdraw_msg()` function allows users to withdraw assets; however, if the asset being withdrawn is `USDC`, a vulnerability exists which allows users to prematurely access their `unsettled_perp_profits`, bypassing the standard PnL settlement mechanisms. This can lead to withdrawal of funds that are not yet confirmed or backed by the `USDC` PnL pool.

The issue stems from the interaction between how `actual_withdraw_amount` is calculated, and the presumed behaviour of `process_balance_update_for_user_helper()` for `USDC` withdrawals:

protocol/handler.rs::handle_withdraw_msg

```
  pub(crate) fn handle_withdraw_msg(
    &mut self,
    asset_id: AssetId,
    requested_amount: PositiveDecimal,
    context: &Context<S>,
    state: &mut impl TxState<S>,
) -> color_eyre::Result<()> {
    // ...

    let actual_withdraw_amount = requested_amount
        .min(withdrawable_token_amount)
        .min(spot_ledger.asset);

    {
        let borrow_lend_market = borrow_lend_markets
            .get_mut(&asset_id)
            .ok_or(ExchangeError::BorrowLendMarketNotFound { asset_id })?;

        process_balance_update_for_user_helper(
            &mut user_account,
            borrow_lend_market,
            actual_withdraw_amount.neg(),
        )?;
    }

    self.transfer(
        TransferDirection::VaultToUser,
        asset_id,
        user_address,
        actual_withdraw_amount,
        state,
    )?;

    // ...
}
```

When the user is withdrawing `USDC`, the `process_balance_update_for_user_helper()` function calls

`UsdcLedger::apply_balance_update()` to deduct the withdrawal amount from the user's `unsettled_perp_profit`, before touching their actual `USDC` asset balance. This allows the user to bypass the `USDC` PnL pool settlement process.

user/spot_ledger.rs::UsdcLedger::apply_balance_update

```
Ordering::Less => {
  let liability_to_update = PositiveDecimal::try_from(balance_delta.abs())?;
  // @audit this deducts from unsettled_perp_profit before touching assets
  let asset_deduction = self.deduct_from_real_asset(liability_to_update)?;
  let liability_added = liability_to_update.try_sub(asset_deduction)?;
  self.ledger.add_to_liability(liability_added)?;
  Ok(BalanceUpdateDeltas {
      deducted_amount: asset_deduction,
      added_amount: liability_added,
  })
}
```

This vulnerability has a high impact as it allows bypassing a core financial mechanism (PnL settlement), potentially leading to withdrawal of unrealised or unbacked profits. The likelihood is high because any user with unsettled `USDC` profits and a nominal amount of `USDC` in their spot balance can perform this operation.

Recommendations

Rework withdrawals so that they only come from settled spot balances and ensure that any `unsettled_perp_profit` is not directly withdrawable until it has been settled.

Resolution

This issue was resolved in PR #389 by only allowing withdrawals from spot balances. Additionally, margin checks were also modified to only occur on real assets.

BULL-02	Unrealised Profits Can Offset Losses In Conservative Risk Calculations		
Asset	`user/account.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Unrealised losses are understated due to profit netting, allowing users to over-borrow or over-withdraw, while increasing the protocol's exposure to bad debt.

The `calculate_total_perp_unrealized_loss()` method is intended to provide a conservative estimate of unrealised losses by aggregating only negative PnLs from perpetual positions. However, it currently relies on the output of `calculate_total_perp_unrealized_pnl()`, which nets both profits and losses across all positions. As a result, profitable positions can offset losses, leading to an understated representation of total risk.

For example, if one position has a `-50` PnL and another has `+100`, the method may return `0` instead of correctly capturing the `-50` unrealised loss. This undermines conservative risk assessments.

user/account.rs::calculate_total_perp_unrealized_loss

```
fn calculate_total_perp_unrealized_loss(
    &self,
    margin_calculation_state: &MarginCalculationState,
) -> Result {
    // @audit this implementation still allows profits to offset losses
    Ok(Decimal::ZERO.min(self.calculate_total_perp_unrealized_pnl(margin_calculation_state)?))
}
```

This flawed `calculate_total_perp_unrealized_loss()` impacts critical risk management functions:

1. When `calculate_account_equity()` is called with `conservative = true` (e.g., in `handle_borrow_spot_msg` via `calculate_available_margin()`), the understated loss figure leads to an overestimation of the account's equity and available margin. This could allow users to borrow more spot assets than is safe according to conservative risk parameters.
2. The `calculate_total_perp_unrealized_loss()` is also used unconditionally (without a `conservative` flag in its call path) within `calculate_used_margin()`, which is a component of `calculate_withdrawable_amount_of_asset()`. An understated loss (a value closer to zero, or zero when it should be negative) results in a lower calculated "used margin," thereby overstating the amount of collateral a user can withdraw.

In both scenarios, the system misjudges the user's true risk profile under conservative assumptions, potentially allowing for excessive leverage through borrowing or an unsafe reduction in collateral via withdrawals.

For example, if a user has two perp positions:

- Position A: +100 PNL
- Position B: -50 PNL

`calculate_total_perp_unrealized_pnl()` would return `+50`.
`calculate_total_perp_unrealized_loss()` (current implementation) would then calculate `min(0, 50)`, resulting in `0`.
A truly conservative approach for `calculate_total_perp_unrealized_loss()` should sum only actual losses from each position, yielding `-50`.

This issue is classified as high impact. The flawed calculation of unrealised losses directly weakens key risk controls for spot borrowing and collateral withdrawal. By allowing users to borrow more or withdraw more collateral than a truly conservative assessment would permit, the protocol is exposed to an increased risk of accumulating bad debt if these users' positions subsequently deteriorate. The likelihood is high as it affects any user with multiple perpetual positions (some profitable, some not) when they interact with spot borrowing or attempt to withdraw collateral, common operations in a margin trading system.

Recommendations

Change the `calculate_total_perp_unrealized_loss()` function to sum only the negative PnL values (actual losses), while treating any positive PnL values (profits) as zero.

Resolution

This issue was fixed in #399 by using the sum of losses, without netting the profits.

BULL-03	Excessive Perp Liquidation Penalty Worsens Liquidatee Health		
Asset	`protocol/handlers.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Liquidated users can be overcharged beyond intended fees, resulting in excessive and compounding financial losses during perpetual liquidations.

In the current implementation of perpetual liquidations within `handle_liquidate_perp_positions_msg()`, the amount debited from the liquidated user as a penalty, `actual_liquidation_loss`, can significantly exceed the `total_liquidator_reward`, which represents the fee owed to the liquidator for executing the liquidation.

This discrepancy arises when the liquidatee's weighted account equity, following the closure of selected positions, is greater than the computed `total_liquidator_reward`. As a result, the excess amount is removed from the account without a clearly justified redistribution, potentially leading to unfair penalisation.

protocol/handler.rs::handle_liquidate_perp_positions_msg

```
let weighted_account_equity = liquidatee_account.calculate_account_equity(
    false,
    EquityType::Weighted(MarginType::Maintenance(None)),
    &margin_calculation_state,
)?;

let actual_liquidation_loss = weighted_account_equity
    .max(total_liquidator_reward.as_dec())
    .min(max_liquidation_loss.as_dec());

process_pnl_and_rebalance_for_user_helper(
    &mut liquidatee_account,
    usdc_borrow_lend_market,
    &mut usdc_pnl_pool,
    Some(actual_liquidation_loss.neg()),
)?;

process_pnl_and_rebalance_for_user_helper(
    &mut liquidator_account,
    usdc_borrow_lend_market,
    &mut usdc_pnl_pool,
    Some(total_liquidator_reward.as_dec()),
)?;
```

The `actual_liquidation_loss` taken from the liquidatee is determined by the following formula:

$$\text{actual\_liquidation\_loss} = \min(\max(\text{weighted\_account\_equity}, \text{total\_liquidator\_reward}), \text{max\_liquidation\_loss})$$

Since this takes the maximum of the weighted account equity and the total liquidator reward, the liquidatee can lose a substantial part of their remaining account equity, up to the maintenance margin of the positions chosen for liquidation, even if the actual `total_liquidator_reward` (fee) is much smaller.

This excessive penalty can make the liquidatee's account *more* unhealthy relative to their remaining positions. By significantly reducing their equity, their margin ratio can deteriorate, potentially making them immediately eligible for another round of liquidations. This is counterintuitive to the goal of liquidation, which should ideally stabilise an unhealthy account or close it out, not push it into a worse state that invites cascading liquidations.

The impact is classified as high because it directly causes liquidatee accounts to lose more funds than potentially intended by the fee structure, and can worsen their account health, leading to further liquidations. This constitutes a direct and potentially substantial financial loss for affected users beyond the standard fee, even if the system itself does not lose funds. The likelihood is high, as this calculation is applied in every perp liquidation where the liquidatee's relevant weighted equity exceeds the base liquidator reward.

Recommendations

Modify the calculation and application of the liquidation penalty in `handle_liquidate_perp_positions_msg()`. The amount debited from the liquidatee specifically for the liquidation event (distinct from their position's PnL) should be strictly `total_liquidator_reward`.

Resolution

The finding has been closed with the following rationale provided by the development team:

> *"During backstop liquidation the entire account is up for grabs, so the calculation is entirely intended. If the liquidation loss is larger than the account equity, you essentially want to take the entire accounts equity.*
> *In the future, any excess equity that is taken will be sent to the protocol."*

BULL-04	Denial Of Service (DoS) Issues Due To Expensive Vector Operations		
Asset	`*.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

Inefficient vector operations can be exploited to trigger denial of service by growing storage cheaply, and forcing costly removals that exceed gas or compute limits.

The codebase contains multiple instances of expensive vector operations, such as `.remove()` and `.shift_remove()`, which are repeatedly called within loops. These operations have linear time complexity $O(n)$, unlike item insertion, which occurs in constant time.

This imbalance allows an attacker to exploit the system by continuously adding entries to the vectors at low cost. Over time, the growing size of these vectors will cause subsequent removal operations to exceed computational or gas limits, ultimately resulting in transaction failures and denial of service conditions.

The testing team noted these issues in the following places:

- In the `queue_tpsls_for_execution_for_market()` function, a loop iterates over all TPSL orders for a given `market_id` and `tpsl_price_condition` and checks if they should be queued for execution. Given a large number of TPSLs this function will not be executable. Seeing as `queue_tpsls_for_execution_for_market()` is called after updating oracle and mark prices and after placing orders, this could DoS large parts of the platform.

  protocol/primitives.rs::queue_tpsls_for_execution_for_market

```
pub(crate) fn queue_tpsls_for_execution_for_market(
    &mut self,
    market_id: MarketId,
    tpsl_price_condition: TpslPriceCondition,
    prev_trade_execution: Option,
    current_timestamp: UnixTimestampMillis,
    state: &mut impl TxState,
) -> Result<()> {

// ...

for tpsl_order_id in tpsl_order_ids_to_check {
  // ...
}
```

- The `cancel_order()` function iterates through all orders in a book level and removes a specific order. Both iteration and removal exhibit $O(n)$ complexity. As a result, if an attacker places many orders at the same price level, DoS issues can occur. Since force cancelling orders and liquidating positions both require the cancellation of orders, the impact can be significant.

matching_engine/orderbook.rs::cancel_order

```
pub(crate) fn cancel_order(&mut self, order: &Order) -> Result<()> {
    // ...
    let book_level = book_side_to_cancel.get_mut(&order.price).ok_or(
        // ...
    )?;

    // @audit O(N) search for the order's index
    let index_to_remove = book_level
        .order_ids // This is a VecDeque
        .iter()
        .position(|&id| id == order.order_id)
        .ok_or(ExchangeError::OrderIdNotFoundInPriceLevel {
            // ...
        })?;

    // @audit O(N) removal in worst-case for VecDeque
    book_level.order_ids.remove(index_to_remove);
    // ...
    Ok(())
}
```

- A similar pattern occurs in `try_remove_tpsl_from_buffer()` where a TPSL is searched in `tpsl_order_ids_to_execute` by iteration and later removal. Given a large amount of TPSL orders this may cause DoS issues.

protocol/primitives.rs::try_remove_tpsl_from_buffer

```
pub(crate) fn try_remove_tpsl_from_buffer(
    // ...
) -> Result<()> {
    let mut tpsl_buffer = self.get_tpsl_order_ids_to_execute_required(market_id, state)?;
    // @audit O(N) search for the tpsl_order_id's index
    let buffer_index = tpsl_buffer.iter().position(|x| *x == tpsl_order_id);

    if let Some(i) = buffer_index {
        // @audit O(N) removal in worst-case for VecDeque
        tpsl_buffer.remove(i);
        // ...
    }
    Ok(())
}
```

- The `calculate_margin_requirement()` function iterates through all orders on the users' orderbook to calculate the total potential loss. Since adding an order can be done in constant time, this function is vulnerable to DoS attacks.
- The `cancel_tpsl()` function uses `shift_remove()` to remove a TPSL from `exchange_tpsl_order_ids_by_price_condition`, which also has $O(n)$ complexity and, as such, is vulnerable to DoS attacks.
- The `cleanup_exchange_tpsls_for_market()` function iterates through all TPSL orders for a given market and checks if they should be deleted; if so, they are cancelled. In the worst case scenario, where all TPSL are marked for deletion, this exhibits $O(n^2/2)$ complexity. Seeing as placing a TPSL can be done in constant time, this function is susceptible to DoS attacks.

protocol/primitives.rs::cleanup_exchange_tpsls_for_market

```
pub(crate) fn cleanup_exchange_tpsls_for_market(
    &mut self,
    market_id: MarketId,
    state: &mut impl TxState,
) -> Result<()> {
    let tpsl_order_ids_for_market = self.get_all_tpsl_order_ids_for_market(market_id, state)?;

    // ...

    let mut tpsls_to_delete = Vec::new();
    for tpsl_order_id in tpsl_order_ids_for_market {
        let tpsl = self.get_tpsl_required(tpsl_order_id, state)?;

        // ...

        if should_delete_tpsl(tpsl, &position, self, state)? {
            tpsls_to_delete.push(tpsl_order_id);
        }
    }

    for tpsl_order_id in tpsls_to_delete {
        self.cancel_and_unlink_tpsl(tpsl_order_id, state)?;
    }

    Ok(())
```

- The `remove_order()` function uses `shift_remove()`, which runs in $O(n)$ time. This function is used when processing maker fills. As such, DoS issues may occur when filling orders.

user/perp_ledger.rs::remove_order

```
pub(crate) fn remove_order(&mut self, order_id: OrderId) -> Result<()> {
    if self.orders.shift_remove(&order_id).is_none() {
        return Err(ExchangeError::OrderNotFoundInUserState { order_id });
    }

    Ok(())
}
```

Recommendations

Optimise the data structures or use more efficient removal strategies that avoid linear scans and shifting.

Alternatively, in places where this is impractical, consider placing limits on the amount of items that can be processed in a single call.

Resolution

The finding has been closed with the following rationale provided by the development team:

> *"This is not addressed yet. A state access refactor will be done, but cannot be done now due to other priorities."*

BULL-05	Cancelled TPSLs Are Re-Queued Without Restoring Their State		
Asset	`protocol/primitives.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

Take Profit/Stop Loss (TPSL) orders are re-queued without being reactivated in state, allowing attackers to create stuck orders that cause repeated execution failures and platform-wide denial of service.

In the `execute_tpsls_from_buffer_for_market()` function, a TPSL order is cancelled and removed from state before execution. If the order is subsequently not filled, it is re-added to the `tpsl_buffer` for future retries. However, it is not reinstated into the state as active, resulting in execution failure during the next attempt, as the TPSL no longer exists in the state.

A malicious actor could exploit this by placing a limit TPSL with an unreasonably low bid price, ensuring it will not be filled. This causes the order to be repeatedly re-queued without being executable. Since `execute_tpsls_from_buffer_for_market()` is invoked during price updates and order placements, these failures can cascade, leading to denial of service (DoS) conditions across core parts of the platform.

protocol/primitives.rs::execute_tpsls_from_buffer_for_market

```
fn execute_tpsls_from_buffer_for_market(
    &mut self,
    market_id: MarketId,
    current_timestamp: UnixTimestampMillis,
    num_tpsls_to_execute: usize,
    state: &mut impl TxState,
) -> Result<(usize, TradeExecution)> {

        // ...

        // Remove this tpsl from everywhere in state that might use it because it is now being executed
        // This is done so we don't accidentally overwrite or read stale state etc;
        let linked_tpsl_order_ids = self.cancel_tpsl(tpsl.tpsl_order_id, state)?;
        cancelled_tpsl_info.insert(tpsl.tpsl_order_id, linked_tpsl_order_ids);

        // ...

        // If it didn't trade, push it back into the buffer because we want to try again next time
        if local_min_trade_price.is_none() && local_max_trade_price.is_none() {
            tpsl_buffer.push_back(tpsl_order_id); //@audit marked for re-execution although it was cancelled
            continue;
        }

        // ...
}
```

Recommendations

Ensure the TPSL orders are always reinserted into state when it is marked for re-execution.

Resolution

This issue was resolved in PR #371 by refactoring `execute_tpsls_from_buffer_for_market()`.

BULL-06	TPSL Order With Invalid Price Allows Griefing and Market Disruption		
Asset	`protocol/args.rs, protocol/primitives.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The system fails to validate the `order_price` of Take Profit/Stop Loss (TPSL) orders against the market's `min_tick_size` during order creation. As a result, TPSL orders submitted with invalid prices repeatedly fail upon triggering. Rather than being cancelled, these orders are continuously re-queued in the `tpsl_buffer`.

This leads to unnecessary resource consumption for the originating user and exposes the platform to a denial of service risk. A malicious actor could deliberately flood the TPSL execution buffer with such invalid orders, obstructing the processing of legitimate TPSL orders from other users. This undermines automated risk management mechanisms, especially during periods of high volatility, and may cause substantial financial losses. To mitigate this, TPSL orders should be validated against `min_tick_size` at the time of creation, and invalid entries should be rejected or automatically cancelled if they cannot be executed.

The issue unfolds as follows:

1. The `PlaceTpslOrderArgs::validate()` method in `protocol/args.rs` checks if `order_price` is zero, but does not ensure it aligns with the market's `min_tick_size`.

   protocol/args.rs::PlaceTpslOrderArgs::validate

   ```
   impl PlaceTpslOrderArgs {
       pub fn validate(&self) -> Result<()> {
           // ...
           // @audit this doesn't validate that the ORDER price is multiple of tick size
           if self.order_price.is_zero() || self.trigger_price.is_zero() {
               return Err(ExchangeError::ZeroPrice);
           }
           Ok(())
       }
   }
   ```

2. In `protocol/primitives.rs`, the `create_tpsls()` function validates the `trigger_price` and `size`, but not the `order_price` that will be used when the TPSL converts into an actual order.

   protocol/primitives.rs::create_tpsls

   ```
   for tpsl in &new_tpsls {
       perp_market.validate_price_and_lot_tick_size(&tpsl.trigger_price, &tpsl.size)?;
   }
   ```

3. When a TPSL with an invalid `order_price` is triggered, `execute_tpsls_from_buffer_for_market()` in `primitives.rs` attempts to place an order. This fails internally within `place_order()` due to the `order_price` not adhering to the `min_tick_size`.

4. The error handling in `execute_tpsls_from_buffer_for_market()` does not flag `ExchangeError::InvalidPriceTickSize` (among others) as a reason to cancel the TPSL.

protocol/primitives.rs::execute_tpsls_from_buffer_for_market

```
            Err(e) => {
                let should_cancel = // ... conditions that don't include InvalidPriceTickSize ...
                // ...
                if should_cancel {
                   // ... cancel TPSL ...
                } else {
                    // @audit TPSL is reinserted if place_order fails with an error not in should_cancel
                    self.reinsert_tpsl_for_user(&tpsl.owner, tpsl.clone(), state)?;
                    cancelled_tpsl_info.shift_remove(&tpsl.tpsl_order_id);
                }
            }
        }
        // ...
        if local_min_trade_price.is_none() && local_max_trade_price.is_none() {
            // @audit The TPSL whose place_order() failed is pushed back to the local buffer here
            tpsl_buffer.push_back(tpsl_order_id);
            continue;
        }
// ...
// @audit The (potentially modified) local tpsl_buffer is written back to state
self.tpsl_order_ids_to_execute
    .set(&market_id, &tpsl_buffer, state)
    .map_err(|e| ExchangeError::StateAccessError(Box::new(e)))?;
```

The failing TPSL is reinserted into general TPSL tracking structures via `reinsert_tpsl_for_user()` and then re-added to the end of the `tpsl_buffer` being processed. This buffer (now containing the re-queued invalid TPSL) is then saved back to the state's `tpsl_order_ids_to_execute` for that market.

An attacker may exploit this vulnerability by submitting a large number of invalid TPSL orders. These orders clog the execution buffer for a specific market and, because `execute_tpsls_from_buffer_for_market()` only processes a limited number of entries per call (e.g., `MAX_TPSLS_TO_EXECUTE = 50`), they can occupy all available slots if their trigger conditions continue to be met.

This tactic effectively prevents legitimate TPSLs from being processed in a timely manner, resulting in a denial of service condition for affected users. During periods of high volatility, this disruption can cause critical stop-loss orders to be delayed or entirely missed, leading to significant financial losses.

The impact is high as this exploit can lead to direct financial loss for users by disabling a key risk management feature, and constitutes a market disruption. The likelihood is medium, as while it requires specific knowledge and intent for malicious exploitation, the underlying condition can also be met accidentally, and the cost of spamming might not be prohibitive for a motivated attacker.

Recommendations

Implement strict validation of the `order_price` within `PlaceTpslOrderArgs` against the respective `PerpMarket`'s `min_tick_size`.

This validation should occur in the `protocol/primitives.rs::create_tpsls()` function, before a `Tpsl` object is created and persisted.

Resolution

This issue was fixed in PR #400 by validating that a TPSL follows the minimum tick size. Additionally, price heuristics were implemented to ensure TPSL prices are not unreasonably far away from the mark price.

BULL-07	Insufficient Order Cleanup On Perp Market Updates Breaks Liquidation Flow		
Asset	`protocol/handlers.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

Reinitialising a `PerpMarket` deletes its order book without cancelling existing orders, leaving behind orphaned orders that block liquidations.

The `handle_update_perp_market_msg()` function creates a new `PerpMarket`, which includes initialising a new orderbook. However, this process deletes the existing orderbook without properly cancelling the existing orders. As a result, those orders continue to persist in other structures such as `orders` and `user_containers.perp_order_ids`.

When attempting to cancel these lingering orders via `cancel_order()`, the operation fails because `perp_market.orderbook.cancel_order(&order)` cannot find the corresponding order in the newly created orderbook. Seeing as `force_cancel_all_orders_for_user()` is called during perp liquidations, any users who still have these lingering orders can not be liquidated.

protocol/handlers.rs::handle_update_perp_market_msg

```
pub(crate) fn handle_update_perp_market_msg(
    &mut self,
    market_args: PerpMarketArgs,
    context: &Context,
    state: &mut impl TxState,
) -> color_eyre::Result<()> {
    self.validate_admin(AdminType::Protocol, context.sender(), state)?;

    let market_id = market_args.market_id;
    self.get_perp_market_required(market_id, state)?;
    let perp_market = PerpMarket::create_from_args(market_args)?; //@audit new orderbook is created here
    self.perp_markets.set(&market_id, &perp_market, state)?;

    self.emit_event(state, Event::UpdateMarket { market_id });
    Ok(())
}
```

Recommendations

Change implementation to ensure that, before replacing the existing `PerpMarket`, all open orders are properly cancelled.

Alternatively, keep the existing orderbook such that all previous orders remain valid in the new `PerpMarket`.

Resolution

This issue was fixed in #365 by modifying `handle_update_perp_market_msg()` such that the existing market is updated instead of replaced with a new market.

BULL-08	Liquidation Threshold Mismatch Enables Spot Liquidation Evasion		
Asset	`protocol/handlers.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

Misaligned liquidation thresholds allow users to evade spot liquidation by holding small perp positions, creating a loophole that blocks enforcement against under collateralised accounts.

During a spot liquidation, the system checks whether the user holds any perpetual (perp) positions or open orders. If so, execution is reverted, requiring the account to first undergo perp liquidation. However, the conditions under which each type of liquidation can occur are misaligned.

Spot liquidations are permitted when `weighted_account_equity` is less than or equal to `total_perp_maintenance_margin_requirement`, whereas perp liquidations are only allowed when `weighted_account_equity` falls below `backstop_liquidation_threshold * total_perp_maintenance_margin_requirement`. Since `backstop_liquidation_threshold` is less than 1, spot liquidations can be triggered earlier, for comparatively healthier accounts.

This creates a grey area in which an account cannot be spot liquidated due to the presence of perp state, but also does not meet the stricter criteria for perp liquidation. Consequently, users can strategically acquire a minimal perp position to block spot liquidation and remain operational, despite being under collateralised.

protocol/handlers.rs::handle_liquidate_spot_liability_msg

```
if liquidatee_account.has_perps_state() {
    return Err(ExchangeError::CannotLiquidateSpotLiabilityOfUserWithPerpsState.into());
}
```

Recommendations

Adjust the liquidation thresholds and eligibility logic to eliminate this exploitable gap.

Resolution

The finding has been closed with the following rationale provided by the development team:

> *"This is a non-issue. Force closing a position is part of the liquidation process and that can happen as soon as someone is below maintenance margin requirement. It is also impossible for anyone to enter a perps position if they are at this point since they are below initial maintenance margin requirements which is always larger than maintenance margin."*

BULL-09	Precision Loss In `transfer()`		
Asset	`common/decimals/positive_decimal.rs, protocol/primitives.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

Users may be over-credited on token deposits due to a mismatch between decimal precision handling and integer-based accounting.

When depositing certain tokens, the `token_amount` is converted from a `PositiveDecimal` to a `u128` to comply with the Sovereign SDK Bank module, which represents token balances as integers. In contrast, the Bullet Exchange operates using decimal values. During this conversion, any precision beyond the specified `asset_metadata.decimals()` is truncated. However, this truncated precision is not reflected in the amount ultimately credited to the user, resulting in an inconsistency where users may receive slightly more than the underlying integer representation permits.

For example, given an asset that has 2 decimals, if `transfer()` is called with `amount = 0.009`, then the actual transferred amount is truncated to `0`. This means that, when depositing an asset, a user can be credited `0.009` assets, while not depositing any.

In the general deposit flow, this is protected against by disallowing an `amount` that is more precise than the asset's decimals. However, for `DepositToInsuranceFund` or `DepositToUsdcPnlPool` calls, this check is omitted and this issue can be exploited. Only USDC can be deposited using these calls, which usually has 6 decimals. As such, the lost value per deposit is low, and this issue can only be exploited if transaction fees are free for long periods of time.

protocol/primitives.rs::transfer

```
let coins = Coins {
    amount: sov_modules_api::Amount(
        token_amount
            .try_convert_integer_with_precision(asset_metadata.decimals().into())?, //@audit truncates after
      ↪  `asset_metadata.decimals()`
    ),
    token_id: asset_metadata.token_id(),
};
```

protocol/handlers.rs::handle_deposit_msg

```
if amount.as_dec().scale() > asset_metadata.decimals().into() {
    return Err(ExchangeError::TooManyDecimalPlaces {
        max_decimal_places: asset_metadata.decimals(),
        input: amount.as_dec(),
    }
    .into());
}
```

Recommendations

Check the decimals of `amount` during `DepositToInsuranceFund` or `DepositToUsdcPnlPool` calls, similar to the general deposit flow.

Resolution

This issue was resolved in PR #392 by validating the decimal precision during a deposit to the insurance fund and the PnL pool.

BULL-10	Large State Objects May Cause Denial Of Service		
Asset	`protocol/state.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

A malicious user can bloat a `PerpMarket`'s order book to the point where it exceeds gas limits, rendering the entire market unusable.

In the Sovereign SDK, each state value is serialised and stored as a whole object. For example, the `StateMap` `perp_markets` stores each `PerpMarket` as a single serialised unit. Because a `PerpMarket` includes its `Orderbook`, users can arbitrarily inflate its size by placing large numbers of orders.

This design becomes a liability when reading or writing to storage, as the entire `PerpMarket` must be fetched and deserialised in full. If the object grows too large, these operations may exceed gas limits, rendering the market unusable. A malicious actor could exploit this by flooding the order book, effectively denying access to key market functions such as price updates, order placement, and cancellations.

protocol/state.rs

```
#[state]
pub perp_markets: sov_modules_api::StateMap<MarketId, PerpMarket>,
```

protocol/state.rs::PerpMarket

```
pub struct PerpMarket {
    pub market_id: MarketId,
    pub is_active: bool,
    pub min_tick_size: PositiveDecimal,
    pub min_lot_size: PositiveDecimal,
    pub funding_parameters: FundingParameters,
    pub impact_margin: PositiveDecimal,
    pub orderbook: Orderbook,
    pub last_trade_price: PositiveDecimal,
}
```

Other potentially problematic state objects are `user_containers`, `tpsl_order_ids_to_execute` and `tpsl_order_ids_by_price_condition`.

Recommendations

Consider representing large, user-controllable objects directly with a `StateMap` or `StateVec` as these do not have size concerns.

Resolution

This issue was resolved in PR #365 by limiting the maximum size of an orderbook. Additionally, `tpsl_order_ids_to_execute` and `tpsl_order_ids_by_price_condition` are now stored in different, more manageable containers and may also have a maximum size limit in the future.

BULL-11	Overestimated Margin Due To Ignored `reduce_only` Flag In Loss Calculation		
Asset	`matching_engine/order.rs, user/perp_ledger.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

The `potential_loss()` function for an order does not consider whether an order is `reduce_only`. This leads to an overestimation of potential losses when calculating initial margin requirements, as `reduce_only` orders are treated as if they are opening new positions, rather than closing existing ones.

The `potential_loss()` function in `matching_engine/order.rs` calculates the loss an order might incur if the market moves against it before execution:

matching_engine/order.rs::potential_loss

```
pub fn potential_loss(&self, mark_price: PositiveDecimal) -> Result {
    let price_delta = match self.side {
        Side::Bid if self.price > mark_price => self.price.try_sub(mark_price)?,
        Side::Ask if self.price < mark_price => mark_price.try_sub(self.price)?,
        _ => return Ok(PositiveDecimal::ZERO),
    };

    calculate_notional(price_delta, self.remaining_size)
}
```

This calculated `potential_loss` is then used in `calculate_margin_requirement()` in `user/perp_ledger.rs` to determine a user's initial margin:

user/perp_ledger.rs::calculate_margin_requirement

```
// ...
    for order in self.orders.values() {
        orders_potential_loss =
            orders_potential_loss.try_add(order.potential_loss(mark_price)?)?; //@audit includes reduce-only orders
// ...
let initial_margin_requirement = leverage_table.calculate_initial_margin_requirement(
    total_notional,
    self.user_selected_max_leverage,
)?;

initial_margin_requirement.try_add(orders_potential_loss)
// ...
```

Because `reduce_only` orders are designed to decrease an existing position, they should not contribute to the `orders_potential_loss` in the same way as orders that increase or open new positions. This miscalculation results in higher initial margin requirements than necessary, which can unnecessarily restrict a user's ability to place further orders or utilise their available margin.

This issue is classified as medium impact because it affects core margin calculation logic and can prevent users from trading as intended, though it does not directly lead to fund loss. The likelihood is high as it affects any user placing `reduce_only` orders.

Recommendations

Modify `potential_loss()` to return zero if the order's `reduce_only` flag is true.

Resolution

This issue was resolved in PR #396 by correctly calculating the potential loss for `reduce_only` orders.

BULL-12	Oracle Price Staleness Check Lacks Grace Period For Minor Update Delays		
Asset	`pricing/oracle_prices.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The current oracle price staleness check in `validate_oracle_prices_up_to_date()` uses `pricing_frequency_seconds` as a strict upper bound for the age of an oracle price.

This can lead to critical protocol operations being halted, if oracle price updates experience even minor delays beyond this configured period, potentially impacting core functionalities, such as trading or liquidations.

pricing/oracle_prices.rs::validate_oracle_prices_up_to_date

```
pub fn validate_oracle_prices_up_to_date(
    &self,
    asset_id: AssetId,
    current_timestamp: UnixTimestampMillis,
) -> Result<()> {
    let last_update_timestamp = self
        .get_oracle_price_with_timestamp(asset_id)?
        .last_update_timestamp;

    let last_valid_timestamp = current_timestamp
        .as_secs()
        .checked_sub_secs(self.pricing_frequency_seconds as i64)?;

    // @audit If last_update_timestamp is None, map_or evaluates to true, correctly flagging stale.
    // @audit The condition t.as_secs() < last_valid_timestamp means an error occurs if:
    // @audit last_update_timestamp.as_secs() < current_timestamp.as_secs() - self.pricing_frequency_seconds
    // @audit which is equivalent to:
    // @audit current_timestamp.as_secs() - last_update_timestamp.as_secs() > self.pricing_frequency_seconds
    // @audit So, if the age of the price is strictly greater than pricing_frequency_seconds, it's an error.
    if last_update_timestamp.map_or(true, |t| t.as_secs() < last_valid_timestamp) {
        return Err(ExchangeError::OraclePriceNotUpToDate {
            asset_id,
            last_update_timestamp,
            current_timestamp,
        });
    }

    Ok(())
}
```

Recommendations

Introduce a new parameter `max_oracle_price_age_seconds` that is slightly larger than `pricing_frequency_seconds` to account for slight delays in updates.

Resolution

The issue was resolved in PR #401 by creating two types of pricing validity times.

BULL-13	Insufficient Liquidity In USDC Market Can Halt Core Protocol Functions		
Asset	`protocol/utils.rs, user/spot_ledger.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The `process_pnl_and_rebalance_for_user_helper()` function relies on borrowing from the `usdc_borrow_lend_market`. If this market lacks sufficient available liquidity (e.g., due to high utilization or insufficient deposits), the required borrow operations via `usdc_borrow_lend_market.borrow_from_pool()` will fail.

Such failures can propagate, disrupting essential protocol operations, such as applying funding, trade executions (including TPSL orders), liquidations, and potentially even deposits/withdrawals.

There are two scenarios within `process_pnl_and_rebalance_for_user_helper()` where these borrows occur:

1. When a user realises a PnL loss, the function attempts to borrow the deficit:

protocol/utils.rs::process_pnl_and_rebalance_for_user_helper

```
// ...
        // @audit if the pool doesn't have enough available liquidity, this will fail
        usdc_borrow_lend_market.borrow_from_pool(realize_pnl_deltas.usdc_borrowed)?;
// ...
```

2. When a user's unrealised loss borrow needs to increase to cover growing unrealised losses on open positions:

protocol/utils.rs::process_pnl_and_rebalance_for_user_helper

```
// ...
    if rebalance_unrealized_loss_borrow_deltas.usdc_borrow_delta > Decimal::ZERO {
        // @audit this can fail if there isn't enough supply-side liquidity in the borrow lend market
        usdc_borrow_lend_market.borrow_from_pool(
            rebalance_unrealized_loss_borrow_deltas
                .usdc_borrow_delta
                .try_into()?,
        )?;
    }
// ...
```

A failure in `borrow_from_pool()` in either of these cases will cause `process_pnl_and_rebalance_for_user_helper()` to return an error. Given that this helper function is called by critical handlers such as `handle_apply_funding_msg()`, `process_fill_for_user_helper()` (used in order placement and TPSL execution), `handle_liquidate_perp_positions_msg()`, and `handle_auto_deleverage_msg()`, its failure can have cascading effects, potentially halting funding rounds, preventing trades or liquidations, and undermining overall protocol stability.

The impact is classified as high because a failure here breaks core functionalities, essential for the exchange's operation and risk management. The likelihood is medium as it is contingent on specific market conditions (`USDC` market being illiquid or high utilization), which are plausible scenarios, especially under market stress, but not continuously present or easily triggered by a single malicious user without capital.

Recommendations

Implement more robust error handling or fallback mechanisms for scenarios where the `usdc_borrow_lend_market` cannot fulfill borrow requests made by `process_pnl_and_rebalance_for_user_helper()`.

Resolution

The finding has been closed with the following rationale provided by the development team:

> *"This is not addressable in my opinion and is a core problem with multi collateral implementations of perps exchanges. We will be cold starting the borrow lend pool and trying to incentivize market makers to collateralize in USDC as much as possible."*

BULL-14	Missing Access Control in Order and TPSL Cancellation Functions		
Asset	`protocol/handlers.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The `handle_cancel_order_msg()` and `handle_cancel_tpsl_msg()` functions in `handlers.rs` do not implement access control to verify if the caller is authorised to cancel the specified order or Take Profit/Stop Loss (TPSL) order.

This could allow any user to cancel orders or TPSL orders belonging to other users.

protocol/handlers.rs

```rust
/// Cancels an order
/// - `order_id` - the id of the order to cancel
#[instrument(level = "error", skip(self, state))]
pub(crate) fn handle_cancel_order_msg(
    &mut self,
    order_id: OrderId,
    state: &mut impl TxState,
) -> color_eyre::Result<()> {
    self.cancel_order(order_id, state)?;
    Ok(())
}

/// Cancels a TPSL
/// - `tpsl_order_id` - the id of the TPSL to cancel
#[instrument(level = "error", skip(self, state))]
pub(crate) fn handle_cancel_tpsl_msg(
    &mut self,
    tpsl_order_id: TpslOrderId,
    state: &mut impl TxState,
) -> color_eyre::Result<()> {
    self.cancel_and_unlink_tpsl(tpsl_order_id, state)?;
    Ok(())
}
```

Recommendations

Implement access control for these functions by ensuring the caller is either the owner of the order/TPSL or an authorised admin before proceeding with the cancellation.

Resolution

Note, this issue is known by the project team and has been reported to testing team before commencement of this review. The issue has been addressed in commit efb0515 by implementing appropriate access control.

BULL-15	Spot Liquidation Fails If Insurance Fund Cannot Cover Full Liquidator Reward		
Asset	`protocol/handlers.rs, usdc_insurance_fund.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

Spot liquidations fail entirely if the USDC insurance fund can not fully cover the liquidator's reward, allowing bad debt to accumulate even when partial repayment is possible.

When a spot liability is liquidated via `handle_liquidate_spot_liability_msg()`, if the liquidatee's assets are insufficient to cover the liquidator's full reward, the system attempts to draw the remaining amount from the USDC insurance fund. However, if the insurance fund cannot cover this *full* remaining reward, the entire liquidation process halts. This prevents the liquidatee's bad debt from being addressed and can lead to an accumulation of bad debt for the protocol.

The issue stems from the `try_take_from_fund()` method in `usdc_insurance_fund.rs`:

usdc_insurance_fund.rs::try_take_from_fund

```
pub(crate) fn try_take_from_fund(&mut self, amount: PositiveDecimal) -> Result<()> {
    if amount > self.amount {
        return Err(ExchangeError::InsuranceFundTooLow);
    }

    // ...
}
```

If the conditions for taking the full `amount` are not met, it returns an error. In `handle_liquidate_spot_liability_msg()` within `handlers.rs`, this error is propagated due to the use of the `?` operator when calling `try_take_from_fund()`:

protocol/handlers.rs::handle_liquidate_spot_liability_msg

```
// ...
if remaining_notional_liquidation_reward_to_take.gt(&PositiveDecimal::ZERO) {
    let mut usdc_insurance_fund = self.get_usdc_insurance_fund_required(state)?;
    let usdc_borrow_lend_market = borrow_lend_markets.get_mut(&USDC_ASSET_ID).ok_or(
        ExchangeError::BorrowLendMarketNotFound {
            asset_id: USDC_ASSET_ID,
        },
    )?;
    usdc_insurance_fund
        .try_take_from_fund(remaining_notional_liquidation_reward_to_take)?; // @audit This '?' causes the function to exit if
      ↪  try_take_from_fund returns an Err

    // ...
```

This halts the liquidation, even if a partial liquidation or payment to the liquidator was possible. The primary goal of liquidating the unhealthy position is therefore subverted.

Recommendations

Modify the liquidation process to account for situations where the insurance fund cannot cover the full liquidator reward. The priority should be to settle the liquidatee's unhealthy position as much as possible.

Resolution

The finding has been closed with the following rationale provided by the development team:

> *"We are aware of this problem, this requires us to do some form of socialized loss. We would prefer to handle this with more careful consideration which is not possible within the time constraint of the audit."*

BULL-16	User Set Max Leverage Unnecessarily Restricts Withdrawals		
Asset	`user/account.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

The `calculate_withdrawable_amount_of_asset()` function determines how much collateral a user can withdraw based on their initial margin, which incorporates the user-defined maximum leverage. This creates an unintended consequence - users who set a lower, more conservative leverage are permitted to withdraw less collateral than those who opt for higher risk.

As a result, this can negatively impact capital efficiency and user experience for those managing their risk proactively.

user/account.rs::calculate_withdrawable_amount_of_asset

```
pub fn calculate_withdrawable_amount_of_asset(
    &self,
    asset_id: AssetId,
    margin_calculation_state: &MarginCalculationState,
) -> Result {
    let mut combined_spot_ledgers = self.spot_ledgers.clone();
    combined_spot_ledgers.insert(USDC_ASSET_ID, self.usdc_ledger.ledger.clone());

    let spot_ledger = combined_spot_ledgers
        .get(&asset_id)
        .filter(|ledger| !ledger.asset.is_zero())
        .ok_or(ExchangeError::UserNoDepositsInAsset { asset_id })?;

    if !self.perp_ledgers.is_empty() || self.has_liabilities() {
        // @audit: Using Initial margin here.
        let margin_type = MarginType::Initial(None);

        //@audit this takes user_selected_max_leverage into account
        let used_margin = self.calculate_used_margin(margin_type, margin_calculation_state)?;

        // ...
}
```

Recommendations

Consider overriding the user's max leverage when calculating the withdrawable amount of assets, decoupling withdrawable collateral from the user's leverage preference, or enforcing a consistent baseline leverage for withdrawal calculations.

Resolution

This issue was addressed in PR #397 by ignoring the user's max leverage during withdrawals.

BULL-17	ADL May Overfill Positions By Using Outdated Position Size Snapshot		
Asset	`protocol/handlers.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The Auto Deleveraging (ADL) mechanism in `handle_auto_deleverage_msg()` can mismanage a user's position by incorrectly calculating the trade `fill_size` when matching the deleveraged user (`bad_user`) with counterparties.

Specifically, the function determines `fill_size` based on the `bad_user`'s initial position size as recorded in `initial_bad_user_perp_ledgers`, rather than the live, remaining size of the position still requiring closure. Since `bad_user_ledger.position.size.abs()` is derived from this pre-fill snapshot, it does not account for partial fills already processed during that market's iteration. This can result in overfills or incorrect matching behaviour.

protocol/handlers.rs::handle_auto_deleverage_msg

```
pub(crate) fn handle_auto_deleverage_msg(
  // ...
) -> color_eyre::Result<()> {
    // ...

    for bad_account_address in accounts_to_adl {
        // ...

        // @audit `initial_bad_user_perp_ledgers` is a clone made before processing fills for that specific market.
        let initial_bad_user_perp_ledgers = bad_user_account.perp_ledgers.clone();
        // if account_equity is +, we can close all positions @ mark price
        // if account_equity is -, we need to close -upnl positions @ distributed bankruptcy prices and +upnl positions at mark
    ↪  price
        for (market_id, bad_user_ledger) in initial_bad_user_perp_ledgers.into_iter() {
            // ...
            for counterparty_address in counterparties_to_adl.iter() {
                // ...

                // @audit `bad_user_ledger` is from a clone of `initial_bad_user_perp_ledgers`
                //        which is made before processing fills for that specific market.
                //        This means that `bad_user_ledger.position.size.abs()` is the original
                //        size of the position, not the remaining size to close.
                let fill_size = PositiveDecimal::try_from(
                    counterparty_ledger
                        .position
                        .size
                        .abs()
                        .min(bad_user_ledger.position.size.abs()),
                )?;
                // ...

                // @audit using the sign of `remaining_size_to_close` to determine
                //        the direction of the fill is not correct after `remaining_size_to_close'
                //        has overshot zero.
                remaining_size_to_close = remaining_size_to_close
                    .try_sub(fill_size.try_mul(remaining_size_to_close.signum())?)?;

                if remaining_size_to_close.is_zero() {
                    break;
                }
            // ...
```

If multiple counterparties are used to close a single position of the `bad_user`, the `fill_size` calculated with later counterparties does not account for the portion of the `bad_user`'s position already filled by earlier counterparties. This can lead to `fill_size` being larger than what's actually left to close. Consequently, the `remaining_size_to_close` variable, which tracks the progress against the original position, can overshoot zero and flip its sign (e.g., from `+10` to `-10`). However, the `fill_side` (the direction of the `bad_user`'s fill) remains fixed based on the *original* side of the `bad_user_ledger`. If the `bad_user`'s position is overfilled and flips (e.g., a long position becomes short), subsequent fills with the original `fill_side` will further extend this new, incorrect position instead of moving it towards zero. This can result in the `bad_user` not being properly deleveraged and potentially ending up with an unintended position.

Furthermore, once an overshot `remaining_size_to_close` has occurred, the subsequent `remaining_size_to_close` calculations will not be correct, as the fill side remains unchanged but the sign of the `fill_size` has flipped as described in the code snippet above.

To better illustrate the issue, consider a `bad_user` with an initial long position of 100 units in a market (`remaining_size_to_close` = `+100`, `fill_side` = `Short`).

1. Counterparty A has a short position of 60 units.
   - `bad_user_ledger.position.size.abs()` is 100.
   - `fill_size` = `min(60, 100)` = 60.
   - `bad_user` is filled short 60. `bad_user_account` position becomes long 40.
   - `remaining_size_to_close` becomes `100 - 60 = 40`.
2. Counterparty B has a short position of 70 units.
   - `bad_user_ledger.position.size.abs()` is *still* 100 (from the clone).
   - `fill_size` = `min(70, 100)` = 70.
   - The `bad_user` (currently long 40) is filled short 70. `bad_user_account` position becomes short 30 (overshot).
   - `remaining_size_to_close` becomes `40 - 70 = -30`.
3. Counterparty C has a short position of 20 units.
   - `bad_user_ledger.position.size.abs()` is *still* 100.
   - `fill_size` = `min(20, 100)` = 20.
   - The `bad_user` (currently short 30) is filled short another 20 (because `fill_side` is still `Short`). `bad_user_account` position becomes short 50.
     The `bad_user` is not deleveraged to flat but ends up with an unintended short position.
   - `remaining_size_to_close` becomes `-30 - (-20) = -10`, even though the position is actually short 50.

This incorrect handling of position sizing during ADL fails to properly deleverage the user and can leave their account in an undesirable state, potentially with a new, unmanaged risk. ADL is a critical safety function, and its incorrect behaviour has a high impact. The likelihood is low because the `handle_auto_deleverage_msg()` function is expected to be called only in rare and severe market scenarios.

Recommendations

Change the `fill_size` calculation within the counterparty loop to use the `bad_user`'s *current remaining size to be closed* for that market, instead of their *initial total position size*.

Furthermore, calculate `remaining_size_to_close` using the `fill_side` to determine the direction of the fill, instead of using the sign of `remaining_size_to_close`.

Resolution

This issue was addressed in PR #405 by refactoring `handle_auto_deleverage_msg()`.

BULL-18	Incorrect ADL Fill Price For User With Negative Equity		
Asset	`protocol/handlers.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

During the Auto Deleveraging (ADL) process, the fill price for a trade is typically set to the liquidated user's bankruptcy price to ensure all bad debt is absorbed and no residual equity remains. However, if `remaining_equity_to_distribute` is negative, the system incorrectly sets `pnl_to_realize` to zero instead of allowing it to reflect the loss.

This results in a fill price that is more favourable than the true bankruptcy price, allowing the liquidated user to retain positive account value post ADL, a value they can potentially withdraw. In turn, counterparties are unfairly penalised, as they must relinquish a larger portion of their profit than intended.

protocol/handlers.rs::handle_auto_deleverage_msg

```
// ...
let fill_price =
    if unweighted_account_equity.is_sign_negative() && upnl.lt(&Decimal::ZERO) {
        let pnl_to_realize = if remaining_equity_to_distribute.is_sign_negative() {
            PositiveDecimal::ZERO
        } else {
            bad_user_ledger
                .position
                .notional_value(mark_price)?
                .try_div(total_notional_value_neg_upnl_positions)?
                .try_mul(remaining_equity_to_distribute)?
        };

        bad_user_ledger
            .position
            .calculate_fill_price_for_given_realized_pnl(pnl_to_realize.as_dec())?
    } else {
        mark_price
    };
// ...
```

Recommendations

Modify the implementation to allow the fill price to be equal to the bankruptcy price, even if `remaining_equity_to_distribute` is negative.

Resolution

This issue was addressed in PR #405 by refactoring `handle_auto_deleverage_msg()`.

BULL-19	Auto Deleverage Can Be Frontrun		
Asset	`protocol/handlers.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

When `handle_auto_deleverage_msg()` is called, counterparties have a clear incentive to frontrun the transaction and close their positions manually. By doing so, they can avoid having their profitable positions socialised due to the auto-deleveraging mechanism.

This undermines the effectiveness of the ADL process. Since positions may be closed before the ADL executes, the sizes available for deleveraging are reduced, leading to incomplete execution of the intended deleveraging.

protocol/handlers.rs::handle_auto_deleverage_msg

```
for counterparty_address in counterparties_to_adl.iter() {
  // ...

  let fill_size = PositiveDecimal::try_from(
      counterparty_ledger
          .position
          .size
          .abs()
          .min(bad_user_ledger.position.size.abs()),
  )?;

  if fill_size.is_zero() {
      continue;
  }

  // ...
}
```

Recommendations

Ensure the above comments are understood and consider implementing changes if desired.

Potential mitigations could include making the ADL process non-interactive and internal-only, such that users cannot observe or front-run it via the mempool. This could involve executing ADL actions as part of a batch operation, internal queue, or validator-triggered process. Alternatively, obscure the list of selected counterparties or delay public visibility of their selection until after the ADL has been processed.

Resolution

The finding has been closed with the following rationale provided by the development team:

> *"This is not a problem, in fact it is encouraged, since if a user can close their huge upnl positions this will essentially de-risk the platform, which is the goal of auto deleveraging."*

BULL-20	Misleading Success In `handle_withdraw_msg()`		
Asset	`protocol/handlers.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

The `handle_withdraw_msg()` function in `code/sov-rollup/crates/exchange/src/protocol/handlers.rs` may return a general success status `Ok(())` in scenarios that may be misleading to the caller, specifically:

1. It returns `Ok(())` even if no funds are actually withdrawn, because the user's balance for the specified asset is zero, despite a non-zero `requested_amount`.
2. If the user has some balance, but less than the `requested_amount` (or if the `withdrawable_token_amount` limits it), the function silently withdraws a reduced amount and returns `Ok(())`. The caller is not explicitly informed via the return value of the exact amount that was withdrawn.

This behaviour can mislead integrating systems or callers who might interpret a generic success response as a confirmation that the full `requested_amount` was transferred.

protocol/handlers.rs::handle_withdraw_msg

```
pub(crate) fn handle_withdraw_msg(
    &mut self,
    asset_id: AssetId,
    requested_amount: PositiveDecimal,
    context: &Context,
    state: &mut impl TxState,
) -> color_eyre::Result<()> {
    if requested_amount.is_zero() { // @audit This case for a zero requested_amount is handled correctly by doing nothing.
        return Ok(());
    }

    // ... (omitted setup and validation code) ...

    if spot_ledger.asset.is_zero() { // @audit This early return means Ok(()) is returned even if requested_amount > 0 and no
      ↪  withdrawal happens.
        return Ok(());
    }

    // @audit actual_withdraw_amount can be less than requested_amount, or even zero if requested_amount was > 0
    // @audit but withdrawable_token_amount or spot_ledger.asset forced it down.
    let actual_withdraw_amount = requested_amount
        .min(withdrawable_token_amount)
        .min(spot_ledger.asset);

    // @audit If actual_withdraw_amount is zero (for a non-zero requested_amount), or if it's less than requested_amount,
    // @audit the function proceeds and returns Ok(()), without explicitly communicating the actual amount withdrawn.
    // @audit An event is emitted, but the direct return value does not convey this crucial detail.
    {
        // ...

        process_balance_update_for_user_helper(
            &mut user_account,
            borrow_lend_market,
            actual_withdraw_amount.neg(),
        )?;
    }

    self.transfer(
        TransferDirection::VaultToUser,
        asset_id,
        user_address,
        actual_withdraw_amount,
        state,
    )?;

    // ... (omitted event emission and state updates) ...

    Ok(()) // @audit Returns a generic success, regardless of whether actual_withdraw_amount matched requested_amount or was zero.
}
```

The impact is classified as low because this behaviour does not directly lead to a loss of funds within the exchange or break its core functionality. However, it presents a misleading and incomplete API response that could cause incorrect state tracking or logic errors in external systems. The likelihood is high, as any user can trigger these scenarios.

Recommendations

Modify the `handle_withdraw_msg()` function to provide clear feedback to the caller about the outcome of the withdrawal operation.

Consider returning `Ok(actual_withdraw_amount)` instead of an empty `Ok(())`.

Resolution

This issue has been addressed in PR #383. An error is now returned if no withdrawal occurs. If an amount less than `requested_amount` is withdrawn, this amount is emitted in the event.

BULL-21	Stale State for Counterparties in Auto Deleverage (ADL)		
Asset	`protocol/handlers.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

During an Auto Deleverage (ADL) event, the system does not update the pricing and funding states for users in the `counterparties_to_adl` list before their accounts are modified.

The `handle_auto_deleverage_msg()` function calls `validate_pricing_for_users()` and `apply_funding_for_users()` only for the `accounts_to_adl`. However, the `counterparties_to_adl` also participate in the ADL process, and their positions and balances are modified.

Without updating their states, their spot balances might be incorrect due to missed interest accrual updates before their account is fetched. As a result, future funding payments might be inaccurate as their position sizes change due to the ADL fill.

protocol/handlers.rs::handle_auto_deleverage_msg

```
pub(crate) fn handle_auto_deleverage_msg(
    &mut self,
    accounts_to_adl: Vec,
    counterparties_to_adl: Vec,
    context: &Context,
    state: &mut impl TxState,
) -> color_eyre::Result<()> {
    self.validate_admin(AdminType::Protocol, context.sender(), state)?;
    let current_timestamp: UnixTimestampMillis = self.chain_state.get_time(state)?.into();

    self.accrue_interest_on_borrow_lend_markets(current_timestamp, state)?;
    self.validate_pricing_for_users(
        accounts_to_adl.iter().collect::<>(), // @audit: counterparty users are not included here
        vec![],
        vec![],
        current_timestamp,
        state,
    )?;
    self.apply_funding_for_users(
        accounts_to_adl.iter().collect::<>(), // @audit: counterparty users are not included here
        current_timestamp,
        state,
    )?;
// ... existing code ...
            for counterparty_address in counterparties_to_adl.iter() {
                // @audit: counterparty_account is fetched here, but its state (pricing, funding)
                // @audit: might be stale as it wasn't updated above.
                let mut counterparty_account =
                    self.get_user_account(counterparty_address, state)?;

                let counterparty_ledger = counterparty_account.perp_ledgers.get(&market_id);
// ... existing code ...
                let (offbook_events, bad_account_realized_pnl, counterparty_realized_pnl) =
                    process_trade_off_book_for_two_users_helper(
                        &mut bad_user_account,
                        usdc_borrow_lend_market,
                        &mut usdc_pnl_pool,
                        user_fill,
                        &margin_calculation_state,
                        &mut counterparty_account, // @audit: Stale counterparty_account used here
                    )?;
// ... existing code ...
```

This issue is classified as high impact because it can lead to incorrect accounting and potentially significant financial miscalculations for the users involved in an ADL. The likelihood is low because the `handle_auto_deleverage_msg()` function is expected to be called only in rare and severe market scenarios.

Recommendations

Call the `validate_pricing_for_users()` and `apply_funding_for_users()` functions for all users involved in the ADL process, including `counterparties_to_adl`.

Resolution

This issue was addressed in PR #405 by validating pricing and applying funding for the counterparties.

BULL-22	Logic Issues In `find_risk_increasing_order_ids()`		
Asset	`user/perp_ledger.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `find_risk_increasing_order_ids()` function incorrectly assesses risk by failing to account for the cumulative size of multiple closing orders and by not explicitly excluding `reduce_only` orders. This can lead to incomplete or improper order cancellations during risk management events such as `force_cancel_all_orders_for_user`, potentially leaving users with unintended net positions or larger than expected risk exposure.

The current function identifies an order as risk-increasing if it meets one of two conditions:

1. The order is on the same side as the existing position, intending to increase its size (e.g., a bid order for an existing long position).
2. The order is on the closing side of the position and its individual size is greater than the absolute size of the current position (e.g., an ask order for a long position that is large enough to flip the position).

This logic has the following issues:

- **Cumulative Effect Ignored:** Each order is evaluated individually. If a user has, for example, a $100 long position and places two separate $100 ask orders, neither order would be flagged individually as risk-increasing by the current logic because each $100 ask is not greater than the $100 long position. However, if both orders were to fill, the user's position would flip to a $100 short position. This scenario, where multiple orders collectively increase risk or flip the position, is not detected.
- **`reduce_only` Orders Not Excluded:** The function does not check if an order is `reduce_only`. Such orders, by definition, should only decrease position size and never increase risk. They should be explicitly excluded from being flagged as risk-increasing.
- **Position Flipping to Smaller Size:** The condition `order.remaining_size.as_dec().gt(&self.position.size.abs())` for individual closing orders means that an order flipping the position to a smaller net size on the opposite side (e.g., a $101 ask order for a $100 long position, resulting in a $1 short) is currently flagged as risk-increasing. Cancelling such an order leaves the user with their original, larger ($100 long) position, which might be contrary to minimising absolute risk exposure.

The following code snippet highlights the relevant logic:

user/perp_ledger.rs::find_risk_increasing_order_ids

```
pub fn find_risk_increasing_order_ids(&self) -> Result> {
    if self.position.size.is_zero() {
        return Ok(self.get_all_order_ids());
    }

    let mut risk_increasing_order_ids = Vec::new();

    let closing_side = self
        .position
        .side_to_close()
        .ok_or(ExchangeError::ZeroSize)?;

    for (order_id, order) in self.orders.iter() {
        // @audit The core logic for determining "risk-increasing" resides here.
        // @audit It does not check `order.reduce_only`.
        // @audit It evaluates orders individually, missing cumulative effects.
        let should_cancel = order.side == closing_side.reverse()
            || order.side == closing_side
                && (order.remaining_size.as_dec().gt(&self.position.size.abs()));

        if should_cancel {
            risk_increasing_order_ids.push(*order_id);
        }
    }

    Ok(risk_increasing_order_ids)
}
```

The impact is medium because faulty risk assessment during forced cancellations can lead to suboptimal position management, potentially resulting in unexpected exposure or losses for the user, or breaking the intended functionality of system risk reduction. The likelihood is medium as the conditions for triggering this behaviour (having an open position and multiple relevant orders) are plausible scenarios for active traders.

Recommendations

Refactor the `find_risk_increasing_order_ids()` function such that risk-increasing orders are identified more accurately.

Resolution

This issue was resolved in PR #398 by refactoring `find_risk_increasing_order_ids()`.

BULL-23	Incorrect Asset Sorting with Preferred Asset Priority		
Asset	`user/account.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `get_assets_sorted_by_weight_with_priority()` function in `user/account.rs` does not correctly maintain the descending sort order of assets by their `asset_weight` when a `preferred_asset_id` is provided.

If the preferred asset is not already the highest weighted asset, swapping it to the front disrupts the sort order of the remaining assets. This can result in liquidators or other processes receiving assets in an order that does not reflect their true quality or liquidity ranking after the preferred asset.

The function is intended to return a list of `AssetId`s sorted primarily by a `preferred_asset_id` (if specified and present) and secondarily by descending `asset_weight`. The current logic is as follows:

1. It collects all spot assets held by the user along with their respective `asset_weight`.
2. It sorts this collection in descending order based on these weights.
3. If a `preferred_asset_id` is specified and found within the user's assets, the function swaps this preferred asset with the element at the first position in the sorted list.

The issue arises in step 3. For example, consider the following assets and weights after the initial sort (step 2): `AssetA(weight=0.9)`, `AssetB(weight=0.7)`, `AssetC(weight=0.5)`, `PreferredD(weight=0.3)`, `AssetE(weight=0.1)`. If `PreferredD` is the `preferred_asset_id`, the current code will swap `PreferredD` with `AssetA`, resulting in `AssetA` being lower on the list than it should be.

user/account.rs::get_assets_sorted_by_weight_with_priority

```rust
pub fn get_assets_sorted_by_weight_with_priority(
    &self,
    preferred_asset_id: AssetId,
) -> Result> {
    let mut assets_and_weights: Vec<_> = self
        .spot_ledgers
        .iter()
        .filter(|(_, ledger)| ledger.has_asset())
        .map(|(asset_id, ledger)| (*asset_id, ledger.weights.asset_weight))
        .collect();

    // Sort by descending weight
    assets_and_weights.sort_by_key(|(_, weight)| std::cmp::Reverse(*weight));

    // Move preferred asset to front if it exists
    if let Some(idx) = assets_and_weights
        .iter()
        .position(|(id, _)| *id == preferred_asset_id)
    {
        assets_and_weights.swap(0, idx); // @audit: This swap disrupts the sort order for other elements
    }

    Ok(assets_and_weights.into_iter().map(|(id, _)| id).collect())
}
```

Recommendations

To address this issue, the preferred asset should be identified and temporarily removed from the list. The remaining assets should then be sorted by weight in descending order. Finally, if the preferred asset was found, it should be inserted at the beginning of the sorted list.

This ensures that the preferred asset is prioritized while the rest of the assets maintain their correct descending sort order by weight.

Resolution

This issue was addressed in PR #404 by refactoring `get_assets_sorted_by_weight_with_priority()`.

BULL-24	Incorrect Impact Price Calculation Due To Strict Inequality		
Asset	`common/helpers.rs, protocol/handlers.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

A strict inequality in impact price calculation can cause valid price levels to be skipped, resulting in zero or inaccurate premium indexes.

The `calculate_impact_price()` function in `crates/exchange/src/common/helpers.rs` uses a strict inequality (`>`) when comparing the accumulated `current_cash_delta` (total notional value of orders processed so far) with the target `impact_margin_notional`.

common/helpers.rs::calculate_impact_price

```
pub(crate) fn calculate_impact_price<'a>(
    impact_margin_notional: PositiveDecimal,
    orders_iter: impl Iterator,
) -> Result {
    let mut current_cash_delta = PositiveDecimal::ZERO;
    for (level, level_orders) in orders_iter {
        let level_cash_delta = calculate_notional(*level, level_orders.total_size)?;
        current_cash_delta = current_cash_delta.try_add(level_cash_delta)?;
        if current_cash_delta > impact_margin_notional {
            return Ok(*level);
        }
    }
    Ok(PositiveDecimal::ZERO)
}
```

If `current_cash_delta` becomes exactly equal to `impact_margin_notional` at a specific price level, the current logic will either proceed to the next (and worse) price level or return `PositiveDecimal::ZERO` if it was the last level in the iteration. This results in an inaccurately calculated impact price.

The `calculate_impact_price()` function is called in `handle_update_premium_indexes_msg()` to determine the `impact_bid_price` and `impact_ask_price` for a market which are used for updating a market's `premium_index`. An inaccurate `premium_index` can directly lead to miscalculated funding rates.

Recommendations

Change the strict inequality `>` in `calculate_impact_price()` to an inclusive inequality `>=`. This will ensure that if the `current_cash_delta` exactly matches the `impact_margin_notional` at a given price level, that price level is correctly returned as the impact price.

Resolution

This issue was resolved in PR #403 by using an inclusive inequality in `calculate_impact_price()`.

BULL-25	Premature Termination Of Batch Mark Price Updates On Single Market Error		
Asset	`protocol/primitives.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

A single faulty market update can halt the entire batch, leaving all subsequent markets with stale mark prices.

The `update_mark_prices()` function in `primitives.rs` is designed to process a batch of mark price updates for multiple perpetual markets. However, if an error occurs during the data retrieval phase for any single market in the batch (e.g., the market metadata is not found, the perpetual market itself cannot be fetched, or its associated oracle price is stale or missing), the function immediately propagates this error and terminates.

This premature termination prevents any subsequent mark price updates in the same batch from being processed, even if they are valid. This behaviour can lead to multiple markets having stale mark prices if just one problematic update is included in a batch.

The current implementation uses the `?` operator for several critical data retrieval steps within the loop, which causes this early exit:

protocol/primitives.rs::update_mark_prices

```
// ...
for p in updates {
    let market_metadata = markets_metadata.get(&p.market_id).ok_or( // @audit If this errors, the whole function stops
        ExchangeError::MarketMetadataNotFound {
            market_id: p.market_id,
        },
    )?;

    let median_orderbook_price = {
        let perp_market = self.get_perp_market_required(p.market_id, state)?; // @audit If this errors, the whole function stops
        // ...
    };

    let market_id = p.market_id;
    let oracle_price = {
        oracle_prices.validate_oracle_prices_up_to_date( // @audit If this errors, the whole function stops
            market_metadata.base_asset_id(),
            current_timestamp,
        )?;
        oracle_prices.get_oracle_price(market_metadata.base_asset_id())? // @audit If this errors, the whole function stops
    };

    // ...

    match perp_prices.update_mark_price( // @audit This part correctly handles its own error per market
        p,
        median_orderbook_price,
        oracle_price,
        publish_timestamp,
    ) {
        Ok(_) => {
            success_markets.push(market_id);
            // ...
        }
        Err(e) => {
            failed_markets.push(market_id);
            // ...
```

Recommendations

Refactor the loop within `update_mark_prices()` to handle errors for each market update individually. Instead of using the `?` operator in a way that terminates the entire function, errors encountered while updating the mark price should be caught and the relevant `market_id` added to `failed_markets`.

Resolution

This issue was resolved in PR #381 by refactoring `handle_update_mark_prices_msg()`.

BULL-26	PNL Pool Carries Unbacked Gains from Insolvent Liquidatees in Perp Liquidations		
Asset	`protocol/handlers.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

Insolvent liquidations can create unbacked debt in the `usdc_pnl_pool`, causing realised PnL to be recorded without actual asset backing.

When a liquidatee in a perpetuals market becomes insolvent (i.e., their unweighted account equity becomes negative) after their positions are closed and the standard liquidator reward is accounted for, the `usdc_pnl_pool` effectively absorbs this bad debt. This occurs because the PnL associated with the liquidatee's losses and fees increases the `usdc_pnl_pool.market_realized_pnl` on an accounting basis, but if the liquidatee cannot cover these amounts with actual assets, this portion of the PnL pool's balance becomes an unbacked IOU.

The `handle_liquidate_perp_positions_msg` function in `crates/exchange/src/protocol/handlers.rs` does not directly engage the `usdc_insurance_fund` to cover a specific liquidatee's negative unweighted equity resulting from a perp liquidation. While the insurance fund is intended as a backstop for the PnL pool systemically, this deferred intervention means the PnL pool carries the immediate risk of unbacked gains. If such bad debts accumulate, they can impair the PnL pool's ability to meet its actual USDC obligations (e.g., paying profits to other users), potentially stressing the insurance fund and, in extreme cases, leading to ADL or system instability. The intended design should have the insurance fund cover such individual insolvencies arising from liquidations directly, ensuring the PnL pool primarily facilitates PnL transfer rather than absorbing bad debt.

This issue is classified as high impact due to its potential to affect core protocol functionality related to settlement and solvency. The likelihood is low, as it is only an issue if the liquidatee's unweighted account equity reaches negative, which is unlikely to occur as the liquidatee is likely to be liquidated before this happens.

Recommendations

Modify the `handle_liquidate_perp_positions_msg` function to perform the following steps after all PnL from position closing and the intended liquidator reward/liquidatee penalty (`total_liquidator_reward`) have been processed for the liquidatee:

1. Calculate the liquidatee's final `unweighted_account_equity`.
2. If `unweighted_account_equity` is negative, this represents bad debt.
3. The `usdc_insurance_fund` should be called upon to cover this specific bad debt amount. This would involve attempting to take the absolute value of the negative equity from the insurance fund.
4. The liquidatee's USDC balance (or liability) should then be adjusted by this amount, effectively bringing their unweighted equity to zero (or as close as possible if the insurance fund is insufficient).
   This ensures that the "gains" attributed to the PnL pool from the insolvent liquidatee's obligations are backed by actual USDC from the insurance fund, maintaining the PnL pool's integrity.

Resolution

The finding has been closed with the following rationale provided by the development team:

> *"This is a non-issue. Using the insurance fund to cover a users "bad debt" on perp liquidation is a bad design choice because you're effectively paying the liquidatee to incur bad debt. The insurance fund will be used to pay the liquidator on their spot / borrow lend position instead of to the liquidatee directly. We prefer this because then you are not paying the liquidatee to take on bad debt, you are paying the liquidator to take on bad debt with the ability to pay it back."*

BULL-27	Incorrect Hourly Check In `update_funding_rate()` Allows Premature Updates		
Asset	`pricing/perp_prices.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The funding rate update mechanism in `update_funding_rate()` can be triggered more frequently than the intended *"no more than once per hour"* limit.

This is due to a flawed timestamp check that compares the hour component of the current time with the hour component of the last update time, rather than ensuring a full hour has elapsed. Consequently, calls made near an hour boundary (e.g., at 10:59 and then again at 11:00) would both be permitted, even though they are only one minute apart.

This premature update means that the average premium index could be based on an accumulation period significantly shorter than one hour (e.g., just one minute in the scenario above), leading to incorrect funding rates.

pricing/perp_prices.rs::update_funding_rate

```
pub(crate) fn update_funding_rate(
    &mut self,
    market_id: MarketId,
    current_timestamp: UnixTimestampMillis,
    market_funding_parameters: FundingParameters,
) -> Result {
    let last_funding_meta = self.get_funding_meta(market_id)?;

    // @audit Incorrect check: allows calls if current_hour_block > last_hour_block,
    // @audit e.g. 10:59 (block 10) and 11:00 (block 11) are 1 min apart but 11 > 10.
    if current_timestamp.as_secs_i64() / SECONDS_PER_HOUR
        <= last_funding_meta.last_update_timestamp.as_secs_i64() / SECONDS_PER_HOUR
    {
        return Err(ExchangeError::FundingRateAlreadyUpdated {
            market_id,
            last_update_timestamp: last_funding_meta.last_update_timestamp,
        });
    }

    // Get accumulated premium index data
    let mut accum_premium_index = self.get_accum_premium_index(market_id)?;
```

Recommendations

Modify the check in `update_funding_rate()` to ensure that at least `SECONDS_PER_HOUR` has elapsed since `last_funding_meta.last_update_timestamp`.

Resolution

The finding has been closed with the following rationale provided by the development team:

"This is fine, not an issue as we don't want delays to cascade into future funding rates being delayed, e.g a funding rate at 11:02 should not mean the next funding rate update should be after 12:02."

BULL-28	`queue_tpsls_for_execution_for_market()` May Be Called With Stale Prices		
Asset	`protocol/handlers.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

When updating oracle prices in `handle_update_oracle_prices_msg()`, the validity of `publish_timestamp` is not checked. This means that if, for example, the call to `handle_update_oracle_prices_msg()` is delayed due to congestion such that `publish_timestamp` is already outdated, then the prices are still successfully updated using stale data.

In general, this is not an issue since every operation that requires pricing info normally calls `validate_pricing_for_users()` beforehand, and thus will fail if prices are stale. However, here `queue_tpsls_for_execution_for_market()` is immediately called after the update without any staleness check having occurred, meaning that it may use stale prices:

protocol/handlers.rs::handle_update_oracle_prices_msg

```
pub(crate) fn handle_update_oracle_prices_msg(
    &mut self,
    prices_to_update: Vec,
    publish_timestamp: i64,
    context: &Context,
    state: &mut impl TxState,
) -> color_eyre::Result<()> {

// ...

self.queue_tpsls_for_execution_for_market(
    market_id,
    TpslPriceCondition::Oracle,
    None,
    current_timestamp,
    state,
)?;
```

Note, the same issue occurs in `handle_update_mark_prices_msg()`.

Recommendations

Validate `publish_timestamp` when updating prices.

Resolution

This issue was addressed in PR #380 by ensuring `publish_timestamp` is valid in `handle_update_oracle_prices_msg()`.

BULL-29	Missing Functionality For Rotating Admins		
Asset	`*.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The protocol includes no functionality for changing the addresses of the admins.

Being unable to rotate admin keys can cause issues in case of key compromise.

The logic for changing admins is implemented in `update_admin()`, but it is not publicly accessible.

Recommendations

Consider making `update_admin()` public facing, ensuring sufficient access controls.

Resolution

This issue was addressed in PR #402 by implementing a `handle_update_admin_msg()` function.

BULL-30	Missing Price Validation		
Asset	`protocol/handlers.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `handle_place_tpsl_for_market_msg()` function uses asset prices without validating their freshness, potentially triggering TPSLs based on stale data.

Functions relying on asset prices should invoke `validate_pricing_for_users()` to ensure up to date pricing. However, `handle_place_tpsl_for_market_msg()` omits this check, despite using prices to evaluate TPSL triggers. As a result, there is a risk that TPSLs may be executed based on outdated pricing information.

Recommendations

Call `validate_pricing_for_users()` at the start of `handle_place_tpsl_for_market_msg()` to ensure up to date pricing.

Resolution

The issue was addressed in PR #436.

BULL-31	No Support For Tokens With Large Supply
Asset	`protocol/primitives.rs`
Status	**Closed:** See Resolution
Rating	Informational

Description

Token amounts on the Sovereign SDK are represented by a `u128` ($\approx 10^{38}$) while they are represented by a `Decimal` on the Bullet Exchange, which only has 96 bits of accuracy ($\approx 10^{28}$). This means tokens with a large amount of supply cannot be represented exactly on the exchange and will cause overflows.

For example, given a token with 18 decimals and a large total supply, if a user tries to deposit 10^{11} amount of these tokens, `try_convert_integer_with_precision` will overflow.

Note that overflows could occur in other places than just deposits. For example, a user could place a limit buy order that increases their balance of the token above 10^{11}, any other user that wants to fill this order will have their transaction reverted due to an overflow. As such, it it crucial that these tokens are not added to the exchange.

Recommendations

Consider moving to a larger token amount representation, ideally `u128`.

Alternatively, allow `asset_metadata.decimals()` to be set to a negative value, this allows these assets to be represented with less accuracy on the exchange.

Resolution

The finding has been closed with the following rationale provided by the development team:

> *"Token amounts are stored only for things with borrow lend markets primarily, or spot markets (in future). For these we can choose which pools we allow and which assets we allow trading of. For perp products this is not an issue as we can offer products like 1MBONK for example where 1 token is denominated as 1 million base bonk tokens."*

BULL-32	Duplicate Liquidation Checks During Perp Liquidations
Asset	`protocol/handlers.rs`
Status	**Resolved:** See Resolution
Rating	Informational

Description

During the liquidation of a perpetual position, the system checks whether the account is eligible for liquidation both before and after cancelling the user's open orders, resulting in a redundant validation step.

protocol/handlers.rs::handle_liquidate_perp_positions_msg

```
let liquidation_check =
    liquidatee_account.check_backstop_liquidatable(&margin_calculation_state)?;
if !liquidation_check.is_liquidatable { //@audit first liquidation check
    return Err(ExchangeError::AccountNotLiquidatable {
        address: liquidatee_address.to_string(),
        weighted_account_equity: liquidation_check.weighted_account_equity,
        perp_maintenance_margin_requirement: liquidation_check
            .perp_maintenance_margin_requirement
            .as_dec(),
    }
    .into());
}

self.force_cancel_all_orders_for_user(&liquidatee_address, false, state)?;

//...

if liquidatee_weighted_account_equity //@audit second liquidation check
    .ge(&backstop_liquidation_threshold_maintenace_margin_requirement.as_dec())
{
    return Err(ExchangeError::AccountNotLiquidatable {
        address: liquidatee_address.to_string(),
        weighted_account_equity: liquidatee_weighted_account_equity,
        perp_maintenance_margin_requirement: liquidatee_perp_maintenance_margin_requirement
            .as_dec(),
    }
    .into());
}
```

Recommendations

Consider removing one of the liquidation checks to improve efficiency and code clarity.

Resolution

This issue was addressed in PR #385 by removing the second liquidation check.

BULL-33	Outdated Dependencies
Asset	`*.rs`
Status	**Closed:** See Resolution
Rating	Informational

Description

The following dependencies are outdated and have known security issues:

- **crossbeam-channel 0.5.14** *(double free on `Drop`)*
- **curve25519-dalek 3.2.0** *(timing variability)*
- **ed25519-dalek 1.0.1** *(double public key signing function oracle attack)*
- **idna 0.1.5** *(`idna` accepts Punycode labels that do not produce any non-ASCII when decoded)*
- **openssl 0.10.71** *(Use-After-Free in `Md::fetch` and `Cipher::fetch`)*
- **rsa 0.9.8** *(Marvin Attack: potential key recovery through timing side channels)*

Note, the full details on vulnerable dependencies can be found by running `cargo audit`.

Recommendations

Update the dependencies with known security issues to their latest, stable versions.

Additionally, consider setting up monitoring infrastructure to ensure any security advisories in dependencies are noticed and addressed quickly.

Resolution

The finding has been closed, with the development team committing to address it at a later date.

Appendix A Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

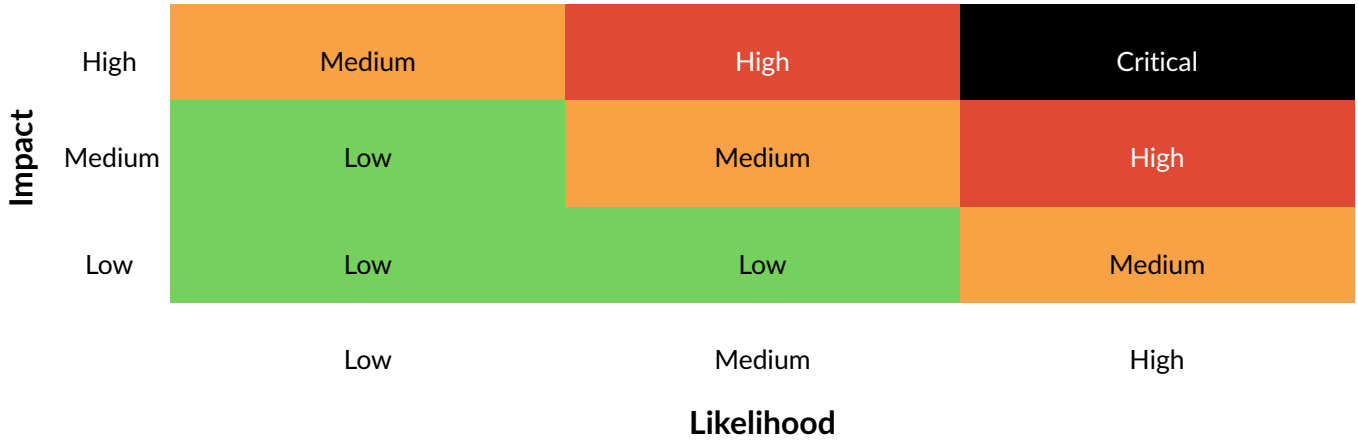

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.