DayHub Platform Security Review

Duration: August 1, 2025 - August 23, 2025

September 15, 2025

Conducted by **KeySecurity**
Georgi Krastenov, Lead Security Researcher

Table of Contents

1 About KeySecurity

KeySecurity is an innovative Web3 security company that hires top talented security researchers for your project. We have conducted over 40 security reviews for different projects which hold over $300,000,000 in TVL. For security audit inquiries, you can reach out to us on Twitter/X or Telegram @gkrastenov, or check our previous work here.

2 About Dayhub

Dayhub is the first ecosystem that revolutionizes funded trading using Blockchain Transparency and Fairness Dayhub leverages cutting-edge blockchain technology to ensure fair and transparent trading opportunities. Participate in custom challenges, access substantial trading capital, and experience a new era of decentralized funded trading.

3 Disclaimer

Audits are a time, resource, and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

4 Risk classification

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

4.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

4.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

4.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

5 Executive summary

Overview

Project Name	Dayhub
Repository	https://github.com/Dayhub-io/dayhub-contracts
Commit hash	b2eb70657574c48e8849142405a443cf8e0bbf47
Review Commit hash	ac02e616b121b1ee60cea8564c6912e3acf8ab3b
Documentation	README file
Methods	Manual review

Scope

./src

Timeline

August 1, 2025	Audit kick-off
August 23, 2025	Mitigation review
September 15, 2025	Final report

Issues Found

Severity	**Count**
High	16
Medium	5
Low	4
Information	16
Total	**41**

6 Findings

6.1 High

6.1.1 getDayUniswapPrice use near-spot price

Severity: *High*

Context: UniswapFacet.sol#L110

Description: When fetching the price of the `DAY` token via Uniswap V3 trough `getDayUniswapPrice` function, the current implementation estimates the amount out for 1 DAY using a `near-spot price` rather than a proper time-weighted average price (TWAP).

This is due to the `OracleLibrary.consult()` function being called with a time parameter of **5 seconds**, which is too short to produce a meaningful average and effectively returns a spot-like value.

```
    function estimateAmountOut(
        address factory,
        address tokenIn,
        address tokenOut,
        uint128 amountIn,
        uint24 fee
    ) public view returns (uint256 amountOut) {
        address pool = IUniswapV3Factory(factory).getPool(tokenIn, tokenOut, fee);

        if (pool == address(0)) {
            revert UniswapFacet__InvalidPool();
        }

        //@audit use SPOT price
        (int24 tick, ) = OracleLibrary.consult(pool, 5);

        amountOut = OracleLibrary.getQuoteAtTick(tick, amountIn, tokenIn, tokenOut);
    }
```

Using TWAP instead of the spot price helps protect against short-term price manipulation and provides a more reliable, averaged market value.

Recommendation: Instead of using a hardcoded 5 seconds, use `uniswapCore.secondsAgo`.

Resolution and Client comment: Resolved. It is used `uniswapCore.secondsAgo` now.

6.1.2 ProfitPercentage bonuses does not work

Severity: *High*

Context: VaultFacet.sol#L310

Description: The withdraw profit workflow/system allows receiving a bonus after certain criteria are met. A user can receive either a fixed amount as a bonus or a percentage of the profit they wish to withdraw.

```
bonusAmount = uint256(uint64(profitFromPhase)* b.value)/ LibAppStorage.PRECISION;
```

The `_findBonusForFunding` function determines which bonus a user should receive. It checks the funding level and if it matches a specific bonus, the `PayoutBonus` is returned. Unfortunately, inside the for loop that iterates over all bonuses, the code only checks for the `FixedAmount` type, which makes receiving a `ProfitPercentage` bonus impossible.

```
    function _findBonusForFunding(
        PayoutBonus[] memory allBonuses,
        uint8 fundingLevel
    ) internal view returns (PayoutBonus memory, uint256) {
        for (uint256 i = 0; i < allBonuses.length; i++) {
            //@audit-issue is it possible only receiving of FixedAmount bonus
            if (allBonuses[i].bonusType == BonusType.FixedAmount && allBonuses[i].
                fundingLevel == fundingLevel) {
                return (allBonuses[i], i);
            }
        }
        return (
            PayoutBonus({
                bonusType: BonusType.FixedAmount,
                value: 0,
                fundingLevel: 0,
                availableAfterPayout: 0,
                fromPhase: 0
            }),
            0
        );
    }
```

Recommendation: Remove the check for `bonusType == BonusType.FixedAmount` in the for loop, as only the funding level should be checked.

Resolution and Client comment: Resolved. The `bonusType == BonusType.FixedAmount` check is removed.

6.1.3 Insufficient dayhubVault funds block referral claims

Severity: *High*

Context: RefferalFacet.sol#L104-L105

Description: When the `updateReferralData` function is called to update the Merkle root and transfer funds for the new reward distribution to new referrals, the funds are moved from the `dayhubVault` to the `marketingVault`. After that, users can call the `claimReferral` function to receive their tokens from the `marketingVault`.

Currently, the `dayhubVault` does not receive funds anywhere in the project, not even when a user joins a challenge. This means the `dayhubVault` will not have the required tokens (both DAY and USDC) for the referral payouts.

```
Vault(dayhubVault).withdrawTo(usdc, marketingVault, _totalUsdc);
Vault(dayhubVault).withdrawTo(day, marketingVault, _totalDay);
```

Recommendation: Send a percentage of the paid fee from users when they join a challenge to the `dayhubVault` to cover all future reward distributions to referral partners.

Resolution and Client comment: Resolved. When a user joins a challenge, he pays a percentage of his fee to the dayhubVault.

6.1.4 Withdrawals can be scheduled without enough prize pool funds

Severity: *High*

Context: VaultValidator.sol#L23

Description: When a new profit withdrawal is made, the system checks if the `prizePool` has enough funds to cover it.

The prize pool is calculated as: `dayPrice * balanceOf(baseToken for challengeToVault)/PRECISION`

```
            uint64(
                (dayPrice * // day price in USDC
                    uint64(
                        IERC20(uniswapPool.baseToken).balanceOf(
                            LibVaultStorage.getLayout().challengeToVault[
                                _challengeId]
                        ) // 6 decimals price * 6 decimal tokens /  1e6 precision
                    )) / LibAppStorage.PRECISION
            ), // rewardPool
```

After calculation, the amount is sent to `VaultValidator.validateProfitWithdrawal`, which internally calls `validatePrizePool`.

Currently, the `validatePrizePool` function **only checks if `rewardPool > 0`**, not whether the prize pool is large enough to cover the requested withdrawal:

```
    /**
     * @notice Validates that the reward pool has sufficient funds
     * @param rewardPool The available reward pool amount
     * @custom:reverts VaultValidator__PrizePoolEmpty if reward pool is zero
     */
    function validatePrizePool(uint256 rewardPool) internal {
        require(rewardPool > 0, VaultValidator__PrizePoolEmpty());
    }
```

This allows a situation where a user can request a large withdrawal, and while it passes validation (since rewardPool > 0), the prize pool may not have enough funds to actually cover the profit when the withdrawal is executed.

Recommendation: Update the validation logic to compare `rewardPool` against the requested withdrawal amount, ensuring sufficient funds before scheduling the withdrawal.

Resolution and Client comment: Resolved. Added the following check: `prizePool > profit + bonus`.

6.1.5 Bulk challenge join bypasses max funding limit check

Severity: *High*

Context: ChallengeValidator.sol#L66-L70

Description: When a user attempts to join a challenge, several validations are performed. One of them ensures that the total funding across active attempts does not exceed the maximum allowed, using the `validateMaxFundingAtOnce` function:

```
    function validateChallengeJoining(bytes32 _challengeId, uint64 _fundingPool,
        uint8 _attemptCount) internal {
                // ....

        validateMaxFundingAtOnce(msg.sender, _challengeId, _fundingPool);
    }
```

The system currently supports both **single join** and **bulk join** for challenges.

In the case of a bulk join, the `_fundingPool` value is sent **only once** to `validateMaxFundingAtOnce`, **not multiplied by the number of joins**.

This allows users to easily exceed the maximum allowed funding.

Example:

- Maximum allowed funding: **10,000K**
- User has **4 active attempts** at **2,000K** each → total **8,000K**
- User then uses **bulk join** to join the same challenge **4 more times** at **2,000K** each
- The validation will check: `existing 8,000K + new 2,000K = 10,000K` passes
- **Actual total after bulk join:** 16,000K exceeds the limit

`validateMaxFundingAtOnce` is not accounting for total funding across all bulk join attempts, allowing users to bypass funding limits.

Recommendation: For bulk joins use `_fundingPool * _attemptCount` in the `validateMaxFundingAtOnce` function.

Resolution and Client comment: Resolved. The new funding is calculated by `_fundingPool * _attemptCount`.

6.1.6 Limit position opens at current price instead of user-specified entry price

Severity: *High*

Context: KeeperFacet.sol#L306

Description: When a user registers a position, they are allowed to specify an `entryPrice`.

- If `entryPrice == 0`, the position will be opened as a market position.
- If `entryPrice != 0`, it will be opened as a limit position.

```
        if (position.props.entryPrice == 0) {
            PositionProcessor.openMarketPosition(
                currentPrice,
                position.challengeId,
                positionId,
                position,
```

```
                spreadConfig
            );
        } else {
            PositionProcessor.openLimitPosition(currentPrice, position.challengeId,
                positionId, position, spreadConfig);
        }
```

Both `openMarketPosition` and `openLimitPosition` fetch the*current asset price at the time of opening.

For limit positions, this creates a mismatch between the user-specified `entryPrice` (at registration) and the actual price used when the position is opened.

Even if the user initially sets the `entryPrice` equal to the current price, if the position is opened later (e.g., by a Keeper) and the asset price has changed due to volatility, the position will open at the new current price, not the intended user-defined entry price.

Users may not get the expected entry price for limit positions. Can cause slippage or unintended trades, especially in volatile markets.

Recommendation: When opening a limit position, enforce the original user-specified entryPrice rather than using the current price directly.

Resolution and Client comment: Partially Resolved. The limit order is correctly registered, but the current price is still used when the position is opened instead of the user-specified price.

6.1.7 Wrong bid and ask price is used in registration of TP and SL

Severity: *High*

Context: PositionProcessor.sol#L148

Description: For long positions, the bid price (lower than oracle) is incorrectly used for take profit and stop loss triggers, causing them to execute earlier or at unfavorable prices.

When a user sets a take profit or stop loss, a small spread is applied above and below the oracle price. The code then checks if the position is long. However, in the case of a long position, the bid price (normally lower than the oracle price) is selected instead of the ask price.

Example:

- Oracle price = $2000
- Spread = 20 bps (0.2%)
- `getBidAsk` returns:
  - Bid = $1998 (sell price)
  - Ask = $2002 (buy price)

If the position is long, the code incorrectly uses the bid ($1998), even though the position was opened at the ask price(current oracle price + spread). This misalignment leads to take profit/stop loss triggering at the wrong price.

```
  function openLimitPosition(
        uint256 oraclePrice,
        bytes32 challengeId,
        bytes32 positionId,
        Position memory position,
        SpreadConfig memory spread
    ) internal {
        (uint256 bid, uint256 ask) = OrderPricing.getBidAsk(oraclePrice, spread.
            spreadBps);
        // when posiiton is long ASK price is get for limit order
        uint256 price = position.props.size == 1 ? ask : bid;

        openPosition(challengeId, positionId, price, position);
    }

function registerTP_SL(Position memory pos, bytes32 positionId) internal {
                // ...
        bool isLong = pos.props.size > 0;
        // ...

        if (pos.props.takeProfit != 0) {
            (uint256 bidAtTP, uint256 askAtTP) = OrderPricing.getBidAsk(pos.props.
                takeProfit, spread.spreadBps);
            takeProfitTriggerPrice = isLong ? bidAtTP : askAtTP;
        }

        if (pos.props.stopLoss != 0) {
            (uint256 bidAtSL, uint256 askAtSL) = OrderPricing.getBidAsk(pos.props.
                stopLoss, spread.spreadBps);
            stopLossTriggerPrice = isLong ? bidAtSL : askAtSL;
        }
        // ...
    }
```

Recommendation: Use the ask price for long positions and the bid price for short positions when evaluating take profit and stop loss triggers.

Resolution and Client comment: Resolved. The spread is removed from TP and SL.

6.1.8 Bonuses are not included when the reward pool is checked for sufficient funds

Severity: *High*

Context: VaultFacet.sol#L155

Description: Withdrawals can become stuck if the reward pool has enough funds to cover profit but not the additional bonus, leading to frozen user withdrawals.

When users withdraw profit, they may receive a fixed or percentage-based bonus during certain phase. This bonus is paid from the reward pool. Before processing a withdrawal, the system checks whether the reward pool has enough funds to cover the requested profit amount. However, the check does not account for the bonus.

This creates an edge case where the pool has sufficient funds for profit but not enough for the bonus. As a result, the withdrawal may fail, effectively freezing the user's funds.

```
        // 2) Validate
        VaultValidator.validateProfitWithdrawal(
            _profit,
            // validatePrizePool(rewardPool);
            uint64(
                (dayPrice *
                    uint64(
                        IERC20(uniswapPool.baseToken).balanceOf(
                            LibVaultStorage.getLayout().challengeToVault[
                                _challengeId]
                        )
                    )) / LibAppStorage.PRECISION
            ),
            challenge.phases.length,
            LibVaultStorage.getLayout().userChallengeToWithdrawals[msg.sender][
                _challengeId][_attemptId].lastPayout,
            challenge.challengeConfig.payoutConfig.payoutFrequency,
            challenge.challengeConfig.payoutConfig.firstWithdrawalAllowedAfter,
            LibPositionStorage.getUserChallengePositionsToAttemptDetails(msg.sender,
                 _challengeId, _attemptId),
            ucd
        );
```

Recommendation: Update the reward pool sufficiency check to include both the profit and the bonus before approving a withdrawal.

Resolution and Client comment: Resolved.

6.1.9 A position can be closed at any time since openedAt is never set

Severity: *High*

Context: KeeperFacet.sol#L543

Description: Any user can close another user's position because `openedAt` is never set, allowing malicious actors to manipulate the victim's PnL.

When a position is created or opened, the `openedAt` field is never set. As a result, `openedAt` always remains `0`, making the check `positionOpenDuration >= maxTimePositionCanBeOpened` always always to be true.

```
            uint32 positionOpenDuration = uint32(block.timestamp) - _p.props.
                openedAt;
            require(
                positionOpenDuration >= maxTimePositionCanBeOpened,
                KeeperFacet__PositionTimeNotViolatedMaxTimeOpened()
            );
```

This introduces the risk of malicious users harming others by force-closing positions and manipulating their PnL.

Recommendation: Set `openedAt = block.timestamp` when a position is opened.

Resolution and Client comment: Resolved.

6.1.10 Anyone can close an already closed position of another user

Severity: *High*

Context: KeeperFacet.sol#L558

Description: Closed positions can be closed again, potentially preventing users from closing their actual open positions and exposing them to harmful PnL outcomes.

The function `closePositionsWhenMaxTimePositionOpenReached` iterates through all provided positions without checking whether a position is already closed. The only requirement is that `openedPositions > 1`.

As a result, if a user has 1 CLOSED position and 1 OPEN position, a malicious actor can call `closePositionsWhenMaxTimePositionOpenReached` and attempt to close the already closed position again (since in most cases the max time will have passed). This can harm the user, for example by locking their open position to be able to be closed and adding negative PnL.

```
    function closePositionsWhenMaxTimePositionOpenReached(bytes32[] memory
        _positions) public {
        for (uint256 index = 0; index < _positions.length; index++) {
            // ....
            PositionProcessor.closeMarketPosition(_positions[index], _price, _p,
                _spreadConfig);
                        // ....
        }
    }
```

Recommendation: Restrict `closePositionsWhenMaxTimePositionOpenReached` to only close positions that are in the OPENED state.

Resolution and Client comment: Resolved. Only the keeper can call `closePositionsWhenMaxTimePositionOpenRea` and it is checked in the position has a OPEN status.

6.1.11 Open position cancellation allows users to bypass losses

Severity: *High*

Context: PositionFacet.sol#L182

Description: Allowing users to cancel open positions enables them to avoid losses (e.g., bypassing stop-loss execution), breaking trading fairness.

When a user opens a position, they may either realize a profit or incur a loss once the position is closed. The system currently allows cancellation of both PENDING and OPEN positions.Canceling a PENDING position makes sense, since it has not yet been executed and the user may want to adjust parameters. Canceling an OPEN position, however, undermines the integrity of trading logic.

For example, if a user opens a long position and the price decreases toward their stop-loss, they can simply cancel the position instead of realizing the loss.

```
    function closePosition(bytes32 _positionId) public {
        Position storage _p = LibPositionStorage.getLayout().positionsDetails[
            _positionId];
```

```
        LibPositionValidator.validateOrder(_p.user, _p.status);

        _p.status = PositionStatus.PENDING_CLOSED;

        emit Events.CloseOrder(_p.challengeId, _positionId, _p.attemptId, msg.sender
            );
    }
```

Recommendation: Restrict cancellation to PENDING positions only; prevent users from canceling OPEN positions.

Resolution and Client comment: Resolved. It is checked if position has OPEN status.

6.1.12 Excluding PENDING_CLOSED positions undervalue totalPositionsValue

Severity: *High*

Context: PositionFacet.sol#L52

Description: `PENDING_CLOSED` positions are excluded from `totalPositionsValue`, causing underestimation of user exposure.

The function `getUserChallengePositionsToAttemptValue` calculates the `totalPositionsValue` by looping through all positions and summing the values of OPEN and PENDING positions.

However, `PENDING_CLOSED` positions are not included. These positions were previously open and marked for closure by the user, but they remain active until the keeper executes the close. By excluding them, the total exposure may be undervalue, leading to incorrect validations and risk checks.

Recommendation: Include PENDING_CLOSED positions in `getUserChallengePositionsToAttemptValue` until they are fully closed.

Resolution and Client comment: Resolved. It is added `p.status == PositionStatus.PENDING_CLOSED` check.

6.1.13 Mismatch in openned position tracking lets users bypass maxSimultaneousPositions

Severity: *High*

Context: PositionFacet.sol#L182

Description: When maxSimultaneousPositions != type(uint8).max, users are restricted in how many positions they can open per phase. This is enforced by validateMaxPositionPerChallenge:

```
        if (_challenge.challengeConfig.consistencyRules.maxSimultaneousPositions !=
            type(uint8).max) {

            // opened < max open
            validateMaxPositionPerChallenge(
                _ucd.openedPositions,
                _challenge.challengeConfig.consistencyRules.maxSimultaneousPositions
            );
        }
```

Here, `_ucd` represents the UserChallengeDetails for the given challenge attempt:

```
    function _validatePositionRequest(
        bytes32 _challengeId,
        uint8 _attemptId,
        PositionProps memory _positionProps
    ) private {
            //...

        UserChallengeDetails storage userChallengeDetails =
            LibChallengeStorage
            .getLayout()
            .userChallengeDetails[msg.sender][_challengeId][_attemptId];

        // ..
        LibPositionValidator.validatePosition(
            challenge,
            userChallengeDetails, // _ucd
            _positionProps,
            phase,
            equity,
            totalPositionsValue
        );
    }
```

openedPositions is incremented when a position goes OPEN and decremented when a position is CLOSED.

```
        // addPosition()
        LibChallengeStorage
            .getLayout()
            .userChallengeDetails[_position.user][_position.challengeId]
            [_ucd.phase]
            .openedPositions++;

        // updatePositionDetails()
        LibChallengeStorage
            .getLayout()
            .userChallengeDetails[position.user][position.challengeId][phaseId]
            .openedPositions--;
```

However, both operations rely on the `phaseId` and are compared with user details obtained from the `attemptId`. This creates a mismatch in how `openedPositions` is tracked.

For example, if a user is on `attemptId = 4`, `_ucd.openedPositions` will be equal to `0`, even though in `phaseId = 1` the same user may already have 6 open positions, exceeding a configured limit of 5. In this case, `validateMaxPositionPerChallenge` will always pass incorrectly, allowing users to bypass the limit.

Recommendation: Ensure that openedPositions is obtained from the phaseId, not from the attemptId, when the check is made in the `validateMaxPositionPerChallenge` function.

Resolution and Client comment: Resolved. It is used attempt id not the phase.

6.1.14 Excluding PENDING_CLOSED positions causes incorrect equity calculation

Severity: *High*

Context: PositionFacet.sol#L256

Description: Excluding `PENDING_CLOSED` positions in `getEquity` leads to incorrect equity calculation.

The `getEquity` function calculates current equity as sum up `ucd.balance` andunrealized PnL from OPEN positions.

Whenever a position is closed, the balance is updated with the realized PnL. For equity calculation, the function loops through all user positions and simulates closing any OPEN positions to include their unrealized PnL.

However, positions with status `PENDING_CLOSED` are skipped. These positions were open but marked for closure by the user and remain open until the keeper finalizes them. By excluding them, `getEquity` undervalue user equity and produces inaccurate results.

Recommendation: Include PENDING_CLOSED positions in getEquity calculations until they are fully closed by the keeper.

Resolution and Client comment: Resolved.

6.1.15 applyFundingFees may exceed block gas limits as system grows

Severity: *High*

Context: VaultFacet.sol#L155

Description: Applying funding fees may become unsustainable and revert due to block gas limits as the system grows.

The `applyFundingFees` function iterates through all users, all challenges, all attempts, and all positions within each attempt to apply funding fees:

```
if (totalFee > 0) {
    details.balance -= uint64(totalFee);
}
```

While this may work initially, as the project scales with more users, challenges, attempts, and positions, this approach will lead to excessive gas consumption. Since block gas limits (e.g., 14M on Base mainnet) are finite, looping over the entire dataset can cause the transaction to revert, preventing funding fees from being applied at all.

Recommendation: Maintain a dedicated set of open position IDs.

- When a position is opened → add its ID to the set.
- When a position is closed → remove its ID from the set. Then, in `applyFundingFees`, iterate only over this set of open positions to apply fees efficiently.

Resolution and Client comment: Resolved.

6.1.16 writeDailyBalances will revert because of block gas limit

Severity: *High*

Context: KeeperFacet.sol#L164

Description: The `writeDailyBalances` function iterates through all users, their challenges, and every attempt to check whether the current equity has fallen below the funding pool. If it has, the daily balance is updated to match the equity.

```
if (equity < userChallenge.fundingPool) {
  LibChallengeStorage
  .getLayout()
  .userChallengeDetails[_users[i]][_challenges[j]][k].dailyBalance = uint64(equity);
   emit Events.DailyBalanceUpdated(_users[i], _challenges[i], equity);
}
```

Although this approach works at a small scale, as the number of users, challenges, attempts, and positions grows, it becomes prohibitively expensive in gas. With finite block gas limits (e.g., ~14M on Base mainnet), iterating over the entire dataset can cause transactions to revert—blocking funding fee updates entirely.

Recommendation: Allow keepers to specify which user challenges and attempts need dailyBalance updates, instead of iterating through all users.

Resolution and Client comment: Resolved.

6.2 Medium

6.2.1 User can join inactive challenge

Severity: *Medium*

Context: ChallengeValidator.sol#L35

Description: When a user tries to join a challenge, only the end time is validated using validateChallengeActive, which checks if `_challengeEnds == 0 || block.timestamp < _challengeEnds`. However, the challenge's start time is not checked.

```
    /**
     * @notice Validates that the challenge is still active (not expired).
     * @param _challengeEnds The challenge expiration timestamp. 0 means no expiry.
     * @custom:reverts ChallengeValidator__ChallengeInactive if the challenge has
         already ended.
     */
    function validateChallengeActive(uint256 _challengeEnds) internal view {
        bool isActive = (_challengeEnds == 0 || block.timestamp < _challengeEnds);
        if (!isActive) {
            revert ChallengeValidator__ChallengeInactive(block.timestamp,
                _challengeEnds);
        }
    }
```

This means users can potentially join challenges that are still inactive (i.e where block.timestamp < challengeStarts), especially if the challenge is scheduled to start in the future. Users may join challenges before they have officially started, which could lead to unexpected behavior or incorrect tracking. **Recommendation:** Add a check to ensure block.timestamp >= challengeStarts when users join a challenge.

Resolution and Client comment: Acknowledged.

6.2.2 getPositions() may revert due to block gas limit

Severity: *Medium*

Context: LibPositionStorage.sol#L35

Description: The getPositions() function retrieves all position IDs currently known in the system by calling values() on an enumerable set. While this works initially, as the project continues to grow and more positions are added, the size of the returned array will increase. Eventually, the number of positions may become too large to retrieve in a single call due to the block gas limit, causing the function to revert.

```
    function getPositions() internal view returns (bytes32[] memory) {
        return getLayout().positions.values();
    }
```

This means users can potentially join challenges that are still inactive (i.e where block.timestamp < challengeStarts), especially if the challenge is scheduled to start in the future. Users may join challenges before they have officially started, which could lead to unexpected behavior or incorrect tracking.

Recommendation: Refactor getPositions() to support pagination or range-based access (e.g., getPositionsInRange(uint256 start, uint256 count)) to ensure safe and scalable retrieval of positions without hitting gas limits.

Resolution and Client comment: Acknowledged.

6.2.3 Phase success check ignores pending positions

Severity: *Medium*

Context: PositionProcessor.sol#L208

Description: When a position is closed, the system checks whether the phase can be marked as successful. One requirement is that the user must not have any OPENED positions. The function `checkIfPositionClosedForChallenge` iterates through all positions and verifies that none are in the OPEN state. If an open position is found, it returns **`false`**, preventing `checkPhaseSuccess` from being called.

However, the function skips checking pending positions. This creates a scenario where a user can open several pending positions and then close one, bypassing the intended restriction and allowing a phase to be marked as successful prematurely.

```
    function closePositionAndCheckPhase(bytes32 positionId, uint256 closePrice,
        Position memory position) internal {
        closePosition(positionId, closePrice, position);

        if (
            LibPositionStorage.checkIfPositionClosedForChallenge(
                position.user,
                position.challengeId,
                position.attemptId
            ) &&
            LibChallengeStorage.getChallenge(position.challengeId).phases.length - 1
                 >
            LibChallengeStorage
            .getLayout()
            .userChallengeDetails[position.user][position.challengeId][position.
                attemptId].phase
        ) {
            ChallengeProcessor.checkPhaseSuccess(position.user, position.challengeId
                , position.attemptId);
        }
    }
```

Recommendation: Update `checkIfPositionClosedForChallenge` to include checking for pending position.

Resolution and Client comment: Resolved.

6.2.4 Pending positions included in size consistency check enable manipulation

Severity: *Medium*

Context: ChallengeRulesValidator.sol#L28

Description: Pending positions are included in size consistency checks, allowing users to manipulate the ratio and bypass requirements.

One requirement for a phase to succeed is that the user maintains consistency in position size. The function `getPositionSizeConsistency` enforces this by looping through all positions, summing their sizes, and dividing by the number of positions to calculate `avgAbsolute`. It then divede the largest position size with the average size to determine the ratio.

However, the function does not skip pending positions. Since pending positions can later be canceled, users can artificially inflate the average size by opening multiple pending positions, pass the size consistency check, and then cancel them. This undermines the purpose of the consistency requirement.
Recommendation: Exclude pending positions from the calculation in `getPositionSizeConsistency` .

Resolution and Client comment: Resolved.

6.2.5 The total funding amount can be calculated incorrectly

Severity: *Medium*

Context: ChallengeValidator.sol#L51-L55

Description: When a user joins a challenge, both the attempt index and the total active attempts are incremented. When an attempt is marked as failed via executeChallengeRules, the totalActiveAttempts is decremented:

```
            LibChallengeStorage
                .getLayout().userChallengeDetails[user][challengeId][attemptId]
                .status = ChallengeStatus.FAILED;

            LibChallengeStorage
                .getLayout()
                .userChallengeToAttemptId[user][challengeId]
                .totalActiveAttempts--;
```

One requirement for joining a challenge is that the user's total funding + new funding must not exceed the configured maximum:

```
        require(
            totalFunding + _fundingPool <=
                LibChallengeStorage.getChallenge(_challengeId).challengeConfig.
                    maxFundingAtOnce,
            ChallengeValidator__FundingExceedsMaxAtOnce(totalFunding, _fundingPool)
        );
```

The total funding is calculated by looping through all active attempts and summing their funding pools. However, the loop incorrectly uses `totalActiveAttempts` instead of the actual attempt index. This causes the iteration to skip some attempts, leading to underestimation of total funding.

As a result, if a user has failed attempts and rejoins multiple times, the loop may not account for all attempts, allowing them to exceed the intended funding cap.

```
    function validateMaxFundingAtOnce(address _user, bytes32 _challengeId, uint64
        _fundingPool) internal view {
        uint256 totalFunding = 0;

                // is it possible attemptIndex > totalActiveAttempts
        uint256 activeAttempts = LibChallengeStorage
                                .getLayout()
                                .userChallengeToAttemptId[_user][_challengeId].
                                    totalActiveAttempts;

        for (uint8 i = 0; i < activeAttempts; i++) {
            if (
                LibChallengeStorage
                .getLayout()
                .userChallengeDetails[_user][_challengeId][i]
                .status == ChallengeStatus.ACTIVE
            ) {
                totalFunding += LibChallengeStorage
                .getLayout()
                .userChallengeDetails[_user][_challengeId][i].fundingPool;
            }
        }
```

Recommendation: Use attemptIndex when iterating through all attempts, not totalActiveAttempts.

```
uint256 activeAttempts = LibChallengeStorage
                         .getLayout()
                         .userChallengeToAttemptId[_user][_challengeId].
                             attemptIndex;
```

Resolution and Client comment: Resolved. It is used attemptIndex now.

6.2.6 Pending positions should only be cancellable

Severity: *Medium*

Context: PositionFacet.sol#L182

Description: Allowing users to close PENDING positions breaks logic consistency, as these positions are not yet open and should only be cancellable.

The `closePosition` function currently allows users to close both PENDING and OPEN positions. While closing OPEN positions is valid, closing PENDING positions is unnecessary and inconsistent with expected behavior.

A **PENDING** position has not yet been executed by the keeper. Allowing it to be "closed" forces closure of a position that was never opened, which blurs the distinction between canceling and closing. Users may simply want to adjust parameters before execution, which should be handled through cancellation.

Recommendation: Restrict closePosition to OPEN positions only; require PENDING positions to be canceled instead of closed.

Resolution and Client comment: Resolved. It was added check for status == OPEN.

6.3 Low

6.3.1 Creator day fee can be stucked in the contract

Severity: *Low*

Context: FundsProcessor.sol#L95

Description: When USDC and DAY tokens are distributed, a portion is transferred to the creator as a `creatorUSDC` / `creatorDAY` fee.The only requirements for the creator to receive fees are `creatorFee` > 0 and `creator` address to be different than `address(0)` .

In the edge case where `creatorFee` > 0 but the `creator` address is `address(0)`, the fee amount is still subtracted from the total distributable tokens, but*not transferred anywhere. This results in fewer tokens being distributed to recipients and leaves the deducted tokens stuck in the Diamond contract.

```
        if (creatorUsdcFee > 0 && creatorAddress != address(0)) {
            usdcToken.transfer(creatorAddress, creatorUsdcFee);
        }

        //@audit-issue creatorUsdcFee is still substracted if creatorAddress =
            address(0)
            swapAmount = _amount - dayhubVaultUsdcFee - treasuryUsdcFee -
                creatorUsdcFee;
```

Recommendation: Do not subtract creatorUsdcFee from the total amount to reduce swapAmount when creatorAddress == address(0). Also, in the same case with creatorDAYFee, transfer these tokens to the treasury instead of leaving them stuck in the contract.

Resolution and Client comment: Acknowledged.

6.3.2 joinTerm is set incorrectly

Severity: *Low*

Context: ChallengeParticipation.sol#L94

Description: The `JoinTerms` struct has two variables, `FEE` and `VOUCHER`, intended to indicate whether a user joined a challenge by paying the full fee or by using a voucher. However, when a user uses a voucher, the `JoinTerms` is incorrectly set to `FEE`, which implies the user paid the full joining fee. As a result, the `validateUserJoinTerms` function always returns true regardless of the actual join method.

This bug causes the system to always treat users as having paid the full fee, ignoring when they join using a voucher.

```
/**
 * @notice Defines how a user joined a challenge
 */
enum JoinTerms {
    FEE, /// @dev Joined by paying entry fee
    VOUCHER /// @dev Joined using discount voucher
}

    function _joinChallenge(bytes32 _challengeId, address _user, uint64 _fundingPool
        ) internal {
        Challenge memory challenge = LibChallengeStorage.getChallenge(_challengeId);

        UserChallengeDetails memory _userChallengeDetails = UserChallengeDetails({
            status: ChallengeStatus.ACTIVE,
            fundingPool: _fundingPool,
            balance: _fundingPool,
            dailyBalance: _fundingPool,
            joinedAt: uint32(block.timestamp),
            firstTrade: 0,
            phase: 0,
            //@audit-issue : if the user use Vaucher, joinTerm should be = JoinTerms
                .Vaucher
            joinTerms: JoinTerms.FEE,
            accountLeverage: AccountLeverage({isActive: false, maxLeverage: 0}),
            openedPositions: 0,
            penaltyExpiry: 0
        });

    /**
     * @notice Validates that user joined via fee payment (not free entry)
     * @param _joinTerms The terms under which user joined the challenge
     * @custom:reverts VaultValidator__DidntJoinViaFee if user didn't pay entry fee
     */
    function validateUserJoinTerms(JoinTerms _joinTerms) internal pure {
        require(_joinTerms == JoinTerms.FEE, VaultValidator__DidntJoinViaFee());
```

```
    }
```

Recommendation: Fix the assignment logic to preserve the correct join method and ensure validateUserJoinTerms accurately reflects whether a voucher was used.

Resolution and Client comment: Resolved.

6.3.3 CHALLENGE_MANAGER can not remove user challenge

Severity: *Low*

Context: LibChallengeStorage.sol#L88

Description: The `removeUserChallenge` function is never used in the codebase and has `internal` visibility, which prevents it from being called externally. As a result, the `CHALLENGE_MANAGER` role can not remove a challenge from a user.

Challenges assigned to users can not be removed through normal role-based operations. Also, limits contract flexibility and may block certain workflows.

```
function removeUserChallenge(address _user, bytes32 _challengeId) internal {
        LibChallengeStorage.getLayout().userChallenges[_user].remove(_challengeId);
    }
```

Recommendation: Change the function's visibility to external or public if it needs to be called by CHALLENGE_MANAGER.

Resolution and Client comment: Resolved.

6.3.4 CHALLENGE_MANAGER can not update fee amount for funding pool

Severity: *Low*

Context: LibChallengeStorage.sol#L84

Description: The `setFeeAmountForFundingPoolfunction` is never used anywhere in the codebase and has internal visibility. Because of this, it can not be called externally, meaning the CHALLENGE_MANAGER role has no way to update the fee amount for a funding pool.

```
    function setFeeAmountForFundingPool(bytes32 _challengeId, uint64 _fundingPool,
        uint32 _feeAmount) internal {
        LibChallengeStorage.getLayout().feeAmounts[_challengeId][_fundingPool] =
            _feeAmount;
    }
```

Recommendation: Allow CHALLENGE_MANAGER to update the fee amount of a funding pool.

Resolution and Client comment: Acknowledged.

6.4 Information

6.4.1 LibRolesStorage layout is never used

Severity: *Information*

Context: LibRolesStorage.sol#L15

Description: The layout defined in the LibRolesStorage library, including the RolesStorage struct, is never used anywhere in the codebase. Only the constant role identifiers are referenced. While the library appears intended to handle role-related logic, this functionality is currently implemented in LibAppStorage instead.

```
//@audit never used
struct RolesStorage {
    mapping(address => mapping(bytes32 => bool)) roles;
}

library LibRolesStorage {
    bytes32 constant ROLES_STORAGE_POSITION = keccak256("diamond.standard.diamond.
        storage.roles");

    bytes32 public constant ORACLE_MANAGER = keccak256("ORACLE_MANAGER");
    bytes32 public constant KEEPER = keccak256("KEEPER");
    bytes32 public constant CHALLENGE_MANAGER = keccak256("CHALLENGE_MANAGER");

    //@audit never used
    function getLayout() internal pure returns (RolesStorage storage ds) {
        bytes32 position = ROLES_STORAGE_POSITION;
        assembly {
            ds.slot := position
        }
    }
}
```

Recommendation: Remove the unused code, or move the roles functionality from LibAppStorage to LibRolesStorage.

Resolution and Client comment: Acknowledged.

6.4.2 Unused referralMerkleRoot in LibFundsStorage

Severity: *Information*

Context: LibFundsStorage.sol#L5

Description: The storage variable referralMerkleRoot in LibFundsStorage is never used throughout the codebase. The Merkle root for referrals is actually set in LibReferralStorage when the KEEPER updates referral data via ReferralFacet. This makes the variable in LibFundsStorage redundant and potentially confusing.

```
    struct FundsStorage {
        bytes32 refferalMerkleRoot; //@audit-issue never used
        Wallets wallets;
    }
```

```
    function updateRefferalData(
        uint256 _timestamp,
        bytes32 _merkleRoot,
        uint256 _totalUsdc,
        uint256 _totalDay // pass usdcAmountInDay
    ) external onlyRole(LibRolesStorage.KEEPER) {
        LibRefferalStorage.getLayout().refferalMerkleRoot = _merkleRoot;

        // ...
    }
```

Recommendation: Remove the unused referralMerkleRoot variable from LibFundsStorage to clean up storage layout and prevent misunderstandings.

Resolution and Client comment: Acknowledged.

6.4.3 Unused ECDSA Library in ReferralFacet

Severity: *Information*

Context: RefferalFacet.sol#L30

Description: The ECDSA library from OpenZeppelin is imported but never used in the ReferralFacet contract, leading to unnecessary bytecode bloat and potential developer confusion.

```
contract RefferalFacet is BaseFacet {
    using ECDSA for bytes32;

    }
```

Recommendation: Remove the unused ECDSA import from ReferralFacet to reduce bytecode size and improve maintainability.

Resolution and Client comment: Acknowledged.

6.4.4 Remove unused DELTA constant

Severity: *Information*

Context: DayhubOracleFacet.sol#L57-L60

Description: When an asset price is updated through DayHubOracleFacet, the requirement to check the maximum allowed seconds for a valid price timestamp has been commented out and is no longer used. Since this requirement is no longer enforced and the DELTA constant is unused, the related code is redundant.

```
    function updatePrices(bytes calldata payload, bytes calldata signature) external
         {
        // Decode the payload
        PriceData memory priceData = abi.decode(payload, (PriceData));

        // Validate basic requirements
```

```
        require(priceData.priceInfo.length > 0, DayhubOracleFacet__InvalidPayload())
            ;

        for (uint256 index = 0; index < priceData.priceInfo.length; index++) {
            require(priceData.priceInfo[index].price > 0,
                DayhubOracleFacet__InvalidPrice());

            // require(
            //     priceData.priceInfo[index].priceTimestamp >= block.timestamp -
                DELTA,
            //     DayhubOracleFacet__InvalidTimestamp(priceData.priceInfo[index].
                priceTimestamp, block.timestamp)
            // );
        }

        require(priceData.nonce > LibOracleStorage.getLayout().nonce,
            DayhubOracleFacet__InvalidNonce());

                // ....
}
```

Recommendation: Remove the unused DELTA constant and the commented-out price freshness check to clean up the codebase.

Resolution and Client comment: Acknowledged.

6.4.5 validateAssetAllowed() is never used

Severity: *Information*

Context: ChallengeValidator.sol#L100

Description: The validateAssetAllowed function in the ChallengeValidator and the PositionValidator is never used.

```
    /**
     * @notice Validates that an asset is whitelisted for the challenge
     * @param asset The asset identifier to validate
     * @param allowedAssets Array of allowed asset identifiers for the challenge
     * @custom:reverts ChallengeValidator__InvalidAsset if the asset is not
         whitelisted
     */
    function validateAssetAllowed(bytes32 asset, bytes32[] memory allowedAssets)
        internal pure {
        for (uint256 i = 0; i < allowedAssets.length; i++) {
            if (asset == allowedAssets[i]) return;
        }
        revert ChallengeValidator__InvalidAsset();
    }
}
```

Recommendation: Remove the redundant function.

Resolution and Client comment: Resolved.

6.4.6 assetSeen() is never used

Severity: *Information*

Context: KeeperFacet.sol#L687

Description: The `_assetSeen` function in the KeeperFacet is never used.

```
    function _assetSeen(bytes32[] memory seenAssets, bytes32 asset) internal pure
        returns (bool) {
        for (uint256 i = 0; i < seenAssets.length; i++) {
            if (seenAssets[i] == asset) return true;
        }
        return false;
    }
```

Recommendation: Remove the redundant function.

Resolution and Client comment: Resolved.

6.4.7 validateAttempts() is never used

Severity: *Information*

Context: ChallengeValidator.sol#L141

Description: The `validateAttempts` function in the ChallengeValidator is never used.

```
    function validateAttempts(uint256 maxAttempts, uint256 userActiveAttempts)
        internal {
        //@audit-issue I-07, never used, used validateMaxActiveAttempts() instead
        require(
            userActiveAttempts < maxAttempts,
            ChallengeValidator__ReachedMaxActiveAttempts(userActiveAttempts,
                maxAttempts)
        );
    }
```

Recommendation: Remove the redundant function.

Resolution and Client comment: Resolved.

6.4.8 Double registration of position

Severity: *Information*

Context: LibPositionStorage.sol#L71

Description: When a position is opened by the Keeper, it is added to the system via the addPosition function. Inside `addPosition`, the `registerPosition` function is called:

```
  function addPosition(bytes32 _challengeId, bytes32 _positionId, Position memory
      _position) internal {
        checkAddUser(_position.user);

        registerPosition(_positionId, _position);
```

```
        UserChallengeDetails memory _ucd = LibChallengeStorage.
            getUserChallengeDetails(
            _position.user,
            _challengeId,
            _position.attemptId
        );
}
```

There is no need to register the position again when it is added, because for a position to be opened, it must already have been registered beforehand. Calling `registerPosition` inside `addPosition` is redundant.

Recommendation: Remove the registerPosition call from addPosition function.

Resolution and Client comment: Acknowledged.

6.4.9 Redundant user existence check

Severity: *Information*

Context: LibPositionStorage.sol#L38

Description: Redundant checks are performed before adding users, increasing gas costs without providing additional safety.

Before registering or adding a position, the system checks whether a user is already registered. The function `checkAddUser` verifies if the user exists in the enumerable set `users`, and if not, it adds the user.

However, this check is unnecessary because the underlying `EnumerableSet.add` function from OpenZeppelin already performs this verification internally. It returns **`true`** if the user was successfully added and **`false`** if the user already existed in the set.

As a result, the pre-check duplicates logic and wastes gas without improving correctness.

```
    function checkAddUser(address _user) internal {
        PositionStorage storage ps = getLayout();
        if (!ps.users.contains(_user)) {
            ps.users.add(_user);
        }
    }
```

There is no need to register the position again when it is added, because for a position to be opened, it must already have been registered beforehand. Calling `registerPosition` inside `addPosition` is redundant.

Recommendation: Remove the redundant existence check (no need from `checkAddUser` function).

Resolution and Client comment: Resolved.

6.4.10 validateMinTradingDays function is not used

Severity: *Information*

Context: ChallengeRulesValidator.sol#L155

Description: The `validateMinTradingDays` function is never used, leading to duplicated logic

The purpose of `validateMinTradingDays` is to ensure that the number of unique trading days is not less than the required minimum trading days for a phase. However, instead of using this function, the check is performed directly within the `isPhasePassed` function.

As a result, `validateMinTradingDays` remains unused in the codebase, introducing redundant logic and making the code harder to maintain.

```
        if (stats.uniqueDaysTraded < phase.phaseProps.minTradingDays) {
            return false;
        }
```

Recommendation: Replace the inline validation in isPhasePassed with a call to `validateMinTradingDays` to ensure consistency and maintainability.

Resolution and Client comment: Acknowledged.

6.4.11 Missing event emission in setMaxAccountLeverage function

Severity: *Information*

Context: ChallengeParticipation.sol#L124

Description: Lack of event emission reduces transparency and makes it harder to track leverage changes on-chain.

The `setMaxAccountLeverage` function allows a user to update their maximum leverage, but no event is emitted when this action occurs. Without an event, it is difficult for off-chain services, indexers, or monitoring tools to detect and track leverage changes. This reduces visibility and may impact auditing, monitoring, or debugging of user activity.

Recommendation: Emit an event whenever a user sets or updates their maximum account leverage to ensure full traceability.

Resolution and Client comment: Acknowledged.

6.4.12 validateEffectiveLeverage function is never used

Severity: *Information*

Context: LeverageValidator.sol#L28

Description: The `validateEffectiveLeverage` function in the LeverageValidator is never used.

```
    function validateEffectiveLeverage(
        uint256 existingPositionValue,
        uint256 newPositionValue,
        uint256 equity,
```

```
        uint256 maxLeverage,
        bool isCustom,
        uint256 userLeverageLimit
    ) internal pure {
        require(equity > 0, PositionFacet__NotEnoughEquity());

        uint256 totalPositionValue = existingPositionValue + newPositionValue;
        uint256 effectiveLeverage = (totalPositionValue * LibAppStorage.PRECISION) /
             equity;

        if (!isCustom) {
            require(effectiveLeverage <= maxLeverage,
                ChallengeValidator__LeverageOverChallengeLimit());
        } else {
            require(effectiveLeverage <= userLeverageLimit,
                ChallengeValidator__ReachedMaxLeverage());
        }
    }
```

Recommendation: Remove the redundant function.

Resolution and Client comment: Acknowledged.

6.4.13 removeUserPosition is never used

Severity: *Information*

Context: LibPositionStorage.sol#L106

Description: The `validateEffectiveLeverage` function in the LibPositionStorageis never used.

```
    function removeUserPosition(bytes32 _challengeId, address _user, bytes32
        _positionId) internal {
        PositionStorage storage ps = getLayout();

        // Get the position to find the attemptId
        Position storage position = ps.positionsDetails[_positionId];
        require(position.user == _user, "Position does not belong to user");

        uint256 attemptId = position.attemptId;

        // Remove position from user's position set
        EnumerableSet.Bytes32Set storage userChallengePositionsToAttempt = ps
        .userChallengePositionsToAttempt[_user][_challengeId][attemptId].positionSet
            ;
        require(userChallengePositionsToAttempt.contains(_positionId), "Position not
             found in user's positions");

        userChallengePositionsToAttempt.remove(_positionId);

        ps.positions.remove(_positionId);
    }
```

Recommendation: Remove the redundant function.

Resolution and Client comment: Resolved.

6.4.14 isMarketEntry() is never used

Severity: *Information*

Context: KeeperFacet.sol#L575

Description: The `_isMarketEntry` function in the KeeperFacet is never used.

```
    function _isMarketEntry(PositionProps memory props) internal pure returns (bool)
        {
        return props.entryPrice == 0;
    }
```

Recommendation: Remove the redundant function.

Resolution and Client comment: Resolved.

6.4.15 Rename OpenOrder event to RegisterOrder to reflect actual behavior

Severity: *Information*

Context: PositionFacet.sol#L115

Description: In `_processOrder`, the position is not actually opened; it is merely registered. Emitting an event named OpenOrder gives a false impression of the operation being performed.

Recommendation: Rename the event from OpenOrder to RegisterOrder to accurately reflect the action.

Resolution and Client comment: Acknowledged.

6.4.16 Unnecessary self-call to getPrice increases gas usage

Severity: *Information*

Context: Global

Description: Unnecessary self-calls to `getPrice` waste gas and add overhead

When the contract needs to retrieve the price of an asset, it makes an external call to itself by calling the `getPrice` function. This self-call is unnecessary, since the same logic could be executed through an internal function call, which is cheaper and more efficient.

Recommendation: Replace the self-call to getPrice with a direct internal function call to avoid unnecessary gas costs.

Resolution and Client comment: Acknowledged.