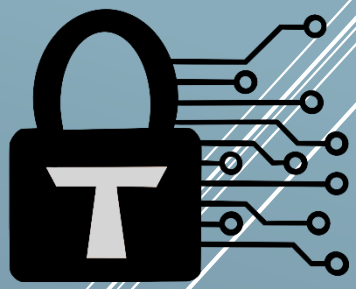
Trust Security

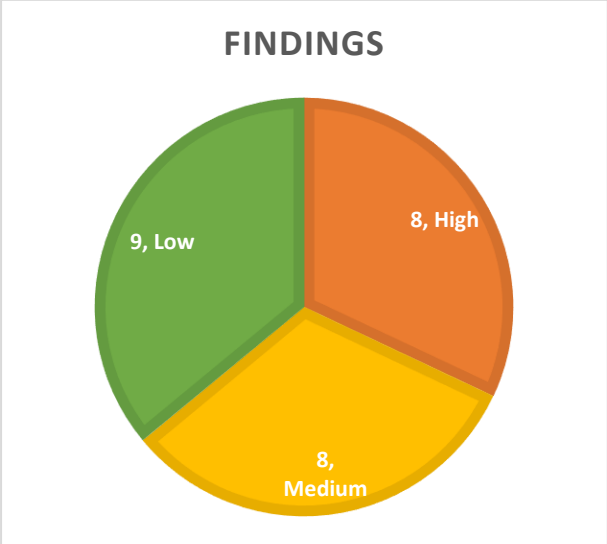


Smart Contract Audit

Stella

29/05/23

Executive summary



Category	Lending
Audited file count	33
Lines of Code	3765
Auditor	cccz Jeiwan
Time period	11-30/05/23

Findings

Severity	Total	Fixed	Acknowledged
High	8	7	1
Medium	8	8	0
Low	9	5	4

Centralization score

Signature

Digitally signed by Trust
Date: 2023-06-18 12:04:59

Document properties

Versioning

Version	Date	Description
0.1	29/05/2023	Client report
0.2	05/06/2023	Mitigation review
0.3	18/06/2023	Minor changes

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

- contracts/stella-lending/common/FreezeBuckets.sol
- contracts/stella-lending/common/LendingGateway.sol
- contracts/stella-lending/common/RewardVault.sol
- contracts/stella-lending/common/RiskFramework.sol
- contracts/stella-lending/lending-pools/BaseLendingPool.sol
- contracts/stella-lending/lending-pools/Erc20LendingPool.sol
- contracts/stella-lending/lending-pools/NativeLendingPool.sol
- contracts/stella-lending/LendingProxy.sol
- contracts/stella-libraries/AccessController.sol
- contracts/stella-libraries/BytesLib.sol
- contracts/stella-libraries/TickMath.sol
- contracts/stella-libraries/TransparentUpgradeableProxyImpl.sol
- contracts/stella-libraries/TransparentUpgradeableProxyReceiveETH.sol
- contracts/stella-libraries/UniswapV3Lib.sol
- contracts/stella-libraries/UsingAccessController.sol
- contracts/stella-libraries/UsingAccessControllerUpgradeable.sol
- contracts/stella-oracles/AggregatorOracle.sol
- contracts/stella-oracles/BandAdapterOracle.sol
- contracts/stella-oracles/ChainlinkAdapterOracle.sol
- contracts/stella-oracles/UniswapV3Oracle.sol
- contracts/stella-oracles/UsingBaseOracle.sol
- contracts/stella-strategies/common/Config.sol
- contracts/stella-strategies/common/LiquidationVault.sol
- contracts/stella-strategies/common/ProfitSharingModel.sol
- contracts/stella-strategies/common/StrategyGateway.sol
- contracts/stella-strategies/factory/UniswapV3StrategyFactory.sol
- contracts/stella-strategies/position-managers/base/BasePositionManager.sol
- contracts/stella-strategies/position-managers/base/BasePositionViewer.sol
- contracts/stella-strategies/position-managers/uniswap-v3/UniswapV3PositionManager.sol
- contracts/stella-strategies/position-managers/uniswap-v3/UniswapV3PositionViewer.sol
- contracts/stella-strategies/strategies/base/BaseStrategy.sol
- contracts/stella-strategies/strategies/uniswap-v3/UniswapV3Strategy.sol
- contracts/stella-strategies/strategies/SwapHelper.sol

Repository details

- **Repository URL:** https://github.com/AlphaFinanceLab/stella-arbitrum-private-contract
- **Commit hash:** 3a4e99307e9cbf790279e49a4d90771e5486c51d
- **Mitigation review hash:** ac5ce4609d57d7d62400a09c6ee9f2fb8d8b6b59

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Trust is the leading auditor at competitive auditing service Code4rena, reported several critical issues to Immunefi bug bounty platform and is currently a Code4rena judge.

About the Auditors

A top competitor in audit contests, cccz has achieved superstar status in the security space. He is a Black Hat / DEFCON speaker with rich experience in both traditional and blockchain security.

After spending many years working as a web developer and studying blockchain technologies in his free time, Jeiwan started his full-time smart contracts security journey in September 2022. Since then, he has participated in more than 50 auditing contests on Code4rena and Sherlock, where he took multiple Top 5 places competing with the best auditors in the field. Jeiwan is the author of Uniswap V3 Development Book. Thanks to his deep knowledge of Uniswap, Jeiwan specializes in projects that integrate or extend Uniswap, as well as any other AMM.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	**Excellent**	Project kept code as simple as possible, reducing attack risks
Documentation	**Good**	Project is mostly very well documented.
Best practices	**Good**	Project mostly follows best practices.
Centralization risks	**Moderate**	Project has some centralization concerns.

Findings

High severity findings

TRST-H-1 Incorrect implementation of getProfitSharingE18() greatly reduces Lender's yield

- **Category:** Logical flaws
- **Source:** ProfitSharingModel.sol
- **Status:** Fixed

Description

ProfitSharingModel.getProfitSharingE18() calculates the share of profit that Lender gets based on the APR of the position. According to the formula, the higher the APR, the lower the share of profit the Lender gets, but due to the wrong implementation of the *getProfitSharingE18()* function, if the APR is smaller than **MAX_ANNUALIZED_YEILD**, the base share of 25% is returned, actually 25% should be returned when the APR is larger than **MAX_ANNUALIZED_YEILD**.

Considering an APR of 5%, Lender's share of the profit should be 77%, while *getProfitSharingE18()* returns 25%, which greatly reduces Lender's share of the profit.

Recommended mitigation

Modify *getProfitSharingE18()* as follows

```
- if (_annualizedYieldE18 < MAX_ANNUALIZED_YEILD) {
+  if (_annualizedYieldE18 >= MAX_ANNUALIZED_YEILD) {
    return 0.25e18;
}
```

Team response

Fixed

Mitigation Review

The team has fixed it as recommended to make the logic correct.

TRST-H-2 On liquidation, if netPnLE36 <= 0, the premium paid by the liquidator is locked in the contract.

- **Category:** Logical flaws
- **Source:** BaseStrategy.sol
- **Status:** Fixed

Description

When liquidating a position, the liquidator is required to pay premium to Lender, which is accumulated in **sharingProfitTokenAmts** together with Lender's profit and paid to Lender in *_shareProfitsAndRepayAllDebts()*.

```
    (
      netPnLE36,
      lenderProfitUSDValueE36,
      borrowTotalUSDValueE36,
      positionOpenUSDValueE36,
      sharingProfitTokenAmts
    ) = calcProfitInfo(_positionManager, _user, _posId);

    // 2. add liquidation premium to the shared profit amounts
    uint lenderLiquidatationPremiumBPS = IConfig(config).lenderLiquidatePremiumBPS();
    for (uint i; i < sharingProfitTokenAmts.length; ) {
      sharingProfitTokenAmts[i] +=
        (pos.openTokenInfos[i].borrowAmt * lenderLiquidatationPremiumBPS) /
        BPS;
      unchecked {
        ++i;
      }
    }
```

However, if **netPnLE36 <= 0**, *_shareProfitsAndRepayAllDebts()* will not pay any profit to Lender and the premium in **sharingProfitTokenAmts** will also not be paid to Lender, which means that the premium paid by the liquidator will be locked in the contract.

```
  function _shareProfitsAndRepayAllDebts(
    address _positionManager,
    address _posOwner,
    uint _posId,
    int _netPnLE36,
    uint[] memory _shareProfitAmts,
    address[] memory _tokens,
    OpenTokenInfo[] memory _openTokenInfos
  ) internal {
    // 0. load states
    address _lendingProxy = lendingProxy;

    // 1. if net pnl is positive, share profits to lending proxy
    if (_netPnLE36 > 0) {
      for (uint i; i < _shareProfitAmts.length; ) {
        if (_shareProfitAmts[i] > 0) {
          ILendingProxy(_lendingProxy).shareProfit(_tokens[i], _shareProfitAmts[i]);
        }
        unchecked {
          ++i;
        }
      }

      emit ProfitShared(_posOwner, _posId, _tokens, _shareProfitAmts);
    }
```

Also, when the position is closed, the tokens in the contract will be sent to the caller, so the next person who closes the position will get the locked tokens.

```
    underlyingAmts = new uint[](underlyingTokens.length);
    for (uint i; i < underlyingTokens.length; ) {
      underlyingAmts[i] = IERC20(underlyingTokens[i]).balanceOf(address(this));
      if (underlyingAmts[i] < _params.minUnderlyingAmts[i]) {
        revert TokenAmountLessThanExpected(
          underlyingTokens[i],
          underlyingAmts[i],
          _params.minUnderlyingAmts[i]
        );
      }
      _doRefund(underlyingTokens[i], underlyingAmts[i]);

      unchecked {
        ++i;
      }
```

Recommended mitigation

Modify *shareProfitsAndRepayAllDebts*() as follows

```
  function _shareProfitsAndRepayAllDebts(
    address _positionManager,
    address _posOwner,
    uint _posId,
    int _netPnLE36,
    uint[] memory _shareProfitAmts,
    address[] memory _tokens,
    OpenTokenInfo[] memory _openTokenInfos
  ) internal {
    // 0. load states
    address _lendingProxy = lendingProxy;

    // 1. if net pnl is positive, share profits to lending proxy
-   if (_netPnLE36 > 0) {
      for (uint i; i < _shareProfitAmts.length; ) {
        if (_shareProfitAmts[i] > 0) {
          ILendingProxy(_lendingProxy).shareProfit(_tokens[i], _shareProfitAmts[i]);
        }
        unchecked {
          ++i;
        }
      }

      emit ProfitShared(_posOwner, _posId, _tokens, _shareProfitAmts);
-   }
```

Team response

Fixed

Mitigation Review

The team has fixed it as recommended to make the logic correct.

TRST-H-3 The liquidated person can make the liquidator lose premium by adding collateral in advance

- **Category:** MEV attacks
- **Source:** BaseStrategy.sol
- **Status:** Fixed

Description

When the position with **debtRatioE18 >= 1e18** or **startLiqTimestamp ! = 0**, the position can be liquidated. On liquidation, the liquidator needs to pay premium, but the profit is related to the position's health factor and **deltaTime**, and when **discount == 0**, the liquidator loses premium.

```
uint deltaTime;

// 1.1 check the amount of time since position is marked
if (pos.startLiqTimestamp > 0) {
  deltaTime = Math.max(deltaTime, block.timestamp - pos.startLiqTimestamp);
}
// 1.2 check the amount of time since position is past the deadline
if (block.timestamp > pos.positionDeadline) {
  deltaTime = Math.max(deltaTime, block.timestamp - pos.positionDeadline);
}

// 1.3 cap time-based discount, as configured
uint timeDiscountMultiplierE18 = Math.max(
  IConfig(config).minLiquidateTimeDiscountMultiplierE18(),
  ONE_E18 - deltaTime * IConfig(config).liquidateTimeDiscountGrowthRateE18()
);

// 2. calculate health-based discount factor
uint curHealthFactorE18 = (ONE_E18 * ONE_E18) /
  getPositionDebtRatioE18(_positionManager, _user, _posId);
uint minDesiredHealthFactorE18 = IConfig(config).minDesiredHealthFactorE18s(strategy);

// 2.1 interpolate linear health discount factor (according to the diagram in documentation)
uint healthDiscountMultiplierE18 = ONE_E18;
if (curHealthFactorE18 < ONE_E18) {
  healthDiscountMultiplierE18 = curHealthFactorE18 > minDesiredHealthFactorE18
    ? ((curHealthFactorE18 - minDesiredHealthFactorE18) * ONE_E18) /
      (ONE_E18 - minDesiredHealthFactorE18)
    : 0;
}

// 3. final liquidation discount = apply the two discount methods together
liquidationDiscountMultiplierE18 =
  (timeDiscountMultiplierE18 * healthDiscountMultiplierE18) /
  ONE_E18;
```

Consider the following scenario.

1. Alice notices Bob's position with **debtRatioE18 >= 1e18** and calls *liquidatePosition()* to liquidate.

2. Bob observes Alice's transaction, frontruns a call *markLiquidationStatus()* to make **startLiqTimestamp == block.timestamp**, and calls *adjustExtraColls()* to bring the position back to the health state.

3. Alice's transaction is executed, and since the **startLiqTimestamp** of Bob's **position.startLiqTimestamp ! = 0**, it can be liquidated, but since **discount = 0**, Alice loses premium.

This breaks the protocol's liquidation mechanism and causes the liquidator not to launch liquidation for fear of losing assets, which will lead to more bad debts

Recommended mitigation

Consider having the liquidated person bear the premium, or at least have the liquidator use the **minDiscount** parameter to set the minimum acceptable discount.

Team response

Liquidator contracts can easily require the min amount in their own logic to ensure profitability anyways.

Add **maxPayAmount** parameter as slippage control in the *liquidate()* and if **requiredPayAmount** exceeds the value, just revert.

Mitigation Review

The fix makes liquidators able to use the **maxPayAmount** parameter to prevent compromise in liquidation.

In addition, after discussing with the team, there are some external conditions / measures that the team could make, which could lead to a lower severity, but the assigned severity is based on the worst-case scenario.

TRST-H-4 First depositor can steal asset tokens of others

- **Category:** front-running
- **Source:** BaseLendingPool.sol
- **Status:** Fixed

Description

The first depositor can be front run by an attacker and as a result will lose a considerable part of the assets provided.

When the pool has no share supply, in *_mintInternal*(), the amount of shares to be minted is equal to the assets provided. An attacker can abuse of this situation and profit of the rounding down operation when calculating the amount of shares if the supply is non-zero.

```
function _mintInternal(
```

```
    address _receiver,
    uint _balanceIncreased,
    uint _totalAsset
  ) internal returns (uint mintShares) {
    unfreezeTime[_receiver] = block.timestamp + mintFreezeInterval;

    if (freezeBuckets.interval > 0) {
      FreezeBuckets.addToFreezeBuckets(freezeBuckets, _balanceIncreased.toUint96());
    }
    uint _totalSupply = totalSupply();
    if (_totalAsset == 0 || _totalSupply == 0) {
      mintShares = _balanceIncreased + _totalAsset;
    } else {
      mintShares = (_balanceIncreased * _totalSupply) / _totalAsset;
    }

    if (mintShares == 0) {
      revert ZeroAmount();
    }

    _mint(_receiver, mintShares);
  }
```

Consider the following scenario.

1. Alice wants to deposit 2M * 1e6 USDC to a pool.

2. Bob observes Alice's transaction, frontruns to deposit 1 wei USDC to mint 1 wei share, and transfers 1 M * 1e6 USDC to the pool.

3. Alice's transaction is executed, since **_totalAsset = 1M * 1e6 + 1** and **totalSupply = 1**, Alice receives 2M * 1e6 * 1 / (1M * 1e6 + 1) = 1 share.

4. The pool now has 3M*1e6 +1 assets and distributed 2 shares.

Bob profits 0.5 M and Alice loses 0.5 M USDC.

Recommended mitigation

When **_totalSupply == 0**, send the first min liquidity LP tokens to the zero address to enable share dilution.

Another option is to use the ERC4626 implementation from OZ.

Team response

Fixed

Mitigation Review

The team increased the interest-bearing token decimal by 18 to prevent attackers from manipulating the share price by precision loss, and made themself the first depositor to prevent potential attacks.

TRST-H-5 The attacker can use larger dust when opening a position to perform griefing attacks

- **Category:** Griefing attacks
- **Source:** BaseStrategy.sol
- **Status:** Acknowledged

Description

When opening a position, unused assets are sent to **dustVault** as dust, but since these dust are not subtracted from **inputAmt**, they are included in the calculation of **positionOpenUSDValueE36**, resulting in a small **netPnLE36**, which can be used by an attacker to perform a griefing attack.

```
    uint inputTotalUSDValueE36;
    for (uint i; i < openTokenInfos.length; ) {
      inputTotalUSDValueE36 += openTokenInfos[i].inputAmt * tokenPriceE36s[i];
      borrowTotalUSDValueE36 += openTokenInfos[i].borrowAmt * tokenPriceE36s[i];

      unchecked {
        ++i;
      }
    }

    // 1.3 calculate net pnl (including strategy users & borrow profit)
    positionOpenUSDValueE36 = inputTotalUSDValueE36 + borrowTotalUSDValueE36;
    netPnLE36 = positionCurUSDValueE36.toInt256() - positionOpenUSDValueE36.toInt256();
```

Consider ETH:USDC = 1:1000, **posMinLpSlippageMultiplierE18s = 0.95e18**

1. Alice opens a position with 2.5 ETH and 2000 USDC, borrows 3 ETH and 3000 USDC, and then dust = 0.5 ETH is sent to **dustVault**. The value of the LP position is actually 10000 USD, since **lpUSDValueE36(10000) > minLpUSDValueE36(10500*0.95 = 9975)**, it can pass the LP value validation.

```
    minLpUSDValueE36 =
      ((inputUSDValueE36 + borrowUSDValueE36) *
        IConfig(_config).posMinLpSlippageMultiplierE18s(strategy)) /
      ONE_E18;

    // 4. get min & max borrow value cap
    (minBorrowUSDValueE18, maxBorrowUSDValueE18) =
IConfig(_config).getMinMaxCapBorrowUSDValueE18(
      strategy
    );
  }
  return
    lpUSDValueE36 >= minLpUSDValueE36 &&
```

2. After a while, the LP position is raised to 8500 USD. Alice closes the position. In **calcProfitInfo**, the calculated **positionOpenUSDValueE36 = 10500 USD** (since the value of Dust is taken into account) and **netProfit = 10500 - 10500 = 0.**

This means that Alice uses Lender's profit as dust, Lender loses their profit.

Recommended mitigation

Consider subtracting dust from **inputAmt** when opening a position.

Team response

Acknowledged, the attacker is not profitable, where the dust vault can later be used to distribute to lenders afterwards if needs be.

Mitigation Review

It would be a good practice to distribute the dust to Lender, which can prevent Lender from being compromised by Griefing attacks.

TRST-H-6 An attacker can increase liquidity to the position's UniswapNFT to prevent the position from being closed

- **Category:** Logical flaws
- **Source:** UniswapV3Strategy.sol
- **Status:** Fixed

Description

UniswapV3NPM allows the user to increase liquidity to any NFT.

```
  function increaseLiquidity(IncreaseLiquidityParams calldata params)
    external
    payable
    override
    checkDeadline(params.deadline)
    returns (
      uint128 liquidity,
      uint256 amount0,
      uint256 amount1
    )
  {
    Position storage position = _positions[params.tokenId];

    PoolAddress.PoolKey memory poolKey = _poolIdToPoolKey[position.poolId];

    IUniswapV3Pool pool;
    (liquidity, amount0, amount1, pool) = addLiquidity(
```

When closing a position, in *_redeemPosition()*, only the initial liquidity of the NFT will be decreased, and then the NFT will be burned.

```
 function _redeemPosition(
  address _user,
  uint _posId
 ) internal override returns (address[] memory rewardTokens, uint[] memory rewardAmts) {
  address _positionManager = positionManager;
  uint128 collAmt = IUniswapV3PositionManager(_positionManager).getPositionCollAmt(_user,
_posId);
  // 1. take lp & extra coll tokens from lending proxy
```

```
    _takeAllCollTokens(_positionManager, _user, _posId, address(this));

    UniV3ExtraPosInfo memory extraPosInfo = IUniswapV3PositionManager(_positionManager)
      .getDecodedExtraPosInfo(_user, _posId);

    address _uniswapV3NPM = uniswapV3NPM; // gas saving
    // 2. remove underlying tokens from lp (internal remove in NPM)
    IUniswapV3NPM(_uniswapV3NPM).decreaseLiquidity(
      IUniswapV3NPM.DecreaseLiquidityParams({
        tokenId: extraPosInfo.uniV3PositionId,
        liquidity: collAmt,
        amount0Min: 0,
        amount1Min: 0,
        deadline: block.timestamp
      })
    );
    ...
    // 4. burn LP position
    IUniswapV3NPM(_uniswapV3NPM).burn(extraPosInfo.uniV3PositionId);
  }
```

If the liquidity of the NFT is not 0, burning will fail.

```
    function burn(uint256 tokenId) external payable override isAuthorizedForToken(tokenId) {
        Position storage position = _positions[tokenId];
        require(position.liquidity == 0 && position.tokensOwed0 == 0 && position.tokensOwed1 == 0,
'Not cleared');
        delete _positions[tokenId];
        _burn(tokenId);
    }
```

This allows an attacker to add 1 wei liquidity to the position's NFT to prevent the position from being closed, and later when the position expires, the attacker can liquidate it.

Recommended mitigation

Consider decreasing the actual liquidity(using uniswapV3NPM.positions to get it) of the NFT in *_redeemPosition()*, instead of the initial liquidity

Team response

Fixed

Mitigation Review

The team addressed this issue by decreasing NFT's latest liquidity in *_redeemPosition().*

TRST-H-7 Pending position fees miscalculation may result in increased PnL

- **Category:** Logical flaw
- **Source:** UniswapV3PositionViewer.sol
- **Status:** Fixed

Description

When calculating pending liquidity position fees, **liquidity**, **tokensOwed0**, and **tokensOwed1** are read from a Uniswap V3 pool using a position belonging to the NonfungiblePositionManager contract. However, the read values will also include the liquidity and the owed token amounts of all Uniswap V3 users who deposited funds in the price range of the position via the NonfungiblePositionManager contract. Since NonfungiblePositionManager manages positions in pools on behalf of users, the positions will hold liquidity of all NonfungiblePositionManager users. As a result, the PnL of UniswapV3Strategy positions may be significantly increased, resulting in increased payouts to lenders and loss of funds to borrowers/liquidators.

Recommended mitigation

Consider reading the values of **liquidity**, **tokensOwed0**, and **tokensOwed1** from the *IUniswapV3NPM(uniV3NPM).positions()* call on line 95. The call returns values specifically for the position identified by the token ID.

Team response

Fixed

Mitigation Review

The team has fixed it as recommended to make the logic correct.

TRST-H-8 "Exact output" swaps cannot be executed, blocking repayment of debt

- **Category:** logical flaw
- **Source:** SwapHelper.sol
- **Status:** Fixed

Description

When performing "exact output" swaps via Uniswap V2 and V3, the maximum input amount argument (**amountInMax** when calling Uniswap V2's *swapTokensForExactTokens()*, **amountInMaximum** when calling V3's *exactOutput()*) is set to 0. As a result, swapping attempts will always revert because no more than 0 input tokens can be sold (the slippage check in the Uniswap contracts will always revert because the swaps will require more input tokens).

We consider it high-severity because an "exact output" swap is mandatory when closing a position that doesn't have enough tokens to repay the borrowed amount. Thus, since "exact output" swaps are not possible, closing some positions won't be possible as well, leaving funds locked in the contract.

Recommended mitigation

Taking into account that the protocol implements delayed slippage checks, consider setting the maximum input amount arguments to **type(uint256).max**.

Team response

Fixed

Mitigation Review

The team has fixed it as recommended to make the logic correct.

Medium severity findings

TRST-M-1 markLiquidationStatus() may cause the liquidator to lose premium

- **Category:** MEV attacks
- **Source:** BasePositionManager.sol
- **Status:** Fixed

Description

When the **debtRatioE18** of a position is greater than 1 and less than 1.03 (unmark), the liquidator can call *markLiquidationStatus()* to accumulate **timeDiscountMultiplierE18** by making **pos.startLiqTimestamp == block.timestamp**. The liquidated person can also call *markLiquidationStatus()* to reset **pos.startLiqTimestamp** to clear **timeDiscountMultiplierE18**, which results in that when the **debtRatioE18** of a position hovers between 1.0 and 1.03, the liquidated person can front run the liquidator to make the liquidator lose premium.

Consider the following scenarios:

1. Alice's position has **debtRatioE18** greater than 1.0 and less than 1.03 (unmark).

2. Bob calls *markLiquidationStatus()* to initialize the **startLiqTimestamp** of Alice's position, and after some time, **timeDiscountMultiplierE18** accumulates to 50%, Bob calls *liquidatePosition()* to liquidate Alice's position.

3. Alice observes Bob's transaction and frontruns a call to *markLiquidationStatus()* to reset **startLiqTimestamp**, **timeDiscountMultiplierE18** is also reset to 0.

4. Bob's transaction is executed and the premium paid by Bob may be less than the profit received, in the extreme case, if **debtRatioE18 = 1e18**, Bob will not have any profit and will pay premium.

Recommended mitigation

Consider allowing *markLiquidationStatus()* to set the **startLiqTimestamp** only when **debtRatioE18 >= 1.03**(unmark), or allowing the liquidator to set the minimum acceptable discount.

```
-   if (debtRatioE18 >= ONE_E18 && startLiqTimestamp == 0) {
+   if (debtRatioE18 >= IBasePositionViewer(_positionViewer).unmarkLiqDebtRatioE18() &&
startLiqTimestamp == 0) {
      // mark liquidatable if the position is unhealthy and is not marked yet
      pos.startLiqTimestamp = block.timestamp.toUint32();
    } else if (
```

```
    startLiqTimestamp != 0 &&
    debtRatioE18 < IBasePositionViewer(_positionViewer).unmarkLiqDebtRatioE18()
   ) {
    // unmark liquidatable if the position is already marked and debt ratio falls below "unmark debt
ratio"
    pos.startLiqTimestamp = 0;
   } else {
    // revert otherwise
    revert MarkLiquidationStatusFailed();
   }
```

Team response

Fixed

Mitigation Review

The team addressed this issue by changing the unmark value to ~0.97.

TRST-M-2 SwapHelper.getCalldata should check whitelistedRouters[_router]

- **Category:** Validation issues
- **Source:** SwapHelper.sol
- **Status:** Fixed

Description

SwapHelper.getCalldata() returns data for swap based on the input, and uses **whitelistedRouters** to limit the **_router** param. The issue here is that when *setWhitelistedRouters()* sets the **_routers** state to **false**, it does not reset the data in **routerTypes** and **swapInfos**, which results in the router still being available in *getCalldata().* As a result, users can still swap with invalid router data.

```
   for (uint i; i < _statuses.length; ) {
    whitelistedRouters[_routers[i]] = _statuses[i];
    if (_statuses[i]) {
     routerTypes[_routers[i]] = _types[i];
     emit SetRouterType(_routers[i], _types[i]);
    }
    emit SetWhitelistedRouter(_routers[i], _statuses[i]);

    unchecked {
     ++i;
    }
   }
```

Recommended mitigation

Consider checking **whitelistedRouters[_router]** in *SwapHelper.getCalldata()*

Team response

Fixed.

Mitigation Review

The team addressed this issue by setting **routerTypes** to **UNSET** status in whitelist function when delisting.

TRST-M-3 No check for active Arbitrum Sequencer in Chainlink Oracle

- **Category:** Oracle integration issues
- **Source:** ChainlinkAdapterOracle.sol
- **Status:** Fixed

Description

If the Arbitrum sequencer were to go offline the Chainlink oracle may return an invalid/stale price. It should always be checked before consuming any data from Chainlink.

The Chainlink docs on L2 Sequencer Uptime Feeds specify more details.

Recommended mitigation

Check sequencer uptime before consuming any price data.

Team response

Fixed

Mitigation Review

The team addressed this issue by checking sequencer uptime before consuming any price data.

TRST-M-4 The swap when closing a position does not consider shareProfitAmts

- **Category:** Logical flaws
- **Source:** BaseStrategy.sol
- **Status:** Fixed

Description

When closing a position, token swap is performed to ensure that the closer can repay the debt, for example, when **operation == EXACT_IN**, tokens of **borrowAmt** are required to be excluded from the swap, and when **operation == EXACT_OUT**, tokens of **borrowAmt** are required to be swapped. The issue here is that the closer needs to pay not only the **borrowAmt** but also the **shareProfitAmts**, which causes the closure to fail when **percentSwapE18 = 100%** due to insufficient tokens. Although the closer can adjust the **percentSwapE18** to make the closure successful, it greatly increases the complexity.

```
for (uint i; i < swapParams.length; ) {
 // find excess amount after repay
 uint swapAmt = swapParams[i].operation == SwapOperation.EXACT_IN
  ? IERC20(swapParams[i].tokenIn).balanceOf(address(this)) - openTokenInfos[i].borrowAmt
  : openTokenInfos[i].borrowAmt - IERC20(swapParams[i].tokenOut).balanceOf(address(this));
 swapAmt = (swapAmt * swapParams[i].percentSwapE18) / ONE_E18;
```

```
if (swapAmt == 0) {
  revert SwapZeroAmount();
}
```

Recommended mitigation

Consider taking **shareProfitAmts** into account when calculating **swapAmt**.

Team response

Fixed

Mitigation Review

The team has fixed it as recommended to make the logic correct.

TRST-M-5 Freezing of repaid debts can cause DoS when borrowing

- **Category:** Logical flaws
- **Source:** BaseLendingPool.sol#L138-L140
- **Status:** Fixed

Description

When a debt is repaid, the repaid amount gets frozen via *freezeBuckets.addToFreezeBuckets()*. In most scenarios, the repaid amount won't be frozen by the mint freezing mechanism since the amount of time that has passed since the borrowed and repaid amount was deposited will almost always be greater than **mintFreezeInterval** (which is expected to be 1 day). Thus, a lender can withdraw a repaid amount while it's frozen in FreezeBuckets. This can cause a miscalculation of borrowable funds in the *BaseLendingPool.getBorrowableAmount()* function: in the worst case scenario, *freezeBuckets.getLockedAmount()* can return a value that's bigger (it'll include the repaid amount) than the current balance of the pool (the repaid amount will be withdrawn), which will case a revert and block borrowing.

Recommended mitigation

Consider not freezing repaid funds.

Team response

Fixed

Mitigation Review

The team addressed this issue by changing the algorithm (unlocking the same amount from the buckets when the user made withdrawals).

TRST-M-6 Pending fees calculations don't allow overflowing/underflowing

- **Category:** Arithmetic flaw
- **Source:** UniswapV3PositionViewer.sol

- **Status:** Fixed

Description

When computing pending fees in the *UniswapV3PositionViewer._computePendingFeesToBeEarned()* function, the calculations of **feeGrowthBelowX128**, **feeGrowthAboveX128**, and **feeGrowthInsideX128** don't allow under- and overflowing. However, the respective calculations in Uniswap V3 are designed to underflow and overflow (for more information, refer to this issue and this discussion). As a result, executing *_computePendingFeesToBeEarned()* can revert in some situations, causing transaction reverts.

Recommended mitigation

In the *_computePendingFeesToBeEarned()* function, consider wrapping the fee growth calculations in **unchecked**. This is what Uniswap does in the 0.8 branch.

Team response

Fixed

Mitigation Review

The team addressed this issue by wrapping the fee growth calculations in **unchecked** in *_computePendingFeesToBeEarned().*

TRST-M-7 Changing liquidation vault or token makes liquidations impossible

- **Category:** Logical flaws
- **Source:** UniswapV3Strategy.sol
- **Status:** Fixed

Description

When *UniswapV3Strategy* is initialized, it approves spending of the liquidation token to the liquidation vault. The addresses of the vault and the token are read from the *Config* contract, which allows the "exec" role to change them. However, after liquidation vault or token is changed, token spending is not re-approved. As a result, liquidations will always revert because the new vault won't be able to take liquidation tokens from the strategy contract (or the old vault won't be able to take the new liquidation token, if the token was changed).

Recommended mitigation

Strategy contracts need a (restricted) way to approve arbitrary tokens to arbitrary addresses. *BaseStrategy.approve()* allows that, but it only approves to whitelisted routers. Thus, our recommendation is to allow any spender address in the *BaseStrategy.approve()* function.

Team response

Fixed

Mitigation Review

The team addressed this issue by extending target approval to either liquidation vault or router in *BaseStrategy.approve()*.

TRST-M-8 The freeze mechanism reduces the borrowableAmount, which reduces Lender's yield

- **Category:** Logical flaws
- **Source:** BaseLendingPool.sol
- **Status:** Fixed

Description

The contract has two freeze intervals, **mintFreezeInterval** and **freezeBuckets.interval**, the former to prevent users from making flash accesses and the latter to prevent borrowers from running out of funds.

Both freeze intervals are applied when a user deposits, and due to the difference in unlocking time, it significantly reduces **borrowableAmount** and thus reduces Lender's yield.

```
function _mintInternal(
  address _receiver,
  uint _balanceIncreased,
  uint _totalAsset
) internal returns (uint mintShares) {
  unfreezeTime[_receiver] = block.timestamp + mintFreezeInterval;

  if (freezeBuckets.interval > 0) {
    FreezeBuckets.addToFreezeBuckets(freezeBuckets, _balanceIncreased.toUint96());
  }
```

Consider **freezeBuckets.interval == mintFreezeInterval = 1 day**, 100 ETH in the LendingPool, and **borrowableAmount = 100 ETH**.

At day 0 + 1s, Alice deposits 50 ETH, **borrowableAmount = 150 ETH - lockedAmount(50 ETH) = 100 ETH**, the 50 ETH frozen in **freezeBuckets** will be unlocked on day 2, while **unfreezeTime[alice] = day 1 + 1s**.

At day 1 + 1s, **unfreezeTime[Alice]** is reached, Alice can withdraw 50 ETH, **borrowableAmount = 100 ETH - LockedAmount(50 ETH) = 50 ETH**.

If Bob wants to borrow the available funds in the Pool at this time, Bob can only borrow 50 ETH, while the available funds are actually 100 ETH, which will reduce Lender's yield by half.

At day 2 + 1s, **freezeBuckets** is unfrozen and **borrowableAmount = 100 ETH - LockedAmount(0 ETH) = 100 ETH**.

Recommended mitigation

Consider making **mintFreezeInterval >= 2 * freezeBuckets.interval**, which makes **unfreezeTime** greater than the unfreeze time of **freezeBuckets**.

Team response

Fixed

Mitigation Review

The team addressed this issue by changing the algorithm (unlocking the same amount from the buckets when the user made withdrawals).

Low severity findings

TRST-L-1 In some setter functions of the Config.sol, the input should be checked

- **Category:** Validation issues
- **Source:** Config.sol
- **Status:** Fixed

Description

In *Config.setLpCollateralFactorBPSs()/setCollateralFactors()*, the factors set should be less than **ONE_E18**.

Recommended mitigation

In *Config.setLpCollateralFactorBPSs()/setCollateralFactors()* check that the factor is less than **ONE_E18**

Team response

Fixed

Mitigation Review

The team verified the parameters in the setter.

TRST-L-2 In _takeToken(), when the token address is WETH, the user can only use ETH, not WETH

- **Category:** Logical flaws
- **Source:** BaseStrategy.sol
- **Status:** Acknowledged

Description

In *_takeToken(),* when the token address is WETH, it checks **msg.value == amount**, which means that the user can only use ETH and not WETH.

Recommended mitigation

Change to

```
function _takeToken(
  address _gateway,
```

```
    address _token,
    address _from,
    address _to,
    uint128 _amount
  ) internal returns (uint128 amount) {
-   if (_from == _to || (_token == WETH && msg.value != _amount)) {
+   if (_from == _to || (_token == WETH && (msg.value != _amount || msg.value != 0)) {
      revert InvalidTakeToken();
    }
    if (_amount > 0) {
-     if (_token == WETH) {
+     if (_token == WETH && msg.value == _amount) {
        IWETH9(WETH).deposit{value: msg.value}();

        if (_to != address(this)) {
          IERC20(WETH).safeTransfer(_to, msg.value);
        }
        amount = msg.value.toUint128();
      } else {
        uint balanceBefore = IERC20(_token).balanceOf(address(this));
        IStrategyGateway(_gateway).transmitToken(_token, _from, _to, _amount);

        amount = (IERC20(_token).balanceOf(address(this)) - balanceBefore).toUint128();
      }
    }
  }
```

Team response

Intended behavior

Mitigation Review

The team says this makes end users not have to worry about WETH at all.

TRST-L-3 When swapping tokens, swapParams.tokenIn/tokenOut must be underlyingToken

- **Category:** Logical flaws
- **Source:** BaseStrategy.sol
- **Status:** Acknowledged

Description

When swapping tokens in some functions, there is no requirement that **swapParams.tokenIn/tokenOut** must be the **underlyingToken**, allowing an attacker to use the tokens in the contract for arbitrary swaps.

Assuming Alice accidentally sends some **tokenA** into the contract, Bob can use this issue to swap these **tokenA** into **underlyingToken** and withdraw them when closing a position.

Recommended mitigation

Consider requiring **swapParams.tokenIn/tokenOut** to be **underlyingToken** when swapping tokens.

Team response

Acknowledged.

Mitigation Review

The team says that the swapHelper would have prevented this from whitelisting.

TRST-L-4 addToFreezeBuckets() is inconsistent with the documentation

- **Category:** Specification issues
- **Source:** FreezeBuckets.sol
- **Status:** Fixed

Description

Suppose user deposits tokens at **timestamp == startTimestamp+interval**. According to the docs, the tokens will be unlocked at **startTimestamp+2*interval**, but they will actually be unlocked at **startTimestamp+3*interval**.

Recommended mitigation

Consider making the deposit_flow.png diagram consistent with the code.

Team response

Fixed.

Mitigation Review

The team uploaded the correct image to the documentation.

TRST-L-5 UsingAccessControllerUpgradeable doesn't reserve storage slots

- **Category:** Upgradeability issues
- **Source:** UsingAccessControllerUpgradeable.sol
- **Status:** Fixed

Description

The *UsingAccessControllerUpgradeable* contract is an upgradeable contract that implements the logic of interacting with an *AccessController* contract for multiple contracts in the project. *UsingAccessControllerUpgradeable*, however, doesn't reserve storage slots for future updated to the contract: adding a new storage variable to the contract will shift the layout of storage variable of all contract that inherit from it, which will result in corrupted state of the contracts.

Recommended mitigation

Consider reserving 49 storage slots *UsingAccessControllerUpgradeable* by defining a storage variable of type **uint256[49]** at the end of the contract. For more information refer to the OpenZeppelin's Writing Upgradeable Contracts guide.

Team response

Fixed

Mitigation Review

The team reserved storage slots in *UsingAccessControllerUpgradeable*

TRST-L-6 Liquidation token claiming event spam

- **Category:** Event emission issues
- **Source:** LiquidationVault.sol
- **Status:** Fixed

Description

Claiming liquidation tokens via the *LiquidationVault.claim()* function emits the *Claimed* event. The function takes an array of **BatchClaimInfo**, each of which can contain multiple **ClaimInfos**; claiming an amount specified in a **ClaimInfo** emits the *Claimed* event. However, the event is emitted even when the claimed amount is 0 (i.e. when the caller is not eligible for any liquidation tokens). As a result, the *LiquidationVault.claim()* function can emit multiple *Claimed* events in one call, while no tokens are actually claimed. This spamming can affect off-chain monitoring and analysis tools that, for example, watch the event to track all claims.

Recommended mitigation

Consider emitting the *Claimed* event only after a positive amount of liquidation tokens was claimed.

Team response

Fixed

Mitigation Review

The team makes the function revert when trying to claim 0.

TRST-L-7 Total lender profit USD value doesn't include the liquidation premium

- **Category:** Logical flaws
- **Source:** BasePositionViewer.sol
- **Status:** Fixed

Description

During a liquidation, the amount of profit to be shared with lenders is increased by the liquidation premium. However, the total USD value of lender profit (**lenderProfitUSDValueE36**) is not updated accordingly. As a result, the calculation of the funds to return to the borrower is increased by the liquidation premium

(**userUSDValueReturnE36**), leading to an increased required payment for the liquidator (**requiredPayAmount**). In the worst case scenario, when user's position has accrued bad debt (i.e. the total collateral value is less than the amount of funds to pay to lenders), the user may still have some of their funds returned to them due to a false positive in this if-clause:.

Recommended mitigation

When adding the liquidation premium to the **sharingProfitTokenAmts** array elements, consider also adding it to the **lenderProfitUSDValueE36** variable.

Team response

The original design is for the liquidator to pay for this, hence not adding to the value, but we think this will just overcomplicate things, since the liquidator will need to wait for when it's profitable anyways. So we'll just update the value as suggested. But the logic should already work as is, it's just an alternative solution, so the issue should be informational.

Mitigation Review

Informational, agreed.

TRST-L-8 Freezing of deposited funds freezes all user shares

- **Category:** Logical flaws
- **Source:** BaseLendingPool.sol
- **Status:** Acknowledged

Description

Newly provided liquidity in pools is frozen for **mintFreezeInterval** seconds. During the freeze period, share tokens cannot be redeemed and transferred. However, the freeze period disables transferring of all user's share tokens, including those for which the freeze period has expire and those that were received from other users:

1. If a user makes multiple deposits, the most recent of them will freeze all previous deposited funds by the user.
2. If a user receives shares from another user while their recent deposited shares are frozen, they won't be able to transfer or redeem the received shares until the freeze period has expired.

Recommended mitigation

In the *_mintInternal()* function, consider freezing only deposited amounts. The improved mechanism needs to correctly handle multiple deposits and guarantee that amounts deposited earlier get unfrozen in time, while amount deposited later remain frozen.

Team response

Intended behavior.

Mitigation Review

Informational, agreed. Will be left in the report for the awareness of users.

TRST-L-9 RewardVault distributes rewards based on the percentage of the balance, which can cause Lender to lose some profit

- **Category:** Logical flaws
- **Source:** RewardVault.sol
- **Status:** Acknowledged

Description

When RewardVaultWorker calls *RewardVault.distributeReward()* to distribute rewards to the LendingPool, the maximum number of rewards to distribute is less than the **balance * maxRewardDistributionFactorE18 / 1e18**. As long as **maxRewardDistributionFactorE18 < 1e18**, a portion of the reward will remain in RewardVault and cannot be sent to LendingPool, which will cause Lender to lose some profit.

Recommended mitigation

Consider a new emission algorithm, like dividing the current balance by a fixed time (30 days) for the emission rate and multiplying by the time interval for the emission quantity.

Team response

This is the intended behavior.

Mitigation Review

It is better to document this behavior.

Additional recommendations

Using separate maxDelayTime for ETH-USD feed.

In ChainlinkAdapterOracle, when the token uses **refETH** to get the price, it will use the same **maxDelayTime** for both feeds.

On Arbitrum with a 24H heartbeat for ETH-USD and a 24H heartbeat for *-ETH, it will work just fine.

But on Ethereum, ETH-USD has a heartbeat of 1H and *-ETH has a heartbeat of 24H. Since they use the same heartbeat, the heartbeat needs to be slower of the two or else the contract would be nonfunctional most of the time. The issue is that it would allow the consumption of potentially very stale data from the ETH-USD feed.

LendingProxy.initialize() should call __ReentrancyGuard_init()

It is best practice for contracts that inherit from ReentrancyGuardUpgradeable to call *__ReentrancyGuard_init()* in the initialize function.

Centralization risks

transferToTreasury() does not check whether the LendingPool has been delisted

transferToTreasury() is used to transfer rewards from a delisted LendingPool to Treasury, but it does not check if the LendingPool has been delisted, which allows it to transfer rewards from any LendingPool to Treasury.

```
  /// @dev transfer fund to treasury (incase: delist lending pool)
  /// @param _token token address to transfer.
  function transferToTreasury(address _token) external override onlyAuthorized(keccak256('exec'))
{
    uint totalReward = IERC20(_token).balanceOf(address(this));
    // get treasury address.
    address treasury = ILendingProxy(lendingProxy).treasury();
    // transfer to treasury address.
    IERC20(_token).safeTransfer(treasury, totalReward);

    emit TransferToTreasury(treasury, totalReward);
  }
```

feeBPS and lenderLiquidatePremiumBPS need to be limited to a maximum value

feeBPS is used to determine the profit received by the Lending Pool, and lenderLiquidatePremiumBPS is used to determine the premium paid by the liquidator, when both values are large, the user will suffer a loss, so a reasonable limit should be set for both values.

```
  function shareProfit(address _token, uint _profit) external {
    uint toTreasury = (_profit * feeBPS[_token]) / BPS;
    uint toRewardVault = _profit - toTreasury;
    // send to treasury
    IERC20(_token).safeTransferFrom(msg.sender, treasury, toTreasury);
    // send to reward vault
    IERC20(_token).safeTransferFrom(msg.sender, rewardVault, toRewardVault);

    emit ProfitShare(msg.sender, _token, toRewardVault, toTreasury, block.timestamp);
  }
  ...
      uint lenderLiquidatationPremiumBPS = IConfig(config).lenderLiquidatePremiumBPS();
     for (uint i; i < sharingProfitTokenAmts.length; ) {
      sharingProfitTokenAmts[i] +=
        (pos.openTokenInfos[i].borrowAmt * lenderLiquidatationPremiumBPS) /
        BPS;
      unchecked {
        ++i;
      }
     }
```

The owner can grant anyone the whitelistedStrategy role to transfer the tokens of approved users

The owner of an AccessController contract can grant any role to anyone, and the **whitelistedStrategy** role can call *StrategyGateway.transferToken()* to transfer the approved user's tokens.

A more transparent mechanism for granting user roles should be used.

```
function transmitToken(
  address _token,
  address _from,
  address _to,
  uint _amount
) external onlyAuthorized(keccak256('whitelistedStrategy')) {
  IERC20(_token).safeTransferFrom(_from, _to, _amount);
}
```