Zellic

May 30, 2025

Prepared for
enndsz
matteo_gte
ndt1993_01
Liquid Labs, Inc.

Prepared by
Katerina Belotskaia
Weipeng Lai
Zellic

GTE – Perp
Smart Contract Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Liquid Labs, Inc. from May 5th to May 16th, 2025. During this engagement, Zellic reviewed GTE – Perp's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Are there any vulnerabilities that could result in a direct or indirect loss of user or protocol funds?
- Are there any accounting vulnerabilities that could lead to discrepancies in balances, profit/loss calculations, or fee distributions?
- Is the funding-rate calculation correct?
- Are margin calculations accurate?
- Is the liquidation mechanism correctly implemented?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody
- General admin mistakes
- Adding/removing margin to positions
- The `GetPositionValue` function from the GTL contract

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

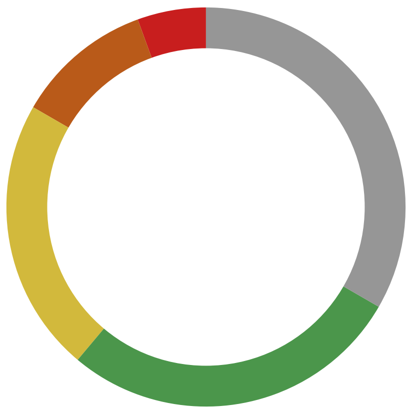
During our assessment on the scoped GTE – Perp contracts, we discovered 18 findings. One critical issue was found. Two were of high impact, four were of medium impact, five were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Liquid Labs, Inc. in the Discussion section (4. ↗).

Based on the number of severe findings uncovered during the audit, it is our opinion that the project is not yet ready for production. We strongly advise a comprehensive reassessment before deployment to help identify any potential issues or vulnerabilities introduced by necessary fixes or changes. We also recommend adopting a security-focused development workflow, including (but not limited to) augmenting the repository with comprehensive end-to-end tests that achieve 100% branch coverage using any common, maintainable testing framework, thoroughly documenting all function requirements, and training developers to have a security mindset while writing code.

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	1
🟧 High	2
🟨 Medium	4
🟩 Low	5
⬜ Informational	6



2. Introduction

2.1. About GTE – Perp

Liquid Labs, Inc. contributed the following description of GTE – Perp:

> GTE is a Perpetual Futures protocol, with a fully on chain CLOB.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and

Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

GTE – Perp Contracts

Type	Solidity
Platform	MegaETH

Target	gte-contracts
Repository	https://github.com/liquid-labs-inc/gte-contracts ↗
Version	`1561154b9066a27dc87479673fd9104de6d1899a`
Programs	`contracts/perps/*`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 3.4 person-weeks. The assessment was conducted by two consultants over the course of two calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Katerina Belotskaia
Engineer
kate@zellic.io ↗

Weipeng Lai
Engineer
weipeng.lai@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

May 5, 2025	Start of primary review period
May 6, 2025	Kick-off call
May 16, 2025	End of primary review period

3. Detailed Findings

3.1. Incorrect position update order relative to `isLiquidatable` check in `processMakerFill` permits fund theft

Target	ClearingHouseLib		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The `processMakerFill` function performs an `isLiquidatable` check to ensure that settling a maker's limit order does not immediately make the maker's account liquidatable.

```
function processMakerFill(ClearingHouse storage self, bytes32 asset,
    MakerFillParams memory params)
    internal
    returns (bool unfillable)
{
    // [...]

    if (self.isLiquidatable(params.maker, params.subaccount, margin)) {
        market.setPosition(params.maker, params.subaccount, position); //
    revert to old position
        return true;
    }


    // [...]
    if (result.fullClose)
    self.positions[params.maker][params.subaccount].remove(asset);
    else self.positions[params.maker][params.subaccount].add(asset);

    // [...]
}
```

However, the `asset` data is added to the account's `positions` EnumerableSet only *after* this `isLiquidatable` check. Consequently, if the maker does not have an existing position in the specified `asset` market, the `isLiquidatable` check does not account for the newly settled position in that market.

If the `isLiquidatable` check does not consider this new position, the maker's account can incur significant bad debt if the trade execution price deviates substantially from the index price. An attacker filling the limit order can then capture this bad debt as profit.

Impact

An attacker can profit from this vulnerability by performing the following steps:

1. Clear the order book by filling existing orders to allow subsequent orders to be settled at arbitrary prices.
2. Use a sacrificial account (account A) to place a limit order at a price designed to create immediate debt for account A upon settlement.
3. Use another account (account B) to fill this order, generating unrealized profit and loss (PNL) for account B from account A's debt.
4. Use a third sacrificial account (account C) to place a limit order that, when matched, closes account B's position.
5. Use account B to fill account C's order, thereby converting account B's unrealized PNL into realized profit.
6. Abandon accounts A and C, retaining the extracted funds in account B.

Because orders can be settled at arbitrary prices once the order book is cleared and there is no cap on trading volume, an attacker's potential profit can significantly outweigh the cost of performing this attack. Such an attack could potentially result in the draining of all funds from the contract.

Recommendations

We recommend performing the `isLiquidatable` check only after the account's `positions` EnumerableSet is updated to include the new `asset` market.

Remediation

This issue has been acknowledged by Liquid Labs, Inc., and fixes were implemented in the following commits:

- `32a0254d` ↗
- `fbdff56d` ↗

3.2. The `isLong` flag for a position is not reset to `false` during liquidation, preventing users from opening long positions

Target	PositionLib		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

During an administrative liquidation, the `isLong` flag of a position is not reset to `false` if the `amount` of that position becomes zero.

```
function liquidatePosition(Position storage self, uint256 quoteTraded,
    uint256 baseTraded)
    internal
    returns (PositionLiquidateResult memory result)
{
    // [...]
    if (position.amount == 0) {
        result.close = true;
        delete margin;
    }
    //[...]
}
```

Following such a liquidation, the account associated with this position may post sell limit orders. If a user then attempts to open a long position, their buy order could match one of these sell limit orders. This match triggers an attempt to update the seller's position by calling the `_updatePosition` function.

Within the `_updatePosition` function, if the `side` parameter is `Side.SELL` and the position's `isLong` flag is `true`, the `_close` function is called. The `_close` function subsequently fails because the position's `amount` is zero.

```
function _updatePosition(Position memory self, Side side, uint256 quoteTraded,
    uint256 baseTraded, uint256 leverage)
    private
    pure
    returns (PositionUpdateResult memory result)
{
    bool openLong = side == Side.BUY && (self.isLong || self.amount == 0);
    bool openShort = side == Side.SELL && !self.isLong; // isLong at default
    value implies at least open, so no need to check amount

    Position memory position = self;

    if (openLong || openShort) {
        int256 marginDelta = _open(position, side, quoteTraded, baseTraded,
    leverage, result.oiDelta);
        result.marginDelta = marginDelta;
        result.collateralDelta = marginDelta;
    } else {
        result = _close(position, side, quoteTraded, baseTraded, leverage);
    }
}
```

Consequently, this issue prevents users from successfully opening certain long positions.

Impact

A malicious actor can exploit this vulnerability to prevent users from opening new long positions by performing the following actions:

- They can open a long position with the minimum-allowable amount and maximum leverage, then wait for an administrator to liquidate this position when it becomes liquidatable.
- After the position is liquidated, they can repeatedly post sell limit orders for a minimal amount, targeting the best ask price. Since these orders cannot be filled when matched, this action blocks other users from opening new long positions.

Recommendations

We recommend resetting the `isLong` flag to `false` for a position if its `amount` becomes zero during liquidation.

Remediation

This issue has been acknowledged by Liquid Labs, Inc., and a fix was implemented in commit a969476a ↗.

3.3. The `_gtlHook` is not triggered when a GTL limit order is fully filled, resulting in inflated GTL `totalAssets`

Target	PerpCLOB		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

When GTL's non-reduce-only limit order is filled, the `_gtlHook` function should be triggered to update the GTL's `orderbookCollateral`. However, in the `_matchIncomingOrder` function within `PerpCLOB`, `_gtlHook` is only triggered if the `orderRemoved` flag is `false`.

```
function _matchIncomingOrder(
    PerpBook storage ds,
    Order storage matchedOrder,
    Order memory incomingOrder,
    bool amountIsBase // true if incomingOrder is in base, false if in quote
) internal returns (MatchData memory matchData) {
    // [...]
    bool orderRemoved = matchData.baseDelta == matchedOrder.amount;

    if (matchData.baseDelta > 0) {
        // [...]

        if (unfillable) {
            _removeUnfillableOrder(ds, matchedOrder);
            return MatchData(0, 0, 0, 0);
        } else if (!orderRemoved) {
            if (matchedOwner == GTL && !matchedOrder.reduceOnly) {
                _gtlHook(-int256(matchData.quoteDelta.fullMulDiv(1e18,
    matchedOrder.leverage)));
            }
            // [...]
        }
        // [...]
    }
    // [...]
}
```

Consequently, if a GTL limit order is fully filled (which sets `orderRemoved` to `true`), `_gtlHook` is not triggered. This prevents the corresponding decrease in GTL's `orderbookCollateral`.

Impact

GTL's `totalAssets` value becomes inflated. This could allow a user to withdraw more assets from GTL than they are entitled to based on their original shares.

Recommendations

We recommend triggering `_gtlHook` in the `_matchIncomingOrder` function, regardless of the state of the `orderRemoved` flag.

Remediation

This issue has been acknowledged by Liquid Labs, Inc., and a fix was implemented in commit `8f495ef1` ↗.

3.4. Stale price pointer after expired-order removal

Target	PerpCLOB		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

Within the `_matchIncomingAsk` function, the code branch for removing expired orders does not update the `bestBid` variable using `ds.getBestBid()` before executing the `continue` statement. Consequently, `bestBid` can become stale, potentially referring to the price of a removed order in the next loop iteration.

```
function _matchIncomingAsk(PerpBook storage ds, Order memory incomingOrder,
    bool amountIsBase)
    internal
    returns (uint256 totalQuoteTokenReceived, uint256 totalBaseTokenSent,
    uint256 totalTakerFeeInQuote)
{
    uint256 bestBid = ds.getBestBid();

    while (bestBid >= incomingOrder.price && incomingOrder.amount > 0) {
        // [...]

        if (bestBidOrder.isExpired()) {
            _removeUnfillableOrder(ds, bestBidOrder);
            continue;
        }
        // [...]
        bestBid = ds.getBestBid();
    }
    // [...]
}
```

Similarly, in the `_matchIncomingBid` function, the code does not update the `bestAsk` variable using `ds.getBestAsk()` after removing an expired order and before the `continue` statement.

```
function _matchIncomingBid(PerpBook storage ds, Order memory incomingOrder,
    bool amountIsBase)
    internal
    returns (MatchResult memory result)
```

```
{
    uint256 bestAsk = ds.getBestAsk();

    while (bestAsk <= incomingOrder.price && incomingOrder.amount > 0) {
        // [...]
        if (bestAskOrder.isExpired()) {
            _removeUnfillableOrder(ds, bestAskOrder);
            continue;
        }
        // [...]
        bestAsk = ds.getBestAsk();
    }
    // [...]
}
```

Impact

If, after removing an expired order, no orders remain at that price, the subsequent `_matchIncomingOrder` call would use an empty order. This will cause the function to fail when it tries to remove this empty order at the end of `_matchIncomingOrder`.

As a result, a filler's transaction will fail if there is an expired order at the end of the limit order array for a matched price.

Recommendations

We recommend updating the `bestBid` variable (in `_matchIncomingAsk`) and the `bestAsk` variable (in `_matchIncomingBid`) immediately after an expired order is removed and before the `continue` statement is executed.

Remediation

This issue has been acknowledged by Liquid Labs, Inc., and a fix was implemented in commit `2d5e753c` ↗.

3.5. Time-weighted average price is manipulatable

Target	PriceHistoryLib		
Category	Business Logic	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

The PriceHistoryLib library's `snapshot` function records the last traded price. If a price snapshot for a given `timestamp` already exists, the function overwrites it with the new `price`. The `twap` function subsequently uses this price history to calculate the time-weighted average price (TWAP) mark price. This TWAP mark price influences the funding rate post settlement.

```
function snapshot(PriceHistory storage history, uint256 price,
    uint256 timestamp) internal {
    uint256 length = history.snapshots.length;

    if (length > 0 && history.snapshots[length - 1].timestamp == timestamp) {
        history.snapshots[length - 1].price = price;
    } else {
        history.snapshots.push(PriceSnapshot(price, timestamp));
    }
}
```

This design presents two vulnerabilities:

1. The `snapshot` function does not differentiate between bid and ask prices when recording trades. This allows an attacker to systematically execute trades on only one side of the order book (e.g., exclusively bids or asks), potentially skewing the resultant TWAP.
2. The function records price snapshots without considering associated trade volumes. This enables an attacker to exert undue influence on the TWAP by executing numerous trades at the minimum permissible volume, thereby giving these small-volume trades disproportionate weight.

Impact

An attacker can exploit these vulnerabilities by repeatedly executing trades at the prevailing best ask (or best bid) price using minimal volume. Such activity introduces a high number of biased price points into the price history, leading to an artificially inflated or deflated TWAP mark price.

Manipulation of the TWAP price directly affects the calculated funding rate, creating an opportunity for the attacker to profit from this distortion.

Recommendations

We recommend separating the price history into distinct bid and ask histories. The mark TWAP should then be calculated as the average of the bid TWAP and the ask TWAP. Additionally, trade volume should be considered when calculating the TWAP.

Remediation

This issue has been acknowledged by Liquid Labs, Inc..

3.6. Incorrect shortcut used in `isLiquidatable`

Target	ClearingHouseLib, PerpManager		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The `isLiquidatable` function in the ClearingHouseLib library directly returns `true` if the `margin` input is less than zero.

```
function isLiquidatable(ClearingHouse storage self, address account,
    uint256 subaccount, int256 margin)
    internal
    view
    returns (bool)
{
    if (margin < 0) return true;

    bytes32[] memory positions = self.getPositions(account, subaccount);

    int256 totalPnL;
    uint256 totalMarginReq;
    int256 pnl;
    uint256 marginReq;
    for (uint256 i; i < positions.length; ++i) {
        (pnl, marginReq) =
    self.market[positions[i]].getLiquidatableUPnLAndMarginRequirement(account,
    subaccount);

        totalPnL += pnl;
        totalMarginReq += marginReq;
    }

    return (margin + totalPnL) < totalMarginReq.toInt256();
}
```

This shortcut bypasses further profit and loss (PNL) computations. Consequently, an account could be flagged for liquidation despite potentially possessing sufficient unrealized profit to cover the margin requirement.

A similar issue exists in the PerpManager contract. Its `isLiquidatable` view function includes the shortcut `if (fundingPaymentResult.debt > 0) return true;`, which also incorrectly bypasses further PNL computations.

Impact

This flaw may lead to the incorrect liquidation of user accounts that have sufficient unrealized profits across markets to meet overall margin requirements.

Recommendations

We recommend removing these shortcuts in the solvency calculation logic to ensure all PNL is considered before determining an account's liquidatable status.

Remediation

This issue has been acknowledged by Liquid Labs, Inc., and a fix was implemented in commit `1222a4dd` ↗.

3.7. A malicious actor could block a market if `minLimitOrderAmountInBase` is set too low

Target	PerpBookLib		
Category	Business Logic	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

The `minLimitOrderAmountInBase` setting for a market specifies the minimum base asset amount acceptable for placing a limit order on the order book.

The lower bound for `minLimitOrderAmountInBase` is 10, as defined by the constant `MIN_MIN_LIMIT_ORDER_AMOUNT_BASE`:

```
uint256 constant MIN_MIN_LIMIT_ORDER_AMOUNT_BASE = 10;
```

If a market sets `minLimitOrderAmountInBase` to this low value, users can place orders with this minimum amount at the minimum-allowed price. When such an order is filled, `matchData.quoteDelta` can become zero due to rounding down during division within the `_matchIncomingOrder()` function.

```
function _matchIncomingOrder(
    PerpBook storage ds,
    Order storage matchedOrder,
    Order memory incomingOrder,
    bool amountIsBase // true if incomingOrder is in base, false if in quote
) internal returns (MatchData memory matchData) {
    // [...]

    if (amountIsBase) {
        // denominated in base
        matchData.baseDelta = matchData.matchedAmount
    = matchedOrder.amount.min(incomingOrder.amount);
        matchData.quoteDelta
    = matchData.baseDelta.fullMulDiv(matchedOrder.price, 1e18);
    } else {
        // denominated in quote
        matchData.baseDelta
    = matchedOrder.amount.min(incomingOrder.amount.fullMulDiv(1e18,
    matchedOrder.price));
```

```
        matchData.quoteDelta
    = matchData.baseDelta.fullMulDiv(matchedOrder.price, 1e18);
        matchData.matchedAmount =
            matchData.baseDelta != matchedOrder.amount ? incomingOrder.amount
    : matchData.quoteDelta;
    }

    // [...]
}
```

Subsequently, the outer function of `_matchIncomingOrder()`, such as `_matchIncomingBid()`, could fail. This failure occurs because a sanity check prevents `result.totalQuoteTokenSent` from being zero when `result.totalBaseTokenReceived` is nonzero.

```
function _matchIncomingBid(PerpBook storage ds, Order memory incomingOrder,
    bool amountIsBase)
    internal
    returns (MatchResult memory result)
{
    uint256 bestAsk = ds.getBestAsk();

    while (bestAsk <= incomingOrder.price && incomingOrder.amount > 0) {
        // [...]
        MatchData memory data = _matchIncomingOrder(ds, bestAskOrder,
    incomingOrder, amountIsBase);

        incomingOrder.amount -= data.matchedAmount;

        result.totalBaseTokenReceived += data.baseDelta;
        result.totalQuoteTokenSent += data.quoteDelta;
        result.totalQuoteTokenFees += data.takerFee;

        bestAsk = ds.getBestAsk();
    }

    if (result.totalBaseTokenReceived == 0 || result.totalQuoteTokenSent == 0)
    {
        if (result.totalBaseTokenReceived != result.totalQuoteTokenSent)
    revert ZeroCostTrade();
    }
}
```

Consequently, aggregations of orders with such a low amount cannot be settled when matched.

Impact

If `minLimitOrderAmountInBase` is set to its minimum value of 10, a malicious actor could block the market by

1. filling all sell orders on the order book, which is easier if the market is newly created.
2. placing numerous sell limit orders with the lowest allowed amount at the lowest allowed price. Since these orders set the best ask price, bid orders will attempt to match with them. However, these orders cause reverts during settlement, preventing users from filling bid orders and effectively blocking the market.

Recommendations

We recommend to increase the `MIN_MIN_LIMIT_ORDER_AMOUNT_BASE` constant. The new minimum value must be set sufficiently high to ensure that `matchData.quoteDelta`, calculated during order matching, does not round down to zero.

Remediation

This issue has been acknowledged by Liquid Labs, Inc., and a fix was implemented in commit `ceba555c` ↗.

3.8. Incorrect `ZeroCostTrade` revert condition

Target	PerpCLOB		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

The `_matchIncomingAsk` function in the PerpCLOB contract includes a check designed to prevent zero-cost trades. The intention is to revert if one of `totalQuoteTokenReceived` or `totalBaseTokenSent` is zero while the other is nonzero.

```
if (totalQuoteTokenReceived == 0 && totalBaseTokenSent == 0) {
    if (totalQuoteTokenReceived != totalBaseTokenSent) revert ZeroCostTrade();
}
```

However, this logic incorrectly uses && instead of `||` in the primary condition. Thus, the check fails to throw a `ZeroCostTrade` where one amount is zero and the other is not.

Impact

The check fails to prevent zero-cost trades. Such trades can occur during order settlement when the base amount is a small nonzero value and the quote amount becomes zero due to rounding down in division.

Recommendations

We recommend modifying the condition to use the `||` operator instead of &&.

```
if (totalQuoteTokenReceived == 0 && totalBaseTokenSent == 0) {
if (totalQuoteTokenReceived == 0 || totalBaseTokenSent == 0) {
```

Remediation

This issue has been acknowledged by Liquid Labs, Inc., and a fix was implemented in commit `fc41a758`↗.

3.9. Unhandled edge case in the `twap` function of PriceHistoryLib

Target	PriceHistoryLib		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

If there is only one `PriceSnapshot` in history and its timestamp is equal to `block.timestamp`, the `elapsedTime` variable will be zero. This causes the `twap` function to fail due to a division-by-zero error.

```
function twap(PriceHistory storage history, uint256 twapInterval)
    internal view returns (uint256) {
    uint256 idx = history.snapshots.length;

    if (idx == 0) return 0;

    PriceSnapshot memory currentSnapshot = history.snapshots[--idx];

    uint256 targetTime = block.timestamp - twapInterval;
    uint256 timePeriod = block.timestamp - currentSnapshot.timestamp;
    uint256 elapsedTime = timePeriod;
    uint256 weightedPrice = currentSnapshot.price * timePeriod;
    uint256 previousTime = currentSnapshot.timestamp;

    while (currentSnapshot.timestamp > targetTime) {
        //[...]
    }

    return weightedPrice / elapsedTime;
}
```

Impact

If this division-by-zero error occurs, the `settleFunding` function, which calls the `twap` function, will also fail. Consequently, the admin cannot update the funding rate in this specific case.

Recommendations

We recommend returning the price value instead of reverting in this specific case to make the `twap` function more robust.

Remediation

This issue has been acknowledged by Liquid Labs, Inc..

3.10. The queueWithdraw function lacks a zero-value check for the shares parameter

Target	GTL		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `queueWithdraw()` function does not validate that the `shares` parameter is nonzero. Consequently, a user can call `queueWithdraw()` with a `shares` value of zero. This action adds an unnecessary withdrawal request to the queue.

```
function queueWithdraw(uint256 shares) external {
    if (queuedShares[msg.sender] + shares > balanceOf(msg.sender))
    revert InsufficientBalance();

    uint256 id = ++_withdrawCounter;

    queuedShares[msg.sender] += shares;
    queuedWithdraw[id] = Withdrawal(msg.sender, shares);
    _withdrawQueue.push(id);

    emit WithdrawQueued(id, msg.sender, shares);
}
```

Impact

If an administrator processes withdrawals according to a schedule for the `processWithdraws` function — for example, processing 100 withdrawals per hour — a malicious actor could add numerous zero-share withdrawal requests to the queue. This could delay withdrawals for legitimate users. Additionally, the administrator would incur higher gas fees for `processWithdraws` if many such malicious requests are present.

Recommendations

We recommend adding a check to ensure that the input `shares` parameter is greater than zero.

Remediation

This issue has been acknowledged by Liquid Labs, Inc., and a fix was implemented in commit `c02f073e` ↗.

3.11. Inflated funding rate during first settlement

Target	FundingLib		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

During the first call to the `settleFunding()` function, the `self.lastFundingTime` storage variable is zero. As a result, the calculation `currentTime - self.lastFundingTime` assigns the current `block.timestamp` value to the `timeSinceLastFunding` variable. This results in an inflated value for traders who were active before the initial funding settlement.

```
function settleFunding(
    FundingRateEngine storage self,
    uint256 markTwap,
    uint256 indexTwap,
    uint256 fundingInterval,
    uint256 maxFundingRate
) internal returns (int256 funding) {
    uint256 currentTime = block.timestamp;
    uint256 timeSinceLastFunding = currentTime - self.lastFundingTime;

    if (timeSinceLastFunding < fundingInterval)
    revert FundingIntervalNotElapsed();

    funding = _getUpdatedFunding({
        markTwap: markTwap,
        indexTwap: indexTwap,
        timeSinceLastFunding: timeSinceLastFunding,
        maxFundingRate: maxFundingRate
    });

    self.cumulativeFunding += funding;
    self.lastFundingTime = currentTime;
}
```

Impact

Traders who are active before the first funding settlement may be charged an inflated funding rate.

Recommendations

We recommend initializing the `lastFundingTime` storage variable for each market when it is created.

Remediation

This issue has been acknowledged by Liquid Labs, Inc., and a fix was implemented in commit `6036436e` ↗.

3.12. Incorrect condition check in `setMinLimitOrderAmountInBase`

Target	PerpBookLib		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `setMinLimitOrderAmountInBase` function contains an incorrect conditional check. The condition compares the `newLimitOrderAmountInBase` parameter with the `MIN_MIN_LIMIT_ORDER_AMOUNT_BASE` constant using the == operator. It should instead use the < operator.

```
function setMinLimitOrderAmountInBase(PerpBook storage self,
    uint256 newLimitOrderAmountInBase) internal {
    if (newLimitOrderAmountInBase == MIN_MIN_LIMIT_ORDER_AMOUNT_BASE)
    revert InvalidMinLimitOrderAmountInBase();

    self.settings.minLimitOrderAmountInBase = newLimitOrderAmountInBase;
}
```

Impact

The incorrect check has two primary consequences:

1. It prevents a valid value, specifically `MIN_MIN_LIMIT_ORDER_AMOUNT_BASE`, from being set as the `minLimitOrderAmountInBase` for a market.
2. It allows invalid values — those less than `MIN_MIN_LIMIT_ORDER_AMOUNT_BASE` — to be set for `minLimitOrderAmountInBase`.

Recommendations

We recommend correcting the conditional check in the `setMinLimitOrderAmountInBase` function as follows:

```
if (newLimitOrderAmountInBase == MIN_MIN_LIMIT_ORDER_AMOUNT_BASE) revert
    InvalidMinLimitOrderAmountInBase();
if (newLimitOrderAmountInBase < MIN_MIN_LIMIT_ORDER_AMOUNT_BASE) revert
```

```
    InvalidMinLimitOrderAmountInBase();
```

Remediation

This issue has been acknowledged by Liquid Labs, Inc., and a fix was implemented in commit 05a84a34 ↗.

3.13. Lack of explicit `asset` validation

Target	PerpManager		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

There is no explicit check for the `asset` parameter in the `postFillOrder`, `postFillCloseOrder`, `postLimitOrder`, and `cancel` functions to ensure that a market exists for the specified asset.

Although using an `asset` parameter corresponding to a nonexistent market causes these functions to revert, it is preferable to detect this condition earlier and revert with a dedicated error message.

Impact

The error messages emitted by these functions when an `asset` parameter corresponding to a nonexistent market is used may not clearly indicate that a market has not been created for the specified asset.

Recommendations

We recommend adding an explicit sanity check for the `asset` parameter at the beginning of these functions to ensure that a market has been created for the specified asset.

Remediation

This issue has been acknowledged by Liquid Labs, Inc..

3.14. Unsafe cast from `int256` to `uint256`

Target	GTL		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In Solidity, type casting from `int256` to `uint256` using `uint256(...)` does not include any bounds or overflow checks. If the result of the `int256(orderbookCollateral) + marginDelta` calculation is less than zero, casting it to `uint256` will produce a very large positive value.

```
function orderUpdated(int256 marginDelta) external onlyPerpManager {
        orderbookCollateral = uint256(int256(orderbookCollateral)
    + marginDelta);
    }
```

Impact

Under normal protocol operation, it is unlikely that `marginDelta` would exceed `orderbookCollateral`, so this issue is considered informational. However, if it does, the unsafe cast may result in an inflated GTL price, which could lead to incorrect pricing and unintended user profit during withdrawal.

Recommendations

To prevent this behavior when casting from `int256` to `uint256`, it is strongly recommended to use the SafeCast library.

Remediation

This issue has been acknowledged by Liquid Labs, Inc..

3.15. Inaccessible market-setting functions via delegate call

Target	PerpCLOBCaller		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `setMaxLimitsPerTx`, `setMaxLimitsExempt`, and `setMinLimitOrderAmountInBase` functions in PerpCLOB are designed to be called via the delegate call to modify the PerpBook's `MarketSettings`.

However, the PerpCLOBCaller contract does not implement wrapper functions to delegate these calls, making them inaccessible.

Additionally, the `reduce` function, which is also intended to be executed via delegate call, is missing from the PerpCLOBCaller implementation.

Impact

As a result, market settings cannot be updated.

Recommendations

Implement the missing delegate-call forwarding functions in PerpCLOBCaller for `setMaxLimitsPerTx`, `setMaxLimitsExempt`, and `setMinLimitOrderAmountInBase`.

Additionally, consider implementing support for the `reduce` function if it is intended to be used.

Remediation

This issue has been acknowledged by Liquid Labs, Inc..

3.16. Centralization risk — owner-controlled deposits to insurance fund using user allowances

Target	PerpManager		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The function `insuranceFundDeposit(...)`, which can only be called by the contract owner or admin, allows the protocol owner to transfer USDC from any account that has approved the PerpManager. Specifically, the function executes

```
function insuranceFundDeposit(address account, uint256 amount)
    external onlyOwnerOrAdmin {
    _getClearingHouse().insuranceFund.deposit(account, amount);
}

function deposit(InsuranceFund storage self, address account, uint256 amount)
    internal {
    self.balance += amount;
    USDC.safeTransferFrom(account, address(this), amount);
    emit InsuranceFundDeposit(account, amount);
}
```

This means that any user who has granted allowance to the PerpManager (e.g., via `approve()` for normal trading) is at risk of having their funds deposited into the insurance fund by the owner without direct consent or action.

Impact

A privileged actor (owner or admin) can arbitrarily move user funds into the insurance fund using standard token allowances, without user intent.

Recommendations

Restrict `insuranceFundDeposit` to only accept funds from the caller.

Additionally, clearly document this behavior to users if this behavior is expected.

Remediation

This issue has been acknowledged by Liquid Labs, Inc..

3.17. Inconsistency between `processWithdraws` and `previewWithdraw`

Target	GTL		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The GTL contract inherits from ERC-4626. However, in this implementation, the standard `withdraw` function is not supported; users cannot directly withdraw their assets. Instead, withdrawal functionality is available only through `queueWithdraw` and the custom admin-only `processWithdraws` function.

The `processWithdraws` handles the actual processing of queued share amounts by calculating the corresponding asset amount to be redeemed. The `previewWithdraw` accepts an asset amount as input and returns a share amount, which is reversed from how `processWithdraws` operates.

This inconsistency between previewing and processing withdrawals can be misleading.

Impact

Users and integrators may misunderstand the withdrawal flow, especially when using `previewWithdraw`, potentially leading to incorrect assumptions about share-to-asset conversions.

Recommendations

Align the function's logic or clearly document the difference in behavior and intended use.

Remediation

This issue has been acknowledged by Liquid Labs, Inc..

3.18. Lack of documentation

Target	Multiple Contracts		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The codebase lacks sufficient documentation and NatSpec annotations. Critical functions, complex logic, and mechanisms are not explained, which makes it difficult to fully understand the system's intended behavior. The absence of clear documentation can lead to misunderstandings for future developers, auditors, and integrators, and increases the risk of incorrect implementation or misuse.

Impact

Code maturity is very important in high-assurance projects. Undocumented code may result in developer confusion, potentially leading to future bugs should the code be modified later on.

In general, a lack of documentation impedes the auditors' and external developers' ability to read, understand, and extend the code. The problem is also carried over if the code is ever forked or reused.

Recommendations

We recommend adding comprehensive documentation, including a high-level system overview, clear explanations of complex logic, NatSpec comments for public and external functions and brief comments to reaffirm developers' understanding. This will improve code maintainability, facilitate audits, and reduce the likelihood of misunderstandings or errors.

Remediation

This issue has been acknowledged by Liquid Labs, Inc..

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Limited test suite

When building a complex contract ecosystem with multiple moving parts and dependencies, comprehensive testing is essential. This includes testing for both positive and negative scenarios. Positive tests should verify that each function's side effect is as expected, while negative tests should cover every revert, preferably in every logical branch.

The test coverage for this project should be expanded to include all contracts, not just surface-level functions. It is important to test the invariants required for ensuring security and also verify mathematical properties as specified in the white paper.

Tests should be added for functions that currently have no coverage at all, such as

- both `insuranceFundWithdraw` functions
- `insuranceFundDeposit(address account, uint256 amount)`
- `setMakerFeeRates`
- `setAccountFeeTier`
- `setTakerFeeRates`
- `setTickSize`
- `revokeAdmin`
- `disapproveOperator`

Additionally, functions `createMarket`, `settleFunding`, `insuranceFundDeposit`, `grantAdmin`, and `setIndexPrice` do not have negative tests for the case that the caller is not an admin.

Therefore, we recommend building a rigorous test suite that includes all contracts to ensure that the system operates securely and as intended.

Good test coverage has multiple effects.

- It finds bugs and design flaws early (preaudit or prerelease).
- It gives insight into areas for optimization.
- It displays code maturity.
- It bolsters customer trust in your product.
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.
- It increases development velocity long-term.

The last point seems contradictory, given the time investment to create and maintain tests. To expand upon that, tests help developers trust their own changes. It is difficult to know if a code refactor — or even just a small one-line fix — breaks something if there are no tests. This is especially true for new developers or those returning to the code after a prolonged absence. Tests have your back here. They are an indicator that the existing functionality *most likely* was not broken by your change to the code.

A limited test suite may lead to

- an increased risk of undetected bugs, especially in edge cases or complex logic paths (e.g., partial fills, liquidation boundaries, or debt edge states);
- a higher likelihood of regressions when code is changed or optimized; and
- a reduced ability to detect inconsistent or unintended state changes, particularly in critical areas like collateral, unrealized/realized PNL, open interest, user balances, and debt tracking.

We recommend to improve the test suite to ensure the following:

- Full functional coverage of all public and internal functions
- Edge-case scenarios (zero values, minimum and maximum values, underflow/overflow boundaries)
- Error-condition testing (e.g., reverts, unauthorized access, failed transfers)
- State assertions after every action, validating that all relevant variables have changed (or not changed) as expected, no unexpected side effects occurred, and balances, positions, and mappings are updated correctly

Additional test-case ideas are outlined in the threat model, section (5. ↗), and could serve as a foundation for expanding coverage.

4.2. The `setIndexPrice()` update frequency

The `setIndexPrice()` function is responsible for updating the index price used across the protocol, a critical input for functions such as funding-rate calculation, liquidation checks, profit and loss (PNL) estimation, and position health monitoring.

While the function is currently designed to be called periodically (e.g., once per hour), it's important to note that the security and correctness of the protocol depend on how frequently this function is triggered. If `setIndexPrice()` is not updated often enough, the on-chain price may drift from the current market price, especially during periods of high volatility or trading activity.

This may lead to the following.

- Delayed reaction to market conditions, which may allow traders to exploit stale prices during funding updates or liquidation checks
- Inaccurate PNL and margin calculations, especially during volatile periods
- Increased risk of incorrect liquidations or missed liquidations due to outdated price references

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: CollateralManager.sol

Function: `settleMakerFill(CollateralManager self, SettleMakerFillParams params)`

This internal function performs maker-side accounting of margin and debt during the matching process after their limit order is partially or fully filled by a taker and calculates the `collateralDelta` to credit an account.

Inputs

- `params.account`
  - **Control**: from `matchedOrder.maker`.
  - **Constraints**: N/A.
  - **Impact**: the address of the maker account.
- `params.subaccount`
  - **Control**: from `matchedOrder.subaccount`.
  - **Constraints**: N/A.
  - **Impact**: subaccount.
- `params.marginDelta`
  - **Control**: the result of `market.processMakerFill` function execution + `fundingPaymentResult.marginDelta`.
  - **Constraints**: N/A.
  - **Impact**: `marginAccount.margin` data of the provided account and subaccount will be updated by the `marginDelta`.
- `params.collateralDelta`
  - **Control**: the result of `market.processMakerFill` function execution - `quoteAmountTraded` (in the case of !reduceOnly and decreasing trade).
  - **Constraints**: N/A.
  - **Impact**: if less than zero, account will be credited by `collateralDelta`.
- `params.debt`
  - **Control**: the result of `market.processMakerFill` function execution +

  `fundingPaymentResult.debt`
    - **Constraints**: N/A.
    - **Impact**: `marginAccount.debt` data of the provided account and subaccount will be updated by the `debt` and `makerFee`.
- `params.makerFee`
    - **Control**: the result of `getClearingHouse().getMakerFee(matchedOwner, matchData.quoteDelta)` function execution.
    - **Constraints**: N/A.
    - **Impact**: `marginAccount.debt` data of the provided account and subaccount will be updated by the `debt` and `makerFee`.
- `params.close`
    - **Control**: set up if subaccount was fully closed.
    - **Constraints**: N/A.
    - **Impact**: if true, `collateralDelta` is increased by `debt`.

Branches and code coverage

Intended branches

- Margin increases correctly when `marginDelta > 0`.
    - [ ] Test coverage
- Margin decreases correctly when `marginDelta < 0`.
    - [ ] Test coverage
- Collateral is updated according to `collateralDelta`.
    - [ ] Test coverage

Function: `settleTakerFill(CollateralManager self, SettleTakerFillArgs args)`

The `settleTakerFill` function processes the taker's side of a trade after a limit order is filled. It processes the margin and debt of an account and handles `collateralDelta`.

Inputs

- `params.account`
    - **Control**: initiate `postLimitOrder` or `postFillOrder` execution.
    - **Constraints**: N/A.
    - **Impact**: the address of the account who place or fill order.

- `params.subaccount`
    - **Control**: full control.
    - **Constraints**: N/A.
    - **Impact**: subaccount.
- `params.settlement`
    - **Control**: full control.
    - **Constraints**: Must be a valid settlement type.
    - **Impact**: Defines how the collateral funds will be provided.
- `params.marginDelta`
    - **Control**: the result of `self.market[asset].postLimitOrder(asset, account, args)/self.market[asset].postFillOrder(asset, account, args)` function execution + `fundingPaymentResult.marginDelta`.
    - **Constraints**: N/A.
    - **Impact**: `marginAccount.margin` data of the provided account and subaccount will be updated by the `marginDelta`.
- `params.collateralDelta`
    - **Control**: the result of `self.market[asset].postLimitOrder(asset, account, args)/self.market[asset].postFillOrder(asset, account, args)` function execution
    - **Constraints**: N/A.
    - **Impact**: this amount is processed by `handleCollateralDelta` to be transferred to or from the account or debited/credited, depends on settlement.
- `params.debt`
    - **Control**: the result of `self.market[asset].postLimitOrder(asset, account, args)/self.market[asset].postFillOrder(asset, account, args)` function execution + `fundingPaymentResult.debt`
    - **Constraints**: N/A.
    - **Impact**: `marginAccount.debt` data of the provided account and subaccount will be updated by the `debt`.
- `params.takerFee`
    - **Control**: the result of `self.market[asset].postLimitOrder(asset, account, args)/self.market[asset].postFillOrder(asset, account, args)` function execution.
    - **Constraints**: N/A.
    - **Impact**: if `collateralDelta > 0`, `collateralDelta` is increased by `takerFee` amount, otherwise `debt` is increased by `takerFee`.
- `params.close`
    - **Control**: `self.positions[account][args.subaccount].length() == 0`

- **Constraints**: N/A.
- **Impact**: if true, `collateralDelta` is increased by `debt`.

Branches and code coverage

Intended branches

- Positive `marginDelta` increases available margin.
  - ☐ Test coverage
- Negative `marginDelta` reduces margin correctly.
  - ☐ Test coverage
- `settlement == INSTANT`.
  - ☐ Test coverage
- `settlement == ACCOUNT`.
  - ☐ Test coverage

Negative behavior

- Reverts if position is closed and `collateralDelta < 0` after debt deduction.
  - ☐ Negative test

5.2. Module: GTL.sol

Function: `cancelWithdraw(uint256 id)`

This function allows a user to cancel their own previously queued withdrawal request, which is identified by the `id` parameter.

Inputs

- `id`
  - **Control**: Full control.
  - **Constraints**: `id` must correspond to a `queuedWithdraw` initiated by the caller.
  - **Impact**: Affects `queuedShares`, `queuedWithdraw`, and `_withdrawQueue`.

Branches and code coverage

Intended branches

- Removes the specified withdrawal from the `queuedWithdraw` mapping.

- ☐ Test coverage
- Decreases the user's total `queuedShares` by the amount of shares associated with the canceled withdrawal.
  - ☐ Test coverage
- Updates the `_withdrawQueue` data.
  - ☐ Test coverage
- Emits a `WithdrawCanceled` event.
  - ☐ Test coverage

Negative behavior

- Reverts if the user tries to cancel a withdrawal they do not own.
  - ☐ Test coverage

Function: `orderUpdated(int256 marginDelta)`

This function is intended to be called by the `_gtlHook` function in the PerpManager contract to reflect changes in GTL's collaterals on the order book.

Inputs

- `marginDelta`
  - **Control**: Controlled by the PerpManager contract.
  - **Constraints**: Must not cause `orderbookCollateral` to become negative.
  - **Impact**: Updates the `orderbookCollateral`.

Branches and code coverage

Intended branches

- Updates the `orderbookCollateral`.
  - ☐ Test coverage

Negative behavior

- Reverts if called by an address other than `perpManager`.
  - ☐ Test coverage

Function: `processWithdraws(uint256 num)`

This function is designed to process a specified number (`num`) of withdrawal requests from a queue. For each withdrawal request, it converts the user's shares to assets, transfers the assets to the user, burns the user's shares, and updates the queue and related storage.

Inputs

- `num`
    - **Control**: Controlled by the admin/owner.
    - **Constraints**: Must not exceed the length of the withdraw queue.
    - **Impact**: Determines how many withdrawal requests to process.

Branches and code coverage

Intended branches

- Removes the withdrawals from the `queuedWithdraw` mapping.
    - ☐ Test coverage
- Dequeues `num` withdrawal requests from `_withdrawQueue`.
    - ☐ Test coverage

Negative behavior

- Reverts if called by non-admin users.
    - ☐ Test coverage
- Reverts if `num` exceeds the total number of withdrawals currently in the queue.
    - ☐ Test coverage
- Reverts if the contract lacks sufficient USDC for withdrawals.
    - ☐ Test coverage
- Burns the user's shares for each withdrawal.
    - ☐ Test coverage
- Transfers assets to the user for each withdrawal.
    - ☐ Test coverage
- Emits a `WithdrawProcessed` event for each withdrawal.
    - ☐ Test coverage

Function: `queueWithdraw(uint256 shares)`

This function allows a user to request queuing a specific number of `shares` for later withdrawal. It updates the `queuedShares` mapping to reflect the total shares the user has queued. The function records the withdrawal request details in the `queuedWithdraw` mapping and adds the withdrawal ID to the `_withdrawQueue` array.

Inputs

- `shares`
  - **Control**: Full control.
  - **Constraints**: Must not exceed `balanceOf(msg.sender) - queuedShares[msg.sender]`. There is no explicit check to ensure `shares > 0`.
  - **Impact**: The amount requested for withdrawal.

Branches and code coverage

Intended branches

- Updates user's `queuedShares`.
  - ☐ Test coverage
- Updates data in `queuedWithdraw[id]`.
  - ☐ Test coverage
- Adds ID to `_withdrawQueue`.
  - ☐ Test coverage
- Emits a `WithdrawQueued` event.
  - ☐ Test coverage

Negative behavior

- Reverts if the `shares` exceeds `balanceOf(msg.sender) - queuedShares[msg.sender]`.
  - ☐ Test coverage

5.3. Module: PerpCLOB.sol

Function: `cancel(byte[32] asset, address account, uint256[] orderIds)`

This function cancels previously placed limit orders. The account should be an owner of all orders from `orderIds`. Provided orders are removed from the order book (if they exist), and the `totalCollateralRefunded` is returned from this function.

This function ensures the order belongs to the account and the order exists.

Branches and code coverage

Intended branches

- One `reduceOnly` order — `totalCollateralRefunded` is zero.
  - ☐ Test coverage
- Successfully cancels all provided active limit orders.
  - ☐ Test coverage
- Emits `OrderCanceled` events with correct data.
  - ☐ Test coverage
- Orders are removed from the order book.
  - ☐ Test coverage
- `orderbookCollateral` in the GTL contract is updated correctly.
  - ☐ Test coverage
- `quoteTokenOpenInterest` is updated properly when `side == Side.BUY`.
  - ☐ Test coverage
- `baseTokenOpenInterest` is updated properly when `side == Side.SELL`.
  - ☐ Test coverage

Negative behavior

- `order.owner != account`.
  - ☐ Negative test
- if order is null, this order will be skipped, and execution is not revert.
  - ☐ Negative test
- `orderIds` contains duplicate IDs. All duplicate IDs are skipped, and execution is not revert.
  - ☐ Negative test

Function: `reduce(byte[32] asset, address account, CLOBReduceArgs args)`

This function reduces the amount of an existing open limit order (or fully cancels it). This function allows the user to lower the unfilled portion of an order without changing its price, side, or leverage.

When the order is reduced, the protocol updates the order's amount, refunds the collateral associated with the removed amount, and keeps the order live in the order book (or cancels it if the amount becomes zero).

Branches and code coverage

Intended branches

- Updates `orderAmount` by correct delta.
  - ☐ Test coverage
- Refunds the correct collateral amount (`quoteAmount` * price / leverage).
  - ☐ Test coverage
- `orderbookCollateral` in the GTL contract is updated correctly.
  - ☐ Test coverage
- Order is fully canceled when `block.timestamp > cancelTimestamp` even if `amountInBase < orderAmount`.
  - ☐ Test coverage
- Order is fully canceled when `args.amountInBase >= orderAmount`.
  - ☐ Test coverage
- Order is fully canceled when `orderAmount - args.amountInBase < ds.settings.minLimitOrderAmountInBase`.
  - ☐ Test coverage
- `quoteTokenOpenInterest` is updated properly when `side == Side.BUY`.
  - ☐ Test coverage
- `baseTokenOpenInterest` is updated properly when `side == Side.SELL`.
  - ☐ Test coverage

Negative behavior

- Order does not exist.
  - ☐ Negative test
- Tries to cancel again.
  - ☐ Negative test
- Account is not an order owner.
  - ☐ Negative test
- Reverts if provided `amountInBase` is greater than `orderAmount`.
  - ☐ Negative test
- Reverts if provided `amountInBase` is zero.
  - ☐ Negative test

Function: `_executeAskLimitOrder(PerpBook ds, Order newOrder, LimitOrderType limitOrderType)`

This internal function handles the matching and placement of a taker ask limit order (a `SELL` order) against the bid side of the order book. If the ask is not fully filled, the remaining portion can be placed into the order book.

Branches and code coverage

Intended branches

- Order fully matches against existing asks.
  - ☐ Test coverage
- Order partially matches — remainder is placed on the book.
  - ☐ Test coverage
- Order does not match at all, and the entire order is placed on the book.
  - ☐ Test coverage

Negative behavior

- `limitOrderType == LimitOrderType.POST_ONLY && BestBid >= newOrder.price`.
  - ☐ Negative test
- `numAsks > maxNumOrders` and `newOrder.price > MaxAskPrice`.
  - ☐ Negative test

Function: `_executeBidLimitOrder(PerpBook ds, Order newOrder, LimitOrderType limitOrderType)`

This internal function handles the matching and placement of a taker bid limit order (a `BUY` order) against the ask side of the order book. If the bid is not fully filled, the remaining portion can be placed into the order book.

Branches and code coverage

Intended branches

- Order fully matches against existing asks.
  - ☐ Test coverage
- Order partially matches — remainder is placed on the book.
  - ☐ Test coverage

- Order does not match at all, and the entire order is placed on the book.
    - ☐ Test coverage

Negative behavior

- `limitOrderType == LimitOrderType.POST_ONLY && BestAsk <= newOrder.price`.
    - ☐ Negative test
- `numBids > maxNumOrders` and `newOrder.price < MinBidPrice`.
    - ☐ Negative test

Function: `_matchIncomingAsk(PerpBook ds, Order incomingOrder, bool amountIsBase)`

This internal function matches an incoming taker ask order (`SELL` order) against existing bid orders in the order book. The function walks through available bids starting from the highest bid price, ensuring that the ask fills at the best possible prices for the seller.

It continues matching until the ask amount is fully matched or there are no more available bids with a price greater than or equal to `incomingOrder.price`.

Branches and code coverage

Intended branches

- The order book is empty, there is no match, and as a result, `totalQuoteTokenReceived` and `totalBaseTokenSent` are zero.
    - ☐ Test coverage
- Correct `totalQuoteTokenReceived` and `totalBaseTokenSent` are calculated across multiple matches.
    - ☐ Test coverage
- The order book contains only one expired order. The expired order is removed without reverting.
    - ☐ Test coverage
- The order book contains only one eligible order with minimum amount.
    - ☐ Test coverage
- The incoming ask is fully filled when the taker ask amount equals the bid.
    - ☐ Test coverage
- The incoming ask is fully filled using multiple bids.
    - ☐ Test coverage

- The incoming ask is partially filled when the taker ask amount is smaller than the bid. The remaining amount is returned correctly.
  - ☐ Test coverage
- All bids are lower the taker's limit price — no match.
  - ☐ Test coverage
- The incoming ask has the same price as the bid limit.
  - ☐ Test coverage
- `totalTakerFeeInQuote` is calculated correctly for one match.
  - ☐ Test coverage
- `totalTakerFeeInQuote` is calculated correctly for multiple matches.
  - ☐ Test coverage

Function: `_matchIncomingBid(PerpBook ds, Order incomingOrder, bool amountIsBase)`

This internal function matches an incoming taker bid order (`BUY` order) against available ask orders in the order book. This function iterates through existing ask orders, starting from the lowest ask price, and fills the taker's bid as much as possible until either the bid amount is fully matched or there are no more available asks with a price less than or equal to `incomingOrder.price`.

Branches and code coverage

Intended branches

- The order book is empty, there is no match, and as a result, `totalBaseTokenReceived` and `totalQuoteTokenSent` are zero.
  - ☐ Test coverage
- Correct `totalBaseTokenReceived` and `totalQuoteTokenSent` are calculated across multiple matches.
  - ☐ Test coverage
- The order book contains only one expired order. The expired order is removed without reverting.
  - ☐ Test coverage
- The order book contains only one eligible order with minimum amount.
  - ☐ Test coverage
- The incoming bid is fully filled when the taker bid amount equals the ask.
  - ☐ Test coverage

- The incoming bid is fully filled using multiple asks.
    - ☐ Test coverage
- The incoming bid is partially filled when the taker bid amount is smaller than the ask. The remaining amount is returned correctly.
    - ☐ Test coverage
- All asks are above the taker's limit price — no match.
    - ☐ Test coverage
- Ask has the same price as the bid limit.
    - ☐ Test coverage
- `totalQuoteTokenFees` is calculated correctly for one match.
    - ☐ Test coverage
- `totalQuoteTokenFees` is calculated correctly for multiple matches.
    - ☐ Test coverage

Function: `_matchIncomingOrder(PerpBook ds, Order matchedOrder, Order incomingOrder, bool amountIsBase)`

This internal function handles a single match between an incoming taker order and a matched maker order during the `_matchIncomingBid` or `_matchIncomingAsk` functions' execution. This function executes the maker fill logic and performs cleanup or updates to the matched maker order. This function prevents self-trade by removing an order if the maker is the same user.

Additionally, this function

- calculates the matched `baseDelta` and `quoteDelta`,
- executes maker-side fill logic,
- updates or removes the maker order in the order book, and
- returns the `MatchData`, which contains `matchedAmount`, `baseDelta`, `quoteDelta`, and `takerFee`.

Branches and code coverage

Intended branches

- The order book contains only the opposite order of the same account. The order was removed, and `matchedAmount`, `baseDelta`, `quoteDelta`, and `takerFee` are zero.
    - ☐ Test coverage
- `matchedOrder` is `reduceOnly`.
    - ☐ Test coverage

- `amountIsBase` is true for the `incomingOrder`. The `baseDelta`, `quoteDelta`, and `matchedAmount` are calculated correctly.
  - ☐ Test coverage
- `amountIsBase` is false for the `incomingOrder`. The `baseDelta`, `quoteDelta` and `matchedAmount` are calculated correctly.
  - ☐ Test coverage
- When `baseDelta > 0`, the `makerFee`, maker's position, and margin are updated correctly.
  - ☐ Test coverage
- Fully matches `matchedOrder`, and it is removed from the order book.
  - ☐ Test coverage
- Partially matches `matchedOrder`, and the amount is updated.
  - ☐ Test coverage

Function: `_removeUnfillableOrder(PerpBook ds, Order order)`

This internal function removes a limit order from the order book if the maker and taker are the same (`self-trade`), the order has expired, and the maker position becomes liquidatable during the `processMakerFill` execution.

Branches and code coverage

Intended branches

- `collateralRefund` is calculated correctly.
  - ☐ Test coverage
- `orderbookCollateral` in the GTL contract is updated correctly.
  - ☐ Test coverage
- The order is fully removed from the order book.
  - ☐ Test coverage
- `quoteTokenOpenInterest` is updated properly when `side == Side.BUY`.
  - ☐ Test coverage
- `baseTokenOpenInterest` is updated properly when `side == Side.SELL`.
  - ☐ Test coverage

5.4. Module: PerpManager.sol

Function: `backstopLiquidate(address account, uint256 subaccount, uint256 leverage, Settlement settlement)`

The `positionTakeover` function allows a liquidator to take over the entire position of the liquidatee that has become liquidatable.

Inputs

- `account`
    - **Control**: Full control.
    - **Constraints**: A valid address that own a position in the specified subaccount. The account must be eligible for liquidation.
    - **Impact**: Identifies which user's subaccount will be liquidated.
- `subaccount`
    - **Control**: Full control.
    - **Constraints**: A valid subaccount of `account` — must contain active positions, which can be liquidated.
    - **Impact**: Determines the subaccount that will be liquidated.
- `leverage`
    - **Control**: Full control.
    - **Constraints**: Must be a valid leverage value.
    - **Impact**: new applied leverage.
- `settlement`
    - **Control**: Full control.
    - **Constraints**: Must be a valid settlement type.
    - **Impact**: Decides where the collateral goes.

Branches and code coverage

Intended branches

- `marginDelta`, `gtlRpnl` and `liquidateeRpnl` are calculated correctly.
    - ☑ Test coverage

Negative behavior

- Reverts if position is not liquidatable.
    - ☐ Negative test

- Reverts if position amount is zero.
    - ☐ Negative test
- Invalid leverage.
    - ☐ Negative test
- Caller is not a GTLOperator.
    - ☐ Negative test

Function call analysis

- `self.isLiquidatable({account: account, subaccount: subaccount, margin: self.collateralManager.getMarginAccountBalance(account, subaccount).toInt256() - liquidateeFundingPayment})`
    - **What is controllable?** `account`, `subaccount`, and `settlement`.
    - **If the return value is controllable, how is it used and how can it go wrong?** If `margin` is less than zero, it returns true, even if `totalPnL` covers the debt.
    - **What happens if it reverts, reenters or does other unusual control flow?** There are not any problems here.

Function: `cancel(byte[32] asset, address account, CancelArgs args)`

The `cancel` function allows an account or approved operator of an account to manually cancel previously submitted limit orders that are still active (not yet filled or expired).

Inputs

- `asset`
    - **Control**: Full control.
    - **Constraints**: Market should exist.
    - **Impact**: Determines from which `asset` market the orders are canceled.
- `account`
    - **Control**: Full control.
    - **Constraints**: Only the order owner (or authorized operator) can cancel the order.
    - **Impact**: Owner of the orders.
- `args.orderIds`
    - **Control**: Full control.
    - **Constraints**: Active orders.
    - **Impact**: Orders to be canceled.

- `args.settlement`
  - **Control**: Full control.
  - **Constraints**: `ACCOUNT` and `INSTANT`.
  - **Impact**: Settlement mode for the collateral refund.

Branches and code coverage

Intended branches

- `reduceOnly` order — no refund.
  - ☐ Test coverage
- Successfully cancels all active limit orders.
  - ☐ Test coverage
- `INSTANT` settlement.
  - ☐ Test coverage
- `ACCOUNT` settlement.
  - ☐ Test coverage
- Refunds correct margin.
  - ☐ Test coverage
- No orders exist. No margin is refunded.
  - ☐ Test coverage
- `orderIds` contains multiple valid orders and one invalid, and it is skipped.
  - ☐ Test coverage
- `orderIds` contains duplicated orders, and they are all skipped, except the first one.
  - ☐ Test coverage
- Cancels the last order in the book.
  - ☐ Test coverage

Negative behavior

- The caller is not an owner or approved operator.
  - ☐ Negative test

Function: `deposit(address account, uint256 amount)`

This function transfers `amount` of the USDC tokens from the provided `account` to this contract and increases the `collateral[account]` balance by this `amount`. It emits the `Deposit` event.

Inputs

- `account`
  - **Control**: Full control.
  - **Constraints**: The caller should be the account itself or an approved operator of this account.
  - **Impact**: The deposit will be made on behalf of the account.
- `amount`
  - **Control**: Full control.
  - **Constraints**: The account should own enough USDC tokens to transfer.
  - **Impact**: The amount of USDC tokens to be deposited.

Branches and code coverage

Intended branches

- A `Deposit` event has been emitted.
  - ☐ Test coverage
- The `collateral` balance of the `account` has been increased by the `amount`.
  - ☐ Test coverage
- The USDC balance of this contract has been increased by the `amount`.
  - ☐ Test coverage
- The USDC balance of the account has been decreased by the `amount`.
  - ☐ Test coverage

Negative behavior

- The caller is not an account or approved operator of this account.
  - ☐ Negative test
- Account does not own enough USDC tokens.
  - ☐ Negative test

Function call analysis

- `MarginAccountLib.deposit(this._getClearingHouse().marginAccount[account], account, amount) -> USDC.safeTransferFrom(account, address(this), amount)`
  - **What is controllable?** `account` and `amount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.

    - **What happens if it reverts, reenters or does other unusual control flow?** It reverts when the account does not have enough USDC tokens or if the allowance is insufficient.
- `MarginAccountLib.deposit(this._getClearingHouse().marginAccount[account], account, amount) -> self.creditAccount(account, amount)`
    - **What is controllable?** `account` and `amount`.
    - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** There are no problems; the `collateral` balance will be increased by the amount.

Function: `postFillCloseOrder(byte[32] asset, address account, PostFillOrderArgs args)`

This function executes a market fill intended to close an existing position for a given account. It attempts to match an opposing order from the order book based on the provided parameters. If the `fillOrderType` is `FILL_OR_KILL`, the entire requested amount must be filled or the transaction reverts. If it is `IMMEDIATE_OR_CANCEL`, it fills as much as possible and cancels the rest.

Inputs

- `asset`
    - **Control**: Full control.
    - **Constraints**: Must be a valid, supported market.
    - **Impact**: Determines which asset the order is for.
- `account`
    - **Control**: Full control.
    - **Constraints**: Must own the subaccount and have an active position in the selected market. Must match `msg.sender` or be authorized to act on behalf of the user.
    - **Impact**: Specifies whose position is being closed. Position and collateral changes will apply to this address.
- `args.amount`
    - **Control**: Full control.
    - **Constraints**: Must be greater than zero — the amount of the order in base or quote units depending on `amountIsBase`.
    - **Impact**: The amount of the order to fill (in base or quote units depending on `amountIsBase`).
- `args.price`

    - **Control**: Full control.
    - **Constraints**: For `BUY`, the maximum price the user is willing to pay. For `SELL`, the minimum price the user is willing to accept.
    - **Impact**: Prevents the order from filling at worse prices than the user expects.
- `args.side`
    - **Control**: Full control.
    - **Constraints**: Must be opposite to the existing position side.
    - **Impact**: Determines the direction of the closing order.
- `args.amountIsBase`
    - **Control**: Full control.
    - **Constraints**: Boolean. If true, the amount is in base asset — if false, in quote asset. Only `amountIsBase` is allowed.
    - **Impact**: Affects how the order amount is interpreted and matched.
- `args.fillOrderType`
    - **Control**: Full control.
    - **Constraints**: Must be a valid enum value (`FILL_OR_KILL`, `IMMEDIATE_OR_CANCEL`). For `FILL_OR_KILL`, fills fully or reverts — or for `IMMEDIATE_OR_CANCEL`, partial fill is allowed.
    - **Impact**: Affects how the order is executed.
- `args.subaccount`
    - **Control**: Full control.
    - **Constraints**: Must be owned by the account. Must have an open position in this market.
    - **Impact**: Identifies which subaccount's position will be closed.
- `args.leverage`
    - **Control**: Full control.
    - **Constraints**: the margin per unit of quote amount should not less than the `initMarginReq`.
    - **Impact**: Leverage to apply if position is reopened.
- `args.settlement`
    - **Control**: Full control.
    - **Constraints**: Must be a valid settlement type (`INSTANT`, `ACCOUNT`). For settlement `INSTANT`, immediately returns, or for `ACCOUNT`, keeps funds inside the protocol.
    - **Impact**: Defines where the resulting funds go after the position is closed.

Branches and code coverage

Intended branches

- Closes entire position (full match).
  - ☑ Test coverage
- Closes partial position (partially filled).
  - ☑ Test coverage
- `side` to close is correct.
  - ☑ Test coverage
- Filled price is only below or equal to the provided `price` for `BUY`.
  - ☐ Test coverage
- Filled price fills only above or equal to the provided `price` for `SELL`.
  - ☐ Test coverage
- Fill order type `FILL_OR_KILL` fully fills.
  - ☐ Test coverage
- Fill order type `IMMEDIATE_OR_CANCEL` partially fills.
  - ☑ Test coverage
- No matching liquidity and `IMMEDIATE_OR_CANCEL`.
  - ☐ Test coverage
- `amountIsBase == true`.
  - ☑ Test coverage
- Realized PNL is correctly computed on close.
  - ☐ Test coverage
- Partially closed position updates parameters of the position.
  - ☐ Test coverage
- Fully closed position fully resets.
  - ☐ Test coverage
- Settlement type is `INSTANT`.
  - ☑ Test coverage
- Settlement type is `ACCOUNT`.
  - ☑ Test coverage
- Margin is reduced proportionally on partial close.
  - ☐ Test coverage
- Margin is fully released if position is closed.
  - ☐ Test coverage
- Closes across multiple prices.

- ☐ Test coverage
- Protocol reserves are updated.
  - ☐ Test coverage
- Closes a position close to liquidation.
  - ☐ Test coverage

Negative behavior

- Closes with an `amount` greater than the position amount.
  - ☑ Negative test
- Closes a position that does not exist.
  - ☐ Negative test
- Incorrect `side`.
  - ☑ Negative test
- Fill order type `FILL_OR_KILL` partly fills.
  - ☐ Negative test
- There is no matching liquidity in the book and `FILL_OR_KILL`.
  - ☐ Negative test
- `amount == 0`.
  - ☐ Negative test
- Invalid asset.
  - ☐ Negative test
- `price == 0`.
  - ☐ Negative test
- Subaccount does not belong to `account`.
  - ☐ Negative test
- Position is already liquidated.
  - ☐ Negative test
- Order book contains only the user's own opposite order.
  - ☐ Negative test
- Very large amount.
  - ☐ Negative test
- `amountIsBase == false`.
  - ☑ Negative test

Function: `postFillOrder(byte[32] asset, address account, PostFillOrderArgs args)`

This function attempts to open or adjust a position by executing an order. It matches the order against existing orders in the order book using the specified parameters. The user can specify how much to buy or sell, at what price limit, and whether to allow partial fills.

If the order is filled, the user's position is updated: either opened, increased, or reduced (if opposite direction). The function supports both `FILL_OR_KILL` (fills fully or reverts) and `IMMEDIATE_OR_CANCEL` (partially fills).

Inputs

- `asset`
    - **Control**: Full control.
    - **Constraints**: Market should exist.
    - **Impact**: Determines which `asset` market the fill operation applies to.
- `account`
    - **Control**: Full control.
    - **Constraints**: The caller should be the account itself or an approved operator.
    - **Impact**: Position, margin, and balance updates are applied to this account.
- `args.amount`
    - **Control**: Full control.
    - **Constraints**: Must be greater than zero.
    - **Impact**: Sets how much the user wants to buy or sell.
- `args.price`
    - **Control**: Full control.
    - **Constraints**: Must be greater than zero, `price % tickSize == 0`.
    - **Impact**: Defines the worst price the user is willing to accept for the trade.
- `args.amountIsBase`
    - **Control**: Full control.
    - **Constraints**: Boolean — affects interpretation of `args.amount`.
    - **Impact**: Defines whether `amount` is in base asset units or quote.
- `args.fillOrderType`
    - **Control**: Full control.
    - **Constraints**: Must be valid enum value.
    - **Impact**: Sets how the order should be filled, either completely or partially.
- `args.subaccount`
    - **Control**: Full control.

    - **Constraints**: N/A.
    - **Impact**: Determines which subaccount to apply the trade to.
- `args.leverage`
    - **Control**: Full control.
    - **Constraints**: (1e18 * 1e18) / leverage should be less than `settings.initMarginReq`.
    - **Impact**: Controls how much borrowed funds are used — affects margin and liquidation risk.
- `args.settlement`
    - **Control**: Full control.
    - **Constraints**: Must be a valid settlement type (`INSTANT`, `ACCOUNT`). For settlement `INSTANT`, immediately returns, or for `ACCOUNT`, keeps funds inside protocol.
    - **Impact**: Defines where the resulting funds go after the position is closed.

Branches and code coverage

Intended branches

- Opens a new long position using `BUY`.
    - [x] Test coverage
- Opens a new short position using `SELL`.
    - [x] Test coverage
- Increases an existing long.
    - [x] Test coverage
- Increases an existing short.
    - [ ] Test coverage
- Decreases a position by filling with the opposite side (partial close).
    - [x] Test coverage
- Fully closes a position with an opposite-side fill.
    - [ ] Test coverage
- Opens a new position after closing the previous one.
    - [ ] Test coverage
- Executes fill against multiple matching maker orders.
    - [ ] Test coverage
- `IMMEDIATE_OR_CANCEL` partially fills.
    - [ ] Test coverage

- Position updates correctly on increase.
  - ☑ Test coverage
- Position decreases correctly.
  - ☑ Test coverage
- Position `amount` and `openNotional` update correctly on open.
  - ☑ Test coverage
- Realized PNL is correctly applied on partial close.
  - ☐ Test coverage
- Full close resets the position to zero and cleans storage.
  - ☐ Test coverage
- Margin is correctly calculated.
  - ☑ Test coverage
- Funding is settled before fill occurs.
  - ☐ Test coverage
- Reverses open with exact 2× amount.
  - ☐ Test coverage
- Opens new position with opposite side.
  - ☐ Test coverage
- Reverses open from long to short with `amountIsBase == false`.
  - ☐ Test coverage
- Realized PNL is correctly calculated for the closed part of the position.
  - ☑ Test coverage
- Margin is released correctly from the closed position before applying to the new one.
  - ☑ Test coverage
- Reverses open. Price and `openNotional` are recalculated correctly for the new position.
  - ☐ Test coverage
- Fees are charged properly.
  - ☑ Test coverage

Negative behavior

- Price deviates too much from the current fair market value.
  - ☐ Negative test
- `price == 0`.
  - ☐ Negative test

- `asset` is not supported.
  - ☐ Negative test
- `price % tickSize != 0`.
  - ☐ Negative test
- `FILL_OR_KILL` is partly filled.
  - ☐ Negative test
- `FILL_OR_KILL` is not filled at all — no orders.
  - ☐ Negative test
- Reverts if reverse open causes undercollateralization.
  - ☐ Negative test
- For `BUY`, order book contains only higher prices.
  - ☐ Negative test
- For `SELL`, order book contains only smaller prices.
  - ☐ Negative test
- User does not have enough funds to post margin.
  - ☐ Negative test
- Order book is empty.
  - ☐ Negative test

Function: `postLimitOrder(byte[32] asset, address account, PostLimitOrderArgs args)`

This function allows a user to create a new limit order to either buy (bid) or sell (ask). Unlike a market fill order, the limit order is placed into the order book and remains there until it is matched and filled, canceled by the owner, or expires or becomes unfillable.

Inputs

- `asset`
  - **Control**: Full control.
  - **Constraints**: Market should exist.
  - **Impact**: Determines which `asset` market the new limit order applies to.
- `account`
  - **Control**: Full control.
  - **Constraints**: The caller should be the account itself or an approved operator.
  - **Impact**: Position, margin, and balance updates are applied to this account.

Collateral will be provided by the account.

- `args.amountInBase`
    - **Control**: Full control.
    - **Constraints**: Must be greater than zero and more than `minLimitOrderAmountInBase`.
    - **Impact**: The order amount.
- `args.price`
    - **Control**: Full control.
    - **Constraints**: Must be greater than zero — `price % tickSize == 0`.
    - **Impact**: Defines the worst price the user is willing to accept for the trade.
- `args.cancelTimestamp`
    - **Control**: Full control.
    - **Constraints**: Should be zero or more than the `block.timestamp`.
    - **Impact**: The expire time of the order.
- `args.side`
    - **Control**: Full control.
    - **Constraints**: `BUY` and `SELL`.
    - **Impact**: The buy (bid) or sell (ask) order.
- `args.limitOrderType`
    - **Control**: Full control.
    - **Constraints**: `GOOD_TILL_CANCELLED` and `POST_ONLY`.
    - **Impact**: If `POST_ONLY` and `BUY`, the current `BestAsk` should be more than `args.price`. If `POST_ONLY` and `SELL`, the current `BestBid` should be less than `args.price`.
- `args.subaccount`
    - **Control**: Full control.
    - **Constraints**: N/A.
    - **Impact**: Determines which subaccount to apply the trade to.
- `args.leverage`
    - **Control**: Full control.
    - **Constraints**: (1e18 * 1e18) / leverage should be less than `settings.initMarginReq`.
    - **Impact**: Controls how much borrowed funds are used — affects margin and liquidation risk.
- `args.reduceOnly`
    - **Control**: Full control.
    - **Constraints**: Only for the existing position.
    - **Impact**: Order to reduce the current account position.

- `args.settlement`
    - **Control**: Full control.
    - **Constraints**: Must be a valid settlement type (`INSTANT`, `ACCOUNT`). For settlement `INSTANT`, directly transfers from the account, or for `ACCOUNT`, gets funds from the protocol balance of the account.
    - **Impact**: Defines how the collateral funds will be provided.

Branches and code coverage

Intended branches

- Successfully posts a valid `BUY` limit order.
    - ☑ Test coverage
- Successfully posts a valid `SELL` limit order.
    - ☑ Test coverage
- Reserves the correct collateral amount based on amount and leverage.
    - ☐ Test coverage
- The order is correctly added to the order book.
    - ☑ Test coverage
- For large `amountInBase` and low leverage, the collateral amount is correct.
    - ☐ Test coverage
- For large `amountInBase` and high leverage, the collateral amount is correct.
    - ☐ Test coverage
- A user posts many orders, and the order limit is enforced.
    - ☐ Test coverage
- The order with extreme price.
    - ☐ Test coverage

Negative behavior

- `amountInBase == 0`.
    - ☐ Negative test
- `price == 0`.
    - ☐ Negative test
- Reverts if `INSTANT` and margin exceed user's available balance.
    - ☐ Negative test
- Reverts if `ACCOUNT` and margin exceed user's available tokens for transferring.

- ☐ Negative test

Function: `withdraw(address account, uint256 amount)`

This function decreases the `collateral[account]` balance by `amount`. Reverts if the current `collateral[account]` is less than `amount`. It transfers the `amount` of the USDC tokens to the provided `account` from this contract and emits a `Withdrawal` event.

Inputs

- `account`
  - **Control**: Full control.
  - **Constraints**: The caller should be the account itself or an approved operator of this account.
  - **Impact**: The withdrawal will be made from the account balance.
- `amount`
  - **Control**: Full control.
  - **Constraints**: The account should own enough USDC tokens to transfer.
  - **Impact**: The amount of USDC tokens to be deposited.

Branches and code coverage

Intended branches

- A `Withdrawal` event has been emitted.
  - ☐ Test coverage
- The `collateral` balance of the `account` has been decreased by the `amount`.
  - ☐ Test coverage
- The USDC balance of this contract has been decreased by the `amount`.
  - ☐ Test coverage
- The USDC balance of the account has been increased by the `amount`.
  - ☐ Test coverage

Negative behavior

- The caller is not an account or approved operator of this account.
  - ☐ Negative test
- The amount is greater than the `collateral` balance.
  - ☐ Negative test

Function call analysis

- `MarginAccountLib.withdraw(this._getClearingHouse().collateralManager, account, amount) -> self.debitAccount(account, amount)`
  - **What is controllable?** `account` and `amount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** It reverts if `amount` is greater than the balance.
- `MarginAccountLib.withdraw(this._getClearingHouse().collateralManager, account, amount) -> USDC.safeTransfer(account, amount)`
  - **What is controllable?** `account` and `amount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** It reverts when the contract does not have enough USDC tokens.

Function: `_executeFillOrder(ClearingHouse self, byte[32] asset, address account, PostFillOrderArgs args)`

The internal `_executeFillOrder` function handles the execution of filling orders. It performs the following:

- Funding-payment settlement by calling `settleFundingPayment`
- Execution of the `market[asset].postFillOrder(...)` function, which handles order execution and returns a `positionResult`
- Execution of `collateralManager.settleTakerFill`
- Liquidation safety check
- Processing of `takerFee`

Branches and code coverage

Intended branches

- `fundingPaymentResult.marginDelta` is applied correctly to `positionResult.marginDelta`.
  - ☐ Test coverage
- `fundingPaymentResult.debt` is applied correctly to `positionResult.debt`.
  - ☐ Test coverage
- The position is increased or decreased as expected.
  - ☐ Test coverage

- If `positionResult.fullClose == true`, the position is removed from user positions.
  - ☐ Test coverage
- Margin and debt balances are updated properly in `settleTakerFill`.
  - ☐ Test coverage

5.5. Module: Position.sol

Function: `positionTakeover(Position storage liquidator, Position memory liquidatee, uint256 price, uint256 leverage)`

If the liquidator already holds a position in the same direction as the liquidatee position or has no position, the `_takeoverOpen` function is executed to increase the liquidator position. If the liquidator holds a position in the opposite direction so the current position is partially or fully closed, a new one may be reopened in the new direction using `_takeoverClose`.

Branches and code coverage

Intended branches

- Liquidator has no position.
  - ☐ Test coverage
- Liquidator and liquidatee are both long.
  - ☐ Test coverage
- Liquidator and liquidatee are both short.
  - ☐ Test coverage
- Liquidator is long and liquidatee is short.
  - ☐ Test coverage
- Liquidator is short and liquidatee is long.
  - ☐ Test coverage
- `oiDelta.long` or `oiDelta.short` are calculated correctly.
  - ☐ Test coverage
- Liquidator's position has correct `price`, `margin`, `openNotional`, and `amount`.
  - ☐ Test coverage
- `collateralDelta` is calculated properly.
  - ☐ Test coverage
- Leverage is used correctly to determine a new margin.
  - ☐ Test coverage

- Liquidatee PNL is correct for a long position closed at a lower price.
    - ☐ Test coverage
- Liquidatee PNL is correct for a short position closed at a higher price.
    - ☐ Test coverage

Function: `settleFundingPayment(Position storage self, int256 cumulativeFunding)`

This internal function calculates the funding payment owed or credited to the user since their last funding update, adjusts their margin accordingly, and updates the funding snapshot in the position to the current global funding index. Function execution will be skipped if the `position.lastCumulativeFunding == cumulativeFunding`. The `fundingPayment` is calculated as the position's amount multiplied by the difference between the current cumulative premium funding rate and the position's last cumulative funding snapshot: `fundingPayment = position.amount × (cumulativePremiumFunding - position.lastCumulativeFunding)`

- If `position.isLong`, `position.amount` is considered as positive.
- If `!position.isLong`, `position.amount` is considered as negative.
- If `fundingPayment` is positive, it will be subtracted from the margin (meaning the user pays funding).
- If `fundingPayment` is negative, it will be added to the margin (meaning the user receives funding).

The `position.lastCumulativeFunding` will be updated to the current `cumulativeFunding`.

Branches and code coverage

Intended branches

- If `fundingPayment` is positive and the user is long, margin is reduced.
    - ☐ Test coverage
- If `fundingPayment` is negative and the user is short, margin is reduced.
    - ☐ Test coverage
- If `fundingPayment` is positive and the user is short, margin is increased.
    - ☐ Test coverage
- If `fundingPayment` is negative and the user is long, margin is increased.
    - ☐ Test coverage
- `lastCumulativeFunding == cumulativeFunding`.
    - ☐ Test coverage
- For a small amount or small `marginDelta`, the result is calculated correctly.

- ☐ Test coverage

6. Assessment Results

During our assessment on the scoped GTE – Perp contracts, we discovered 18 findings. One critical issue was found. Two were of high impact, four were of medium impact, five were of low impact, and the remaining findings were informational in nature.

The existing testing suite used during the assessment is limited in its coverage and does not fully test all components of the smart contracts. This limitation prevented comprehensive testing and identification of potential issues, particularly in terms of negative testing scenarios. Some of the findings reported in this assessment could have been detected if the testing suite had been more comprehensive. To ensure the robustness of the smart contracts, we recommend that Liquid Labs, Inc. expands the testing suite to cover all functionality, including negative testing, as outlined in the threat model. See discussion point 4.1. ↗ for more detail on the effects of a comprehensive test suite.

To ensure the protocol is as secure and reliable as possible, we strongly suggest conducting a comprehensive reaudit upon completion of development and before deployment of the protocol's contracts. A reaudit is valuable in identifying any potential issues or vulnerabilities and allowing for any necessary changes or fixes to be made before deployment.

Key areas for enhancement

- The project currently lacks any form of documentation, including NatSpec annotations and explanatory comments. Introducing clear documentation would significantly improve code readability and facilitate understanding of the system's logic and functionality.
- The current test coverage includes several tests, including some multistep scenarios. However, due to the inherent complexity of the protocol, the system's state can vary widely under different conditions. It is crucial to expand the test suite with more complex, multistep tests that simulate diverse system states and sequences of actions. This includes testing edge cases that can arise in a system with margin requirements, leverage constraints, funding payments, liquidations, and cross-margin. We recommend adding tests for all functions and their branches, including boundary conditions and unexpected user behaviors. It's also important to validate that all state variables are updated consistently across multiple actions to ensure correctness and stability across the wide range of interactions that can occur in a perpetual protocol.

Recommended next steps

- Deploy to MegaETH testnet for several weeks to validate intended behavior under real conditions.
- Consider improving code documentation to support long-term maintenance.
- Expand test coverage.

While the foundation is promising, implementing these recommendations would provide greater assurance for mainnet deployment.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.