SALUS SECURITY

JAN 2025

CODE SECURITY ASSESSMENT

DEDERI

Overview

Project Summary

- Name: Artura - Bitperp_contracts
- Platform: EVM-compatible chains
- Language: Solidity
- Repository:
    - https://github.com/bitperp/bitperp-contracts
- Audit Range: See Appendix - 1

Project Dashboard

Application Summary

Name	Artura - Bitperp_contracts
Version	v4
Type	Solidity
Dates	Jan 03 2025
Logs	Nov 19 2024; Dec 07 2024; Jan 02 2025; Jan 03 2025

Vulnerability Summary

Total High-Severity issues	1
Total Medium-Severity issues	4
Total Low-Severity issues	12
Total informational issues	4
Total	21

Contact

E-mail: support@salusec.io

Risk Level Description

High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for clients' reputations or serious financial implications for clients and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to a moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or defense in depth.

Content

Introduction

1.1 About SALUS

At Salus Security, we are in the business of trust.

We are dedicated to tackling the toughest security challenges facing the industry today. By building foundational trust in technology and infrastructure through security, we help clients to lead their respective industries and unlock their full Web3 potential.

Our team of security experts employ industry-leading proof-of-concept (PoC) methodology for demonstrating smart contract vulnerabilities, coupled with advanced red teaming capabilities and a stereoscopic vulnerability detection service, to deliver comprehensive security assessments that allow clients to stay ahead of the curve.

In addition to smart contract audits and red teaming, our Rapid Detection Service for smart contracts aims to make security accessible to all. This high calibre, yet cost-efficient, security tool has been designed to support a wide range of business needs including investment due diligence, security and code quality assessments, and code optimisation.

We are reachable on Telegram (https://t.me/salusec), Twitter (https://twitter.com/salus_sec), or Email (support@salusec.io).

1.2 Audit Breakdown

The objective was to evaluate the repository for security-related issues, code quality, and adherence to specifications and best practices. Possible issues we looked for included (but are not limited to):

- Risky external calls
- Integer overflow/underflow
- Transaction-ordering dependence
- Timestamp dependence
- Access control
- Call stack limits and mishandled exceptions
- Number rounding errors
- Centralization of power
- Logical oversights and denial of service
- Business logic specification
- Code clones, functionality duplication

1.3 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release and does not give any warranties on finding all possible security issues with the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues.

Findings

2.1 Summary of Findings

ID	Title	Severity	Category	Status
1	Mismatch OrderType	High	Business Logic	Resolved
2	Erc4626 inflation attack	Medium	Business Logic	Resolved
3	Lack of spreadReductionId validation	Medium	Business Logic	Resolved
4	Depth mechanism incomplete	Medium	Business Logic	Resolved
5	Missing slippage check	Medium	Business Logic	Resolved
6	Potential data truncation	Low	Numerics	Resolved
7	Inconsistent Max SL and Max Gain Parameters Across Contracts	Low	Inconsistency	Resolved
8	Insufficient validation of newTp and newSl	Low	Data Validation	Resolved
9	WithdrawEpochsTimelock is discrepant with document	Low	Inconsistency	Resolved
10	Staking reward locked in contract	Low	Business Logic	Resolved
11	Miscalculation of updateSl fee	Low	Business Logic	Resolved
12	Incomplete property of NFT	Low	Business Logic	Resolved
13	Missing leverage checks	Low	Data Validation	Acknowledged
14	Use safeTransfer()/safeTransferFrom() instead of transfer()/transferFrom()	Low	Risky External Calls	Resolved
15	Missing remove mechanism	Low	Business Logic	Resolved
16	Missing events for functions that change critical state	Low	Logging	Resolved
17	Missing two-step transfer ownership pattern	Informational	Business logic	Resolved
18	Missing zero address checks	Informational	Data Validation	Resolved
19	Gas optimization suggestions	Informational	Gas Optimization	Resolved

20	Ineffective check	Informational	Gas Optimization	Resolved
21	Custom error don’t contain revert reason	Informational	Code Quality	Resolved

2.2 Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

1. Mismatch OrderType

Severity: High	Category: Business Logic

Target:
- contracts/pricefeeds/PriceAggregatorV7.sol
- contracts/interfaces/StorageInterfaceV7.sol
- contracts/TradingV7.sol

Description

The `OrderType` has two definitions:

contracts/pricefeeds/PriceAggregatorV7.sol:L19-L23

```
enum OrderType {
    MARKET_OPEN,
    MARKET_CLOSE,
    UPDATE_SL
 }
```

contracts/interfaces/StorageInterfaceV7.sol:L129-L145

```
enum OrderType {
    MARKET_OPEN,
    MARKET_CLOSE,
    LIMIT_OPEN,
    LIMIT_CLOSE,
    UPDATE_SL
}
```

In the `TradingV7.sol`'s `updateSl` function, the code will execute to the `aggregator.getPrice` branch with the `AggregatorInterfaceV7.OrderType.UPDATE_SL` parameter, which is 4. However, the max number of `PriceAggregatorV7.OrderType` is 2. This will cause an EVMerror.

contracts/TradingV7.sol:L351-L354

```
uint256 orderId = aggregator.getPrice(
     pairIndex,
     AggregatorInterfaceV7.OrderType.UPDATE_SL
);
```

Recommendation

Fix the mismatch `OrderType`.

Status

The team has resolved this issue in commit db43f90.

2. Erc4626 inflation attack

Severity: Medium	Category: Business Logic
Target: - contracts/staking/Vault.sol	

Description

ERC-4626 vaults are vulnerable to inflation attacks; an attacker can "donate" funds to the vault without making a formal deposit, inadvertently increasing the value of shares. In certain scenarios, including when an unsuspecting user is the first depositor, the attacker can gain more value than they donated, effectively stealing value from the first depositor.

Fortunately, the version of ERC4624 that Vault inherited has fixed this problem, but Vault's rewrite of some functions destroyed this fix. And the Donation attack can be achieved through the `distributeReward` function.

contracts/staking/Vault.sol:L15, L430-450

```
contract Vault is ERC20Upgradeable, ERC4626Upgradeable, OwnableUpgradeable, IVaultToken
{

function _convertToShares(uint256 assets, Math.Rounding rounding)
    internal
    view
    override
    returns (uint256 shares)
{
    return assets.mulDiv(PRECISION, shareToAssetsPrice, rounding);
}

function _convertToAssets(uint256 shares, Math.Rounding rounding)
    internal
    view
    override
    returns (uint256 assets)
{
    // Prevent overflow when called from maxDeposit with maxMint = uint.max
    if (shares == type(uint256).max && shareToAssetsPrice >= PRECISION) {
        return shares;
    }
    return shares.mulDiv(shareToAssetsPrice, PRECISION, rounding);
}
function distributeReward(uint256 assets) external {
    address sender = _msgSender();
    SafeERC20.safeTransferFrom(_assetIERC20(), sender, address(this), assets);

    accRewardsPerToken += assets * PRECISION / totalSupply();
    updateShareToAssetsPrice();

    totalRewards += assets;
    emit RewardDistributed(sender, assets);
```

```
    }
function maxAccPnlPerToken() public view returns (uint256) {
    // PRECISION
    return PRECISION + accRewardsPerToken;
}

function updateShareToAssetsPrice() private {
    // PRECISION
    shareToAssetsPrice = maxAccPnlPerToken() - (accPnlPerTokenUsed > 0 ?
uint256(accPnlPerTokenUsed) : uint256(0));

    emit ShareToAssetsPriceUpdated(shareToAssetsPrice);
}
```

@openzeppelin/contracts-upgradeable/token/ERC20/extensions/ERC4626Upgradeable.sol

```
function _convertToShares(uint256 assets, Math.Rounding rounding) internal view virtual
returns (uint256) {
    return assets.mulDiv(totalSupply() + 10 ** _decimalsOffset(), totalAssets() + 1,
rounding);
}

/**
 * @dev Internal conversion function (from shares to assets) with support for rounding
direction.
 */
function _convertToAssets(uint256 shares, Math.Rounding rounding) internal view virtual
returns (uint256) {
    return shares.mulDiv(totalAssets() + 1, totalSupply() + 10 ** _decimalsOffset(),
rounding);
}
```

Recommendation

Using the OpenZeppelin's fix or add access control to the `distributeReward` function.

Status

This issue has been resolved by the team with commit 518b03e.

3. Lack of spreadReductionId validation

Severity: Medium	Category: Business Logic

Target:
- contracts/TradingV7.sol
- contracts/TradingCallbacksV7

Description

In the TradingV7 contract, the `openTrade` function allows users to open new trades by providing various parameters, including `spreadReductionId`. However, there is a lack of proper validation on the `spreadReductionId` parameter. This omission can be exploited by users to obtain an unintended advantage in trade executions.

contracts/TradingV7.sol:L142-L208

```
function openTrade(
    StorageInterfaceV7.Trade memory t,
    StorageInterfaceV7.OpenLimitOrderType orderType, // LEGACY => market
    uint256 spreadReductionId,
    uint256 slippageP // for market orders only
) external notContract notDone {
    ...
    storageT.storeOpenLimitOrder(
        StorageInterfaceV7.OpenLimitOrder(
            sender,
            t.pairIndex,
            index,
            t.positionSizeDai,
            spreadReductionId > 0 ? storageT.spreadReductionsP(spreadReductionId - 1) :
0,
            t.buy,
            t.leverage,
            t.tp,
            t.sl,
            t.openPrice,
            t.openPrice,
            block.number,
            0
        )
    );
    ...
}
```

In the code above, `spreadReductionId` is used directly without proper validation to index into the `spreadReductionsP` array:

```
uint256[5] public spreadReductionsP = [15, 20, 25, 30, 35]; // %
```

By providing a `spreadReductionId` value of 5, which is the length of the array, users can manipulate the calculation in the `marketExecutionPrice` function:

contracts/TradingCallbacksV7:L681-L689

```
function marketExecutionPrice(uint256 price, uint256 spreadP, uint256 spreadReductionP,
bool long)
```

```
    private
    pure
    returns (uint256)
{
    uint256 priceDiff = price * (spreadP - spreadP * spreadReductionP / 100) / 100 /
PRECISION;

    return long ? price + priceDiff : price - priceDiff;
}
```

This lack of validation can lead to the following issues:

- Unauthorized Access to Spread Reductions: Users can access higher levels of spread reductions without meeting the necessary criteria.
- Economic Exploitation: By reducing the spread artificially, users can execute trades at more favorable prices, potentially leading to losses for other participants or the platform.

Recommendation

Validate that the `spreadReductionId` is within the acceptable range and corresponds to the user's authorized level.

Status

This issue has been resolved by the team with commit c61b7a8.

4. Depth mechanism incomplete

Severity: Medium	Category: Business Logic

Target:
- contracts/PairInfos.sol

Description

The PairInfos contract includes a mechanism to adjust the price impact based on the trade size relative to the market depth, represented by the `onePercentDepth` parameter. However, the current implementation of the `setOnePercentDepth` function does not dynamically reflect the actual market depth in real-time.

This static approach can lead to inaccurate price impact calculations, opening up opportunities for arbitrage and potentially causing significant slippage between the expected and actual execution prices for users.

contracts/PairInfos.sol:L134-L142

```
// Set one percent depth for pair
function setOnePercentDepth(uint256 pairIndex, uint256 valueAbove, uint256 valueBelow)
public onlyManager {
    PairParams storage p = pairParams[pairIndex];

    p.onePercentDepthAbove = valueAbove;
    p.onePercentDepthBelow = valueBelow;

    emit OnePercentDepthUpdated(pairIndex, valueAbove, valueBelow);
}
```

contracts/PairInfos.sol:L285-L309

```
function getTradePriceImpactPure()
{
    if (onePercentDepth == 0) {
        return (0, openPrice);
     }

     priceImpactP = (startOpenInterest + tradeOpenInterest / 2) * PRECISION / 1e18 /
onePercentDepth;

     uint256 priceImpact = priceImpactP * openPrice / PRECISION / 100;

     priceAfterImpact = long ? openPrice + priceImpact : openPrice - priceImpact;
}
```

Recommendation

Implement the depth calculation function or increase the frequency of depth parameter updates.

Status

This issue has been resolved by the team with commit 6a1c7e3.

5. Missing slippage check

Severity: Medium	Category: Business Logic

Target:
- contracts/TradingV7.sol
- contracts/PriceAggregatorV7.sol
- contracts/TradingCallbacksV7.sol

Description

In the TradingV7 contract, the `openTrade` function allows users to open new trades by providing various parameters, including `slippageP`. which represents the allowable slippage percentage for market orders. However, there is a lack of proper validation on the `slippageP` parameter.

contracts/TradingV7.sol:L142-L228

```
function openTrade(
   StorageInterfaceV7.Trade memory t,
   StorageInterfaceV7.OpenLimitOrderType orderType, // LEGACY => market
   uint256 spreadReductionId,
   uint256 slippageP // for market orders only
) external notContract notDone {
   require(!isPaused, "PAUSED");
   require(t.openPrice * slippageP < type(uint256).max, "OVERFLOW");
   ...
  } else {
    uint256 orderId = aggregator.getPrice(t.pairIndex,
AggregatorInterfaceV7.OrderType.MARKET_OPEN);

    storageT.storePendingMarketOrder(
        StorageInterfaceV7.PendingMarketOrder(
            StorageInterfaceV7.Trade(
                sender, t.pairIndex, 0, 0, t.positionSizeDai, 0, t.buy, t.leverage,
t.tp, t.sl
                  ),
                  0,
                  t.openPrice,
                  slippageP,
                  spreadReductionId > 0 ? storageT.spreadReductionsP(spreadReductionId -
1) : 0,
                  0
              ),
              orderId,
              true
          );

    emit MarketOrderInitiated(orderId, sender, t.pairIndex, true);
  }
}
```

If a user provides a very large `slippageP` value (e.g., greater than 100 * PRECISION), the `openTrade` function initiates a price request from the oracle. Upon receiving the price, the oracle calls `fulfillPrice`:

contracts/PriceAggregatorV7.sol:L163-L188

```
// Fulfill on-demand price requests
function fulfillPrice(uint256 orderId, uint256 price) external onlyOracles {
    Order memory r = orders[orderId];

    require(r.initiated, "WRONG_ORDER");

    TradingCallbacksInterfaceV7.AggregatorAnswer memory a;

    a.orderId = orderId;
    a.price = price;
    a.spreadP = pairsStorage.pairSpreadP(r.pairIndex);

    TradingCallbacksInterfaceV7 c = TradingCallbacksInterfaceV7(storageT.callbacks());

    if (r.orderType == OrderType.MARKET_OPEN) {
        c.openTradeMarketCallback(a);
    } else if (r.orderType == OrderType.MARKET_CLOSE) {
        c.closeTradeMarketCallback(a);
    } else {
        c.updateSlCallback(a);
    }

    delete orders[orderId];

    emit PriceReceived(orderId, msg.sender, r.pairIndex, price);
}
```

which in turn calls `openTradeMarketCallback`, which can cause an underflow in the `openTradeMarketCallback` function:

contracts/TradingCallbacksV7.sol:L228-L286

```
// Callbacks
function openTradeMarketCallback(AggregatorAnswer memory a) external onlyPriceAggregator
notDone {
    StorageInterfaceV7.PendingMarketOrder memory o = getPendingMarketOrder(a.orderId);

    if (o.block == 0) return; //@audit need check, test

    StorageInterfaceV7.Trade memory t = o.trade;

    // priceImpactP: PERCENTAGE
    (uint256 priceImpactP, uint256 priceAfterImpact) = pairInfos.getTradePriceImpact(
        marketExecutionPrice(a.price, a.spreadP, o.spreadReductionP, t.buy),
        t.pairIndex,
        t.buy,
        t.positionSizeDai * t.leverage
    );

    t.openPrice = priceAfterImpact;

    uint256 maxSlippage = o.wantedPrice * o.slippageP / 100 / PRECISION;

    //    What the Aggregaotr Answer looks like
    //    struct AggregatorAnswer {
    //        uint256 orderId;
    //        uint256 price;
    //        uint256 spreadP;
    //    }

    if (
        isPaused || a.price == 0
            || (t.buy ? t.openPrice > o.wantedPrice + maxSlippage : t.openPrice <
o.wantedPrice - maxSlippage)
            || (t.tp > 0 && (t.buy ? t.openPrice >= t.tp : t.openPrice <= t.tp)) // tp
```

```
invalid
            || (t.sl > 0 && (t.buy ? t.openPrice <= t.sl : t.openPrice >= t.sl)) // sl
invalid
            || !withinExposureLimits(t.pairIndex, t.buy, t.positionSizeDai, t.leverage)
            || priceImpactP * t.leverage > pairInfos.maxNegativePnlOnOpenP()
     ) {
        // Handle order cancellation
        ...
    }
    ...
}
```

If the oracle's callback function reverts due to this underflow, and if the oracle lacks proper error handling, it may result in a Denial of Service (DoS) condition. The oracle could become stuck processing the problematic order, affecting its ability to handle other users' orders.

And there is an order cancellation limitation, The `openTradeMarketTimeout` function, which allows users to cancel their own pending orders after a timeout, requires user intervention and does not address the root cause in the oracle's processing logic.

Recommendation

Implement proper validation on the `slippageP` parameter within the `openTrade` function, and implement robust error handling in `fulfillPrice` and `openTradeMarketCallback` functions.

Status

This issue has been resolved by the team with commit 0255fc5.

6. Potential data truncation

Severity: Low	Category: Numerics
Target: - contracts/BTPVesting.sol	

Description

The `AllocationSplit` and `AllocationState` structs are defined as follows:

contracts/BTPVesting.sol:L19-L29

```
struct AllocationSplit {
    address recipient;
    uint64 points;
    uint64 allocationTime;
}

struct AllocationState {
    uint64 points;
    uint64 allocationTime;
    uint128 claimed;
}
```

But state variables `totalPoints` and `totalAllocation` are defined as uint256:

```
uint256 public immutable totalPoints;
uint256 public immutable totalAllocation;
```

In the constructor, `allocations[recipient].points` is assigned using a uint24 cast, which is smaller than the defined uint64 type.

contracts/BTPVesting.sol:L47-L74

```
constructor(
    IERC20 vestingToken_,
    uint256 totalAllocation_,
    uint256 vestingStart_,
    AllocationSplit[] memory allocationSplits
) {
    ...
    for (uint256 i; i < loopEnd; ) {
        address recipient = allocationSplits[i].recipient;
        uint64 allocationTime = allocationSplits[i].allocationTime;
        uint256 points = allocationSplits[i].points;
        if (points == 0) revert ZeroAllocation();
        if (allocationTime == 0) revert ZeroAllocationTime();
        if (allocations[recipient].allocationTime > 0) revert DuplicateAllocation();
        total += points;
        allocations[recipient].points = uint24(points);
        allocations[recipient].allocationTime = allocationTime;
        unchecked {
            ++i;
        }
    }
    totalPoints = total;
}
```

In the `_claim` function, `allocations[account].claimed` is updated using a uint128 cast, while `totalAllocation` is a uint256.

contracts/BTPVesting.sol:L92-L105

```
function _claim(
    address account,
    uint256 points,
    uint256 claimed,
    uint256 allocationTime
) private returns (uint256 claimedUpdated) {
    if (points == 0) revert NothingToClaim();
    uint256 claimable = _claimableAt(block.timestamp, points, claimed, allocationTime);
    if (claimable == 0) revert NothingToClaim();
    claimedUpdated = claimed + claimable;
    allocations[account].claimed = uint128(claimedUpdated);
    // We send to delegate for possible zaps
    vestingToken.transfer(msg.sender, claimable);
}
```

Data truncation during points assignment and claimed amounts can result in incorrect allocation distributions, inaccurate amounts of claimed tokens.

Recommendation

Use consistent data types or OpenZeppelin SafeCast library.

Status

This issue has been resolved by the team with commit 7843ad7.

7. Inconsistent Max SL and Max Gain parameters across contracts

Severity: Low	Category: Inconsistency

Target:
- contracts/TradingV7.sol
- contracts/TradingCallbacksV7.sol
- contracts/TradingStorageV7.sol

Description

In the TradingV7 and TradingCallbacksV7 contracts, `MAX_SL_P` (maximum stop-loss percentage) and `MAX_GAIN_P` (maximum gain percentage) are defined as constants:

contracts/TradingV7.sol:L19

```
uint256 constant MAX_SL_P = 75; // -75% PNL
```

contracts/TradingCallbacksV7.sol:L18-19

```
uint256 constant MAX_SL_P = 75; // -75% PNL
uint256 constant MAX_GAIN_P = 900; // 900% PnL (10x)
```

However, in the TradingStorageV7 contract, these parameters are defined as public variables that can be modified by the governance (gov) address:

contracts/TradingStorageV7.sol:L30-31,L297-L301,L309-313

```
function setMaxGainP(uint256 _max) external onlyGov {
    require(_max >= 300);
    maxGainP = _max;
    emit NumberUpdated("maxGainP", _max);
}
function setMaxGainP(uint256 _max) external onlyGov {
    require(_max >= 300);
    maxGainP = _max;
    emit NumberUpdated("maxGainP", _max);
}
function setMaxSlP(uint256 _max) external onlyGov {
    require(_max >= 50);
    maxSlP = _max;
    emit NumberUpdated("maxSlP", _max);
}
```

This inconsistency results in data discrepancies between contracts, the actual values used in calculations (`MAX_SL_P` and `MAX_GAIN_P`) are immutable.

Recommendation

Ensure that `MAX_SL_P` and `MAX_GAIN_P` are consistently defined and managed across all related contracts.

Status

This issue has been resolved by the team with commit 15d973f.

8. Insufficient validation of newTp and newSl

Severity: Low	Category: Data Validation

Target:
- contracts/TradingV7.sol

Description

The TradingV7 contract has three functions—`updateOpenLimitOrder`, `updateTp`, and `updateSl`—that allow users to update the take profit (`newTp`) and stop loss (`newSl`) values for their trades. However, these functions lack consistent and comprehensive validation of the new tp and sl values, potentially allowing users to set unreasonable or extreme values that could lead to unexpected behavior or financial risk to the platform.

contracts/TradingV7.sol:L311-L361

```
// Manage limit order (TP/SL)
function updateTp(uint256 pairIndex, uint256 index, uint256 newTp) external notContract
notDone {
    //no checks for newTp
    ...
}

function updateSl(uint256 pairIndex, uint256 index, uint256 newSl) external notContract
notDone {
    address sender = _msgSender();

    StorageInterfaceV7.Trade memory t = storageT.openTrades(sender, pairIndex, index);

    require(t.leverage > 0, "NO_TRADE");

    uint256 maxSlDist = t.openPrice * MAX_SL_P / 100 / t.leverage;

    require(
        newSl == 0 || (t.buy ? newSl >= t.openPrice - maxSlDist : newSl <= t.openPrice +
maxSlDist), "SL_TOO_BIG"
    );
    ...
}
```

contracts/TradingV7.sol:L267-L296

```
// Manage limit order (OPEN)
function updateOpenLimitOrder(
    uint256 pairIndex,
    uint256 index,
    uint256 price, // PRECISION
    uint256 tp,
    uint256 sl
) external notContract notDone {
    address sender = _msgSender();

    require(storageT.hasOpenLimitOrder(sender, pairIndex, index), "NO_LIMIT");

    StorageInterfaceV7.OpenLimitOrder memory o = storageT.getOpenLimitOrder(sender,
pairIndex, index);

    require(tp == 0 || (o.buy ? tp > price : tp < price), "WRONG_TP");
```

```
    require(sl == 0 || (o.buy ? sl < price : sl > price), "WRONG_SL");

    o.minPrice = price;
    o.maxPrice = price;
    o.tp = tp;
    o.sl = sl;

    storageT.updateOpenLimitOrder(o);
    storageT.callbacks().setTradeLastUpdated(
        sender, pairIndex, index, TradingCallbacksInterfaceV7.TradeType.LIMIT,
block.number
    );

    emit OpenLimitUpdated(sender, pairIndex, index, price, tp, sl);
}
```

Users could set take profit or stop loss values that exceed the platform's intended limits, potentially leading to inconsistent and excessive losses.

Recommendation

Consider using the following codes to verify:

```
//codes to verify newSl
uint256 maxSlDist = price * MAX_SL_P / 100 / leverage;

require(
    sl == 0 || (buy ? (sl >= price - maxSlDist && sl < price) : (sl <= price + maxSlDist
&& sl > price)),
     "WRONG_SL"
 );

//codes to verify newTp

uint256 maxTpDist = price * maxGainP / 100 / leverage;

require(
    tp == 0 || (buy ? (tp > price && tp <= price + maxTpDist) : (tp < price && tp >=
price - maxTpDist)),
     "WRONG_TP
  );
```

Status

This issue has been resolved by the team with commit 15d973f.

9. WithdrawEpochsTimelock is discrepant with document

Severity: Low	Category: Inconsistency
Target: - contracts/staking/Vault.sol	

Description

There is a mismatch between the vault's documentation and its implementation regarding the withdrawal lock-up periods based on the vault's collateralization ratio.

The `withdrawEpochsTimelock` function:

contracts/staking/Vault.sol:L342-L349

```
function withdrawEpochsTimelock() public view returns (uint256) {
   uint256 collatP = collateralizationP();
   uint256 overCollatP = (collatP - Math.min(collatP, 100 * PRECISION));

   return overCollatP > withdrawLockThresholdsP[1]
       ? WITHDRAW_EPOCHS_LOCKS[2]
       : overCollatP > withdrawLockThresholdsP[0] ? WITHDRAW_EPOCHS_LOCKS[1] :
WITHDRAW_EPOCHS_LOCKS[0];}
```

Documentation: collateralization at 120% or more needs 1 epoch lock-up period.

Code: collateralization at 120% needs 2 epoch lock-up period.

Same as collateralization at 110%.

Specifically, the code uses strict inequality conditions (>) instead of inclusive conditions (>=), causing the lock-up periods at exact boundary values (110% and 120% collateralization) to differ from what the documentation specifies. This discrepancy can lead to user confusion and incorrect expectations about withdrawal timelines.

Recommendation

Consider using inclusive conditions (>=) or modifying the documentation.

Status

This issue has been resolved by the team in the document.

10. Staking reward locked in contract

Severity: Low	Category: Business Logic

Target:
- contracts/TradingCallbacksV7.sol

Description

In the TradingCallbacksV7 contract, the `unregisterTrade` function is responsible for handling the closure of trades and distributing various fees and rewards to users.

Within this function, when the `sssFeeP` is non-zero, the contract calculates a staking reward (`reward3`). However, the distribution of this reward is not executed due to the corresponding function call being commented out, which leads to the staking rewards being locked in the contract, preventing users from accessing their earned rewards.

TradingCallbacksV7.sol:L558-L625

```
function unregisterTrade(
    StorageInterfaceV7.Trade memory trade,
    int256 percentProfit, // PRECISION
    uint256 currentDaiPos, // 1e18
    uint256 initialDaiPos, // 1e18
    uint256 closingFeeDai, // 1e18
    uint256 tokenPriceDai // PRECISION
) private returns (uint256 daiSentToTrader) {

    // 3.1 If collateral in storage (opened after update)
    if (trade.positionSizeDai > 0) {
        // 3.1.1 DAI vault reward
        v.reward2 = closingFeeDai * daiVaultFeeP / 100;
        transferFromStorageToAddress(address(this), v.reward2);
        vault.distributeReward(v.reward2);

        emit DaiVaultFeeCharged(trade.trader, v.reward2);

        // 3.1.2 SSS reward
        v.reward3 = closingFeeDai * sssFeeP / 100;

        // distributeStakingReward(trade.trader, v.reward3);

        // 3.1.3 Take DAI from vault if winning trade
        // or send DAI to vault if losing trade
        uint256 daiLeftInStorage = currentDaiPos - v.reward3 - v.reward2;
    ...
}
```

Recommendation

Implement the `distributeStakingReward` function.

Status

This issue has been resolved by the team with commit e1a56c5.

11. Miscalculation of updateSl fee

Severity: Low	Category: Business Logic

Target:
- contracts/TradingCallbacksV7.sol
- contracts/TradingStorageV7.sol

Description

In the TradingCallbacksV7 contract, the `updateSlCallback` function is intended to charge a fee at half the normal rate when a user updates their stop-loss (SL) order. However, due to an incorrect calculation, the fee charged is actually a quarter of the intended amount.

contracts/TradingCallbacksV7.sol:L470-L510

```
function updateSlCallback(AggregatorAnswer memory a) external onlyPriceAggregator
notDone {
    ...
    if (t.leverage > 0) {
        StorageInterfaceV7.TradeInfo memory i = storageT.openTradesInfo(o.trader,
o.pairIndex, o.index);
        Values memory v;
        v.tokenPriceDai = aggregator.tokenPriceDai();
        v.levPosDai = t.initialPosToken * i.tokenPriceDai * t.leverage / PRECISION / 2;

        // Charge in DAI if collateral in storage or token if collateral in vault
        v.reward1 = t.positionSizeDai > 0
            ? storageT.handleDevGovFees(t.pairIndex, v.levPosDai, true, false)
            : storageT.handleDevGovFees(t.pairIndex, v.levPosDai * PRECISION /
v.tokenPriceDai, false, false)
                * v.tokenPriceDai / PRECISION;
        t.initialPosToken -= v.reward1 * PRECISION / i.tokenPriceDai;
        storageT.updateTrade(t);
        ...
    }
}
```

contracts/TradingStorageV7.sol:L518-L535

```
function handleDevGovFees(uint256 _pairIndex, uint256 _leveragedPositionSize, bool _dai,
bool _fullFee)
    external
    onlyTrading
    returns (uint256 fee)
{
    fee = _leveragedPositionSize * priceAggregator.openFeeP(_pairIndex) / PRECISION /
100;
    if (!_fullFee) fee /= 2; // Fee is halved here

    if (_dai) {
        govFeesDai += fee;
        devFeesDai += fee;
    } else {
        govFeesToken += fee;
        devFeesToken += fee;
    }
    fee *= 2;
}
```

Since `_fullFee` is set to false when calling from `updateSlCallback`, the fee is already halved within this function.

The combination of dividing `v.levPosDai` by 2 and halving the fee in `handleDevGovFees` results in the fee being quartered instead of halved.

Recommendation

Remove the unnecessary division by 2 in the calculation of `v.levPosDai` within the `updateSlCallback` function.

Status

This issue has been resolved by the team with commit 5e6e6a3.

12. Incomplete property of NFT

Severity: Low	Category: Business Logic
Target: - contracts/tokens/VaultTokenLockedDepositNftDesign.sol	

Description

In the DepositNft contract, functions responsible for generating NFT metadata and images, such as `numberToRoundedString` and `generateBase64Image`, are currently incomplete. This results in NFTs with incorrect or incomplete properties, adversely affecting user experience by not displaying the correct information or visuals.

contracts/tokens/VaultTokenLockedDepositNftDesign.sol:L39-L99

```
// TODO: design will evolve
function generateBase64Image(
    uint256 tokenId,
    IVaultToken.LockedDeposit memory lockedDeposit,
    string memory vaultTokenSymbol,
    string memory assetSymbol,
    uint8 numberInputDecimals,
    uint8 numberOutputDecimals
) private pure returns (string memory) {
    ...
}
// Returns readable string of int part of number passed
// TODO: make it return the string with decimals = 'outputDecimals'
function numberToRoundedString(uint256 number, uint8 inputDecimals, uint8
outputDecimals)
    public
    pure
    returns (string memory)
{
    outputDecimals = 0; // silence warning
    return Strings.toString(number / (10 ** inputDecimals));
}
```

Recommendation

Complete the Implementation.

Status

This issue has been resolved by the team with commit 31de3a0.

13. Missing leverage checks

Severity: Low	Category: Data Validation

Target:
- contracts/TradingV7.sol
- contracts/TradingStorageV7.sol

Description

In the TradingV7 contract, the `openTrade` function allows users to open new trades by specifying various parameters, including the leverage. While there is validation to ensure that the leverage is within the minimum and maximum allowed for each trading pair, there is no validation to check whether the trader's requested leverage exceeds their individual `leverageUnlocked` level as defined in the TradingStorageV7 contract.

contracts/TradingV7.sol:L139-L230

```
function openTrade(
    StorageInterfaceV7.Trade memory t,
    StorageInterfaceV7.OpenLimitOrderType orderType, // LEGACY => market
    uint256 spreadReductionId,
    uint256 slippageP // for market orders only
) external notContract notDone {
    ...
    require(t.positionSizeDai * t.leverage >= pairsStored.pairMinLevPosDai(t.pairIndex),
"BELOW_MIN_POS");
    require(
        t.leverage > 0 && t.leverage >= pairsStored.pairMinLeverage(t.pairIndex)
            && t.leverage <= pairsStored.pairMaxLeverage(t.pairIndex),
        "LEVERAGE_INCORRECT"
    );
    ...
}
```

contracts/TradingStorageV7.sol:L64-L69

```
// Structs
struct Trader {
    uint256 leverageUnlocked;
    address referral;
    uint256 referralRewardsTotal; // 1e18
}
```

Traders may be able to open positions with leverage higher than what they have unlocked, potentially bypassing intended restrictions, which could lead to increased risk exposure for both the trader and the platform.

Recommendation

Implement Trader-Specific Leverage Checks.

Status

This issue has been acknowledged by the team.

14. Use safeTransfer()/safeTransferFrom() instead of transfer()/transferFrom()

Severity: Low	Category: Risky External Calls
Target: - contracts/TradingStorageV7.sol	

Description

contracts/TradingStorageV7.sol:L43,L544,L565,L567

```
dai.transfer(gov, govFeesDai);
dai.transfer(dev, devFeesDai);
dai.transfer(_to, _amount);
dai.transferFrom(_from, _to, _amount);
```

Tokens not compliant with the ERC20 specification could return false from the transfer function call to indicate the transfer fails, while the calling contract would not notice the failure if the return value is not checked. Checking the return value is a requirement, as written in the EIP-20 specification:

```
Callers MUST handle false from returns (bool success). Callers MUST NOT assume that
false is never returned!
```

Recommendation

Consider using the SafeERC20 library implementation from OpenZeppelin and call safeTransfer or safeTransferFrom when transferring ERC20 tokens.

Status

This issue has been resolved by the team with commit df57129.

15. Missing remove mechanism

Severity: Low	Category: Business logic
Target: - contracts/TradingStorageV7.sol	

Description

There is a function for adding (push in array) new support token. But there is no opportunity to remove one of the tokens from the array.

It could be helpful if a token was hacked or upgraded to a bad contract(by hack, for example).

contracts/TradingStorageV7.sol:L242-246

```
function addSupportedToken(address _token) external onlyGov {
    require(_token != address(0));
    supportedTokens.push(_token);
    emit SupportedTokenAdded(_token);
}
```

Recommendation

Add function for gov for removing the token from supportedTokens array.

Status

This issue has been resolved by the team with commit 77dc5ad.

16. Missing events for functions that change critical state

Severity: Low	Category: Logging
Target: - contracts/tokens/BTPVesting.sol	

Description

Events allow capturing the changed parameters so that off-chain tools/interfaces can register such changes that allow users to evaluate them. Missing events do not promote transparency and if such changes immediately affect users' perception of fairness or trustworthiness, they could exit the protocol causing a reduction in protocol users.

In the `BTPVesting`, events are lacking in the privileged setter functions (e.g. `setDelegateApproval`).

Recommendation

It is recommended to emit events for critical state changes.

Status

This issue has been resolved by the team with commit f511ae2.

2.3 Informational Findings

17. Missing two-step transfer ownership pattern

Severity: Informational	Category: Business logic
Target: - contracts/staking/Vault.sol	

Description

The `Vault` contract inherits from the Ownable contract. This contract does not implement a two-step process for transferring ownership. Thus, ownership of the contract can easily be lost when making a mistake in transferring ownership.

Recommendation

Consider using the Ownable2Step contract from OpenZeppelin instead.

Status

This issue has been resolved by the team with commit f511ae2.

18. Missing zero address checks

Severity: Informational	Category: Data Validation

Target:
- contracts/tokens/VaultTokenLockedDepositNftV7.sol
- contracts/PairInfosV7.sol
- contracts/PairsStorageV7.sol
- contracts/TradingStorageV7.sol

Description

It is considered a security best practice to verify addresses against the zero address during initialization or setting. However, this precautionary step is absent for address variables.

contracts/tokens/VaultTokenLockedDepositNftV7.sol:L20-L30

```
constructor(
) ERC721(name, symbol) {
    vaultToken = _vaultToken;
    design = _design;
    designDecimals = _designDecimals;
}
```

contracts/PairInfosV7.sol:L82-L84

```
constructor(StorageInterfaceV7 _storageT) {
    storageT = _storageT;
}
```

contracts/PairsStorageV7.sol:L79-L83

```
constructor(StorageInterfaceV7 _storageT, uint256 _currentOrderId) {
    require(_currentOrderId > 0, "ORDER_ID_0");
    storageT = _storageT;
    currentOrderId = _currentOrderId;
}
```

contracts/TradingStorageV7.sol:L171-L183

```
constructor(
) {
    gov = _gov;
    dev = _dev;
    executor = _executor;
    token = _token;
    dai = _dai;
}
```

Recommendation

Consider adding zero address checks for address variables.

Status

This issue has been resolved by the team with commit 94de154.

19. Gas optimization suggestions

Severity: Informational	Category: Gas Optimization

Target:
- contracts/PairInfosV7.soll
- contracts/TradingStorageV7.sol
- contracts/pricefeeds/PriceAggregatorV7
- contracts/pricefeeds/VaultTokenOpenPnlFeedV7.sol
- contracts/utils/Strings.sol

Description

1. Memory reading saves more gas than storage reading multiple times when the state is not changed. So caching the storage variables in memory and using the memory instead of storage reading is effective. So cache array length outside of the loop can save gas.

contracts/PairInfosV7.sol:L128,L15,L168,L189,L467

```
for (uint256 i = 0; i < indices.length; i++) {
```

contracts/TradingStorageV7.sol:L397

```
for (uint256 i = 0; i < orderIds.length; i++) {
```

contracts/pricefeeds/PriceAggregatorV7.sol:L107,L146

```
for (uint256 i = 0; i < oracles.length; i++) {
```

contracts/pricefeeds/VaultTokenOpenPnlFeedV7.sol:L110

```
for (uint256 i = 0; i < oracles.length; i++) {
```

contracts/pricefeeds/VaultTokenOpenPnlFeedV7.sol:L267

```
for (uint256 i; i < array.length; i++) {
```

contracts/utils/Strings.sol:L30

```
for (uint256 i = 0; i < data.length; i++) {
```

2.The `String` contract defined the hexadecimal character table twice.

contracts/utils/Strings.sol:L9, L24-L25

```
bytes16 private constant _SYMBOLS = "0123456789abcdef";
    ...
function toString(bytes memory data) public pure returns (string memory) {
    bytes memory alphabet = "0123456789abcdef";
```

3.Solidity 0.8.22 introduces an overflow check optimization that automatically generates an unchecked arithmetic increment of the counter of for loops.

contracts/utils/Strings.sol:L47-L74

```
constructor(
     ..
    for (uint256 i; i < loopEnd; ) {
        ...
        unchecked {
           ++i;
```

```
        }
```

Recommendation

Consider using the above suggestions to save gas.

Status

This issue has been resolved by the team with commit 72cd606.

20. Ineffective check

Severity: Informational	Category: Gas Optimization

Target:
- contracts/dao/EmissionVoting.sol
- contracts/dao/ListaVault.sol
- contracts/dao/ClisBNBLaunchPoolDistributor.sol

Description

1. The `posDai` is always 0, and it is meaningless to judge that it is more than 0.

contracts/TradingCallbacksV7.sol:L529

```
Values memory v;
v.levPosDai = trade.positionSizeDai * trade.leverage;
v.tokenPriceDai = aggregator.tokenPriceDai();

// 2. Charge opening fee - referral fee (if applicable)
v.reward2 = storageT.handleDevGovFees(trade.pairIndex, (v.posDai > 0 ? v.posDai :
v.levPosDai), true, true);
```

2. The Zero address can't be `MINTER_ROLE`.

contracts/TradingStorageV7.sol:L299-234

```
function addTradingContract(address _trading) external onlyGov {
    require(token.hasRole(MINTER_ROLE, _trading), "NOT_MINTER");
    require(_trading != address(0));
    isTradingContract[_trading] = true;
    emit TradingContractAdded(_trading);
}
```

3. The `getOpenLimitOrder` has checked if `LimitOrder` exists. There are duplicate checks in the following places.

contracts/TradingStorageV7.sol:L619-626

```
function getOpenLimitOrder(address _trader, uint256 _pairIndex, uint256 _index)
        external
        view
        returns (OpenLimitOrder memory)
    {
        require(hasOpenLimitOrder(_trader, _pairIndex, _index), "NO_OPEN_LIMIT_ORDER");
        return openLimitOrders[openLimitOrderIds[_trader][_pairIndex][_index]];
    }
```

contracts/TradingV7.sol:L269-280

```
function updateOpenLimitOrder(
    ) external notContract notDone {
    address sender = _msgSender();

    require(storageT.hasOpenLimitOrder(sender, pairIndex, index), "NO_LIMIT");

    StorageInterfaceV7.OpenLimitOrder memory o = storageT.getOpenLimitOrder(sender,
pairIndex, index);
```

contracts/TradingV7.sol:L299-L304

```
function cancelOpenLimitOrder(uint256 pairIndex, uint256 index) external notContract
notDone {
    address sender = _msgSender();

    require(storageT.hasOpenLimitOrder(sender, pairIndex, index), "NO_LIMIT");

    StorageInterfaceV7.OpenLimitOrder memory o = storageT.getOpenLimitOrder(sender,
pairIndex, index);
```

contracts/TradingV7.sol:L365-L418

```
 function executeExecutorOrder(
    ) external notContract notDone onlyExecutor {
    ...
    StorageInterfaceV7.Trade memory t;
    if (orderType == StorageInterfaceV7.LimitOrder.OPEN) {
        require(storageT.hasOpenLimitOrder(trader, pairIndex, index), "NO_LIMIT");
    }
    ...
    if (orderType == StorageInterfaceV7.LimitOrder.OPEN) {
        StorageInterfaceV7.OpenLimitOrder memory l = storageT.getOpenLimitOrder(trader,
pairIndex, index);
```

4. The `updateTp`/`updateSl` has checked if `leverage` is more than zero. There are duplicate checks in the following places.

contracts/TradingV7.sol:L313-L333

```
function updateTp(uint256 pairIndex, uint256 index, uint256 newTp) external notContract
notDone {
    ...
    require(t.leverage > 0, "NO_TRADE");

    storageT.updateTp(sender, pairIndex, index, newTp);
    ...
    }

function updateSl(uint256 pairIndex, uint256 index, uint256 newSl) external notContract
notDone {
     ...
    require(t.leverage > 0, "NO_TRADE");
```

contracts/TradingStorageV7.sol:L469-L483

```
function updateSl(address _trader, uint256 _pairIndex, uint256 _index, uint256 _newSl)
external onlyTrading {
    ...
    if (t.leverage == 0) return;
    ...
}

    function updateTp(address _trader, uint256 _pairIndex, uint256 _index, uint256
_newTp) external onlyTrading {
    ...
    if (t.leverage == 0) return;
    ...
}
```

Recommendation

Remove the Ineffective check.

Status

This issue has been resolved by the team with commit 9bbda1f.

21. Custom error don’t contain revert reason

Severity: Informational	Category: Code Quality
Target: - contracts/TradingV7.sol	

Description

When a delegatecall to contractAddress has failed, the user could not understand the reason for the revert, because event `CouldNotCloseTrade()` doesn't have a bytes result.

contracts/TradingV7.sol:L457-L480

```
function closeTradeMarketTimeout(uint256 _order) external notContract notDone {
    ...
    (bool success,) = address(this).delegatecall(
        abi.encodeWithSignature("closeTradeMarket(uint256,uint256)", t.pairIndex,
t.index)
    );

    if (!success) {
       emit CouldNotCloseTrade(sender, t.pairIndex, t.index);
    }

    emit AggregatorCallbackTimeout(_order, o);
}
```

Recommendation

Consider Returning the call result.

```
(bool, succes, bytes memory returnData) = address(this).delegatecall(
    abi.encodeWithSignature("closeTradeMarket(uint256,uint256)", t.pairIndex, t.index)
);
if (!success) {
    emit CouldNotCloseTrade(sender, t.pairIndex, t.index, returnData;
}
```

Status

This issue has been resolved by the team with commit 9d52ba5.

Appendix

Appendix 1 - Files in Scope

This audit covered the following files in commit 32ba60f:

File	SHA-1 hash
contracts/TradingV7.sol	65791569906326aaf6d1d43f52e8b0e5c73b40ec
contracts/TradingCallbacksV7.sol	a272f7560be837b690e02daec93c3d1efc74b23e
contracts/PairsStorageV7.sol	99ca389cf995ee5a70096100086150d005b04ee7
contracts/TradingStorageV7.sol	f69db66d39aab9faf2e668839bdc76d120726bf4
contracts/PairInfosV7.sol	3fbd24e0812a784bbb3699970f367ccd66868a97
contracts/staking/Vault.sol	98dfce3db2479559cc35f1aaf42004d8433ed15d
contracts/utils/Delegatable.sol	0c8f2927d5af2b42ea4b9d6c207b02e8b1745b3d
contracts/utils/TradeUtils.sol	288afbbb8864d3d293fb6dacf7a25692156efa2d
contracts/utils/Strings.sol	83a818553cde103f08eee225e67f26343908bcef
contracts/pricefeeds/VaultTokenOpenPnlFeedV7.sol	14e68e1d9656ea4c0dd1e6f6bd39a11d8d492fab
contracts/pricefeeds/TWAPPriceGetter.sol	557201a328c2af831dc22fe205278778da78736e
contracts/pricefeeds/PriceAggregatorV7.sol	f71cb5bf30b2479ac7d4b001ead91690fc552b99
contracts/tokens/VaultTokenLockedDepositNftV7.sol	da61d79f3df6074e39b58c7b2906439714abaf41
contracts/tokens/DaiTokenV7.sol	da2caf7a202445e9661ee4f8008e54f043ce1df4
contracts/tokens/BTPVesting.sol	f2ad0df3ff72047319e6b3e156a4563ff986959d
contracts/tokens/BTP.sol	b3f1f9242ff26139680915d04be90f599945b92d
contracts/tokens/VaultTokenLockedDepositNftDesign.sol	c57fb6874155c7f6c274847184a334f8ee46245e

And we audited the files in commit 80f2b56 that introduced new features:

File	SHA-1 hash
contracts/TradingV7.sol	59b61d71248e604530d8538a0ef79a346dbc4c30
contracts/TradingCallbacksV7.sol	5033b5f3790ae7d0cd2126837b5ccdbcfb8c4db1
contracts/PairsStorageV7.sol	976b24f1be7739aa531d1d927b51943a6e9b8e40
contracts/TradingStorageV7.sol	b625c699ca2aede2265c4f9b89d3e19474dc36ee

contracts/PairInfosV7.sol	db189651a55794fb26f7d78ff54f3cee1e2e8318
contracts/staking/Vault.sol	8b9b42f51a297fcb477077ff1178d252e5892dfa
contracts/utils/Delegatable.sol	55468426efc55c54f9d4e1fa5ab6475c2a8b6124
contracts/utils/TradeUtils.sol	288afbbb8864d3d293fb6dacf7a25692156efa2d
contracts/utils/Strings.sol	af9d579b933d689830344325bb7186835ec25bb5
contracts/pricefeeds/VaultTokenOpenPnlFeedV7.sol	e7296687aae217f647436b9c8000760430c2b4a7
contracts/pricefeeds/TWAPPriceGetter.sol	557201a328c2af831dc22fe205278778da78736e
contracts/pricefeeds/PriceAggregatorV7.sol	a55616d7775efd44b69476f576504c095ec54633
contracts/tokens/VaultTokenLockedDepositNftV7.sol	9535bb400d476c6aa7baedd2871dd415a50fa22c
contracts/tokens/DaiTokenV7.sol	44214f5a53bbef6c1ba90639715f60449921f270
contracts/tokens/BTPVesting.sol	047851c3fc6603192a55eb72e116509aeb7169f5
contracts/tokens/BTP.sol	b3f1f9242ff26139680915d04be90f599945b92d
contracts/tokens/VaultTokenLockedDepositNftDesign.sol	28747781634ac98062aba3b379610c89af088f30