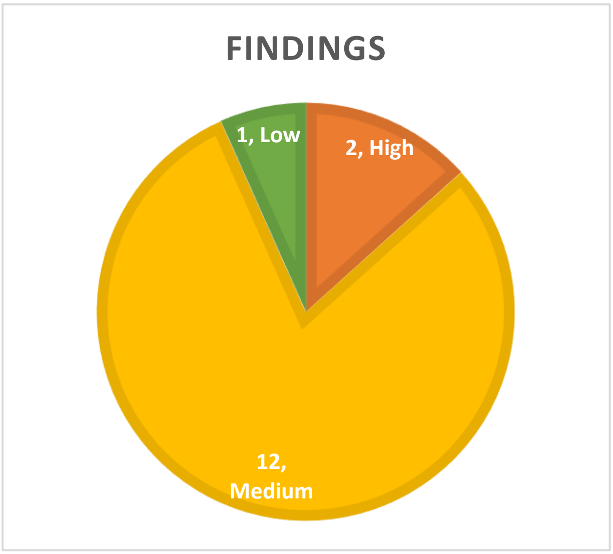
Trust Security

Smart Contract Audit

Init Capital

22/11/2023

Executive summary

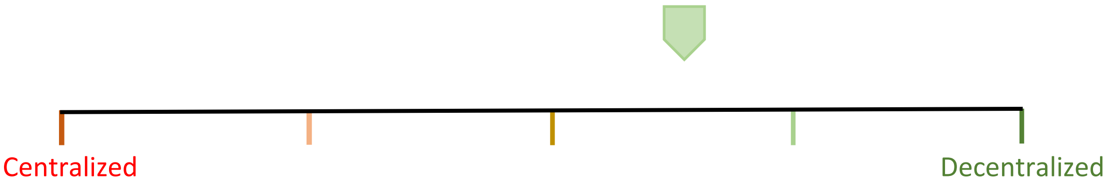


Category	Lending
Audited file count	10
Lines of Code	1189
Auditor	cccz,0xladboy
Time period	13/11/2023-22/11/2023

Findings

Severity	Total	Fixed	Acknowledged
High	2	-	-
Medium	12	-	-
Low	1	-	-

Centralization score



Signature

Digitally signed by Trust
Date: 2023-11-22 16:54:04

Document properties

Versioning

Version	Date	Description
0.1	22/11/2023	Client report

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

- core/InitCore.sol
- core/PositionManager.sol
- core/Config.sol
- core/IncentiveCalculator.sol
- oracle/InitOracle.sol
- oracle/PythOracleReader.sol
- oracle/Api3OracleReader.sol
- wrapper/WrapCenter.sol
- lending_pool/LendingPool.sol
- common/library/UncheckedIncrement.sol

Repository details

- **Repository URL:** https://github.com/init-capital/init-audit
- **Commit hash:** 3399c73b97e0810d7734a11771d10ee99d1753c0

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 a leading auditor at competitive auditing service Code4rena, reported several critical issues to Immunefi bug bounty platform and is currently a Code4rena Supreme Court 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.

0xladboy is a researcher specializing in blockchain security, known for consistently providing top-tier audits for clients. In 2023, he was recognized among the top 10 in the Code4rena rankings and he is the top 5 in the past 90 days in the code4rena leaderboard at the time of the audit.

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	**Good**	Project has kept code as simple as possible, reducing attack risks
Documentation	**Moderate**	Project is moderately documented.
Best practices	**Good**	Project mostly follows best practices.
Centralization risks	**Moderate**	Project has some centralization concerns.

Findings

High severity findings

TRST-H-1 LendingPool is subject to inflation attack via donation

- **Category:** Precision loss errors
- **Source:** LendingPool.sol
- **Status:** Open

Description

This issue is an alternative form of the vault inflation attack. When user wishes to create a collateral position, they need to call *mintTo()* in *initCore*, which then calls *mint()* in *LendingPool*.

```
    function mint(address _receiver) external onlyCore accrue returns (uint shares) {
        uint _cash = cash;
        uint _totalSu
pply = totalSupply();
        uint newCash = IERC20(underlyingToken).balanceOf(address(this));
        uint amt = newCash - _cash;
        _require(amt > 0, Errors.ZERO_VALUE);
        shares = _totalSupply > 0 ? (amt * _totalSupply) / (_cash + totalDebt) : amt;
        cash = newCash;
        _mint(_receiver, shares);
    }
```

When **totalSupply** is 0, the attack pattern below is possible:

1. Attacker front-runs the depositor and deposits 1 wei token and gets 1 share.
2. Attacker transfer the fund into *LendingPool* directly and then call *syncCash()* to own all liquidity of *LendingPool.*
3. the victim deposits 1e18 wei WETH. However, the depositor gets 0 shares: **1e18 * 1 (totalSupply) / (1e18 + 1) = 1e18 / (1e18 + 1) = 0**. Since the depositor gets 0 shares, **totalSupply** remains at 1.
4. the attacker still has the 1 only share ever minted and thus the withdrawal of that 1 share takes away everything in the vault.

```
    function syncCash() external accrue returns (uint newCash) {
        newCash = IERC20(underlyingToken).balanceOf(address(this));
        _require(newCash >= cash, Errors.INVALID_AMOUNT_TO_REPAY); // flash not repay
        cash = newCash;
    }
```

Recommended mitigation

When **totalSupply** == 0, send an initial amount of shares to the zero address to enable share dilution.

Consider reviewing the implementation of ERC4626 from OZ.

Team response

TBD

TRST-H-2 getCollateralCredit_e36() calculates wLp value without considering wLp amount

- **Category:** Logical flaws
- **Source:** InitCore.sol
- **Status:** Open

Description

getCollateralCredit_e36() is used to calculate the collateral credit for an account. When calculating the collateral value, for Pool tokens it multiplies the price by the amount.

```
            uint tokenPrice_e36 = IBaseOracle(_oracle).getPrice_e36(token);
            uint tokenValue_e36 = (tokenPrice_e36 * shares[i] *
ILendingPool(pools[i]).totalAssets()) /
                IERC20(pools[i]).totalSupply();
```

However, for wLp, it only considers the price of a single wLp token, but not the wLp amount, which can lead to a significant reduction in the calculated collateral credit, thus making the user more vulnerable to liquidation.

```
                uint wLpPrice_e36 = IBaseWrapLp(wLps[i]).calculatePrice_e36(ids[i][j],
_oracle);
                TokenFactors memory factors = _config.getTokenFactors(mode,
IBaseWrapLp(wLps[i]).lp(ids[i][j]));
                collateralCredit_e54 += wLpPrice_e36 * factors.collateralFactor_e18;
```

Recommended mitigation

It is recommended to consider the wLp amount when calculating wLp value, which can be fetched through *getCollateralInfo()*.

Team response

TBD

Medium severity findings

TRST-M-1 When admin pauses collateralization, users can still easily collateralize

- **Category:** Logical flaws
- **Source:** InitCore.sol
- **Status:** Open

Description

When user wants to add more collateral, they need to call *collateralize()*, and if the admin pauses the collateral, the transaction should revert in error **Errors.COLLATERALIZE_PAUSED**.

```
    function collateralize(uint _posId, address _pool) public virtual
onlyPosOwner(_posId) nonReentrant {
        IConfig _config = IConfig(config);
        // check mode status
        uint16 mode = _getPositionMode(_posId);
        _require(_config.getModeStatus(mode).canCollateralize,
Errors.COLLATERALIZE_PAUSED);
        // check if the position mode supports _pool
        _require(_config.isAllowedForCollateral(mode,
ILendingPool(_pool).underlyingToken()), Errors.INVALID_MODE);
        // update collateral on the position
        uint amtColl = IPositionManager(POS_MANAGER).addCollateral(_posId, _pool);

        emit Collateralize(_posId, _pool, amtColl);
    }
```

However, even when the collateral is paused, user can still inflate the collateral worth by transferring the asset to the lending pool directly and then call *syncCash().*

This is because the user collateral worth is computed using the code below:

```
            uint tokenValue_e36 = (tokenPrice_e36 * shares[i] *
ILendingPool(pools[i]).totalAssets()) /
                IERC20(pools[i]).totalSupply();
            TokenFactors memory factors = _config.getTokenFactors(mode, token);
            collateralCredit_e54 += tokenValue_e36 * factors.collateralFactor_e18;
```

In case an underlying asset is hacked and hacker can mint the token infinitely, and the admin wants to prevent further collateralization, users can still transferring the asset to the lending pool to inflate *totalAssets()* and borrow all asset out.

Recommended mitigation

Do not let user inflate the collateral worth by transferring asset to the lendingPool directly, consider tracking the total assets internally.

Team response

TBD

TRST-M-2 Some key supplying and borrowing parameters are not checked

- **Category:** Validation flaws
- **Source:** IConfig.sol

- **Status:** Open

Description

```
struct ModeConfiguration {
   EnumerableSet.AddressSet collateralTokens;
   EnumerableSet.AddressSet borrowTokens;
   uint64 targetHealthAfterLiquidation_e18;
   mapping(address => TokenFactors) factors;
   uint128 minBorrowUSD;
   uint128 maxBorrowUSD;
   ModeStatus status;
}

struct PoolConfiguration {
   uint supplyCap;
   uint borrowCap;
   bool canMint;
   bool canBurn;
   bool canBorrow;
   bool canRepay;
}
```

In *IConfig*, the protocol intends to set **minBorrowUSD** and **maxBorrowUSD** in the mode configuration and **supplyCap** and **borrowCap** in pool configuration.

But when user borrows or supplies the token, the code never validates if the **supplyCap** or **borrowCap** is reached, and in borrows, it also never validates if the **minBorrowUSD** and **maxBorrowUSD** is reached.

Recommended mitigation

It is recommended to validate whether the **supplyCap** / **borrowCap / minBorrowUSD / maxBorrowUSD** are reached based on the configuration.

Team response

TBD

TRST-M-3 Incorrect interface for Pyth oracle

- **Category:** Typo issues
- **Source:** PythOracleReader.sol
- **Status:** Open

Description

In the *InitOracle* implementation, the code expects the underlying primary oracle and secondary oracle source to implement the function *getPrice_e36()*.

```
        // get price from primary source
       try IBaseOracle(primarySource[_token]).getPrice_e36(_token) returns (uint
primaryPrice_e36_) {
           primaryPrice_e36 = primaryPrice_e36_;
           isPrimarySourceValid = true;
       } catch {}
```

```
        // get price from secondary source
        try IBaseOracle(secondarySource[_token]).getPrice_e36(_token) returns (uint
secondaryPrice_e36_) {
            secondaryPrice_e36 = secondaryPrice_e36_;
            isSecondarySourceValid = true;
        } catch {}
```

However, in *PythOracleReader*, the implemented function is *getPrice()*. The *PythOracle* cannot be used as either primary or secondary oracle because of interface mismatch.

Recommended mitigation

Change the interface function from *getPrice()* to *getPrice_e36()* in *PythOracleReader*.

Team response

TBD

TRST-M-4 The borrow function perpetually reverts due to division by zero

- **Category:** Division by zero errors
- **Source:** LendingPool.sol
- **Status:** Open

Description

When borrowing, the code will divide by the totalDebt. However, **totalDebt** starts as zero, so borrow will always revert due to division by zero.

```
    function borrow(address _receiver, uint _amt) external onlyCore accrue returns
(uint shares) {
        _require(_amt <= cash, Errors.NOT_ENOUGH_CASH);
        shares = _amt.mulDiv(totalDebtShares, totalDebt, MathUpgradeable.Rounding.Up);
        totalDebtShares += shares;
        totalDebt += _amt;
        cash -= _amt;
        IERC20(underlyingToken).safeTransfer(_receiver, _amt);
    }
```

Recommended mitigation

It is recommended to specifically handle the case where **totalDebt** is 0.

```
    function borrow(address _receiver, uint _amt) external onlyCore accrue returns
(uint shares) {
        _require(_amt <= cash, Errors.NOT_ENOUGH_CASH);
-       shares = _amt.mulDiv(totalDebtShares, totalDebt, MathUpgradeable.Rounding.Up);
+       shares = totalDebt > 0 ?_amt.mulDiv(totalDebtShares, totalDebt,
MathUpgradeable.Rounding.Up) : _amt;
        totalDebtShares += shares;
        totalDebt += _amt;
        cash -= _amt;
        IERC20(underlyingToken).safeTransfer(_receiver, _amt)
```

Team response

TBD

TRST-M-5 If underlying is a rebase token, the protocol exhibits

- **Category:** Logical flaws
- **Source:** LendingPool.sol
- **Status:** Open

Description

In *lendingPool*, the code relies on *syncCash()* to make sure the accounting of cash state is up-to-date.

```
  function syncCash() external accrue returns (uint newCash) {
       newCash = IERC20(underlyingToken).balanceOf(address(this));
       _require(newCash >= cash, Errors.INVALID_AMOUNT_TO_REPAY); // flash not repay
       cash = newCash;
   }
```

However, if the underlying token has rebasing behavior and its balance changed outside of transfer, the require check **newCash >= cash** may not be always true, negative rebasing can make *syncCash()* revert, while positive rebasing gives the funds to the contract, not user.

In case when positive rebase happens, anyone can call *mintTo()* directly to mint Pool token and take the rebase amount. In case a negative rebase happens, the next user *mint()* will grant less tokens than expected, to fill the balance gap created on rebase.

One of the most widely adopted rebase token is stETH.

Recommended mitigation

Recommended to add in documentation that rebase token is not supported, or change the code logic to support changing balances.

Team response

TBD

TRST-M-6 setPositionMode() does not validate the mode status

- **Category:** Validation issues
- **Source:** InitCore.sol
- **Status:** Open

Description

Users can update position mode by calling *setPositionMode()*. It validates that the new mode must contain all of the user's collateral and borrow tokens, but it does not validate the new mode's status.

Consider the following scenario.

1. Mode A contains token A and B, and **canBorrow** and **canRepay** are true.

2. Mode B contains token A and C, and **canBorrow** and **canRepay** are false (this may be a deprecated mode because it no longer supports token C (like LUNA)).

3. Alice can borrow token A in mode A and then move to mode B.

Since **canRepay** is false, she will not be liquidated because liquidation will revert in *_checkRepayStatus()*.

```
    function _liquidateInternal(
        uint _posId,
        address _poolToRepay,
        uint _repayShares
    ) internal returns (LiquidateLocalVars memory vars) {
        vars.config = IConfig(config);
        vars.mode = IPositionManager(POS_MANAGER).getPositionMode(_posId);
        // check status
        _checkRepayStatus(vars.config, vars.mode, _poolToRepay);
```

Recommended mitigation

It is recommended to check that the status of the old and new mode is consistent in *setPositionMode()*.

Team response

TBD

TRST-M-7 Some functions in InitCore do not take interest into account

- **Category:** Logical flaws
- **Source:** InitCore.sol
- **Status:** Open

Description

getPositionHealth_e18() calls *getBorrowCredit_e36()* and *getCollateralCredit_e36()* to calculate the position health factor, which are view functions that do not call *accrue()* to accrue interest in the pools, and since the collateral and borrowing values are not up to date, *getPositionHealth_e18()* will return a stale health factor.

Therefore, before calling *getPositionHealth_e18(),accrue()* must be to get the latest health factor, which is not done in *InitCore*, for example in *borrow()* and *liquidate()*.

Recommended mitigation

Before calling *getPositionHealth_e18()* in *InitCore*, call *accrue()* of all the pools in the position to get the latest health factor. It is recommended to limit the number of pool users to avoid unbounded gas spending in the loop.

Team response

TBD

TRST-M-8 Paused pools still accrue interest, which leads to several risks

- **Category:** Logical flaws
- **Source:** LendingPool.sol
- **Status:** Open

Description

Admin can disable the repay status for given mode, but even when the repay status is paused, interest is always accruing because anyone can trigger *accrueInterest()* in the *LendingPool*.

```
    /// @inheritdoc IInitCore
    function repay(
        address _pool,
        uint _shares,
        uint _posId
    ) public virtual onlyPosOwner(_posId) nonReentrant returns (uint amt) {
        IConfig _config = IConfig(config);
        // check pool and mode status
        _checkRepayStatus(_config, _getPositionMode(_posId), _pool);
        // get position debt share
```

Interest accruing only depends on the time elapsed, but not whether the repayment is paused. This can create debt for user, and make user account unhealthy and eventually user's position is subject to liquidation.

Recommended mitigation

Consider not accruing interest when repayment is paused, or not allowing to disable repayment.

Team response

TBD

TRST-M-9 accrueInterest() pays higher fee to treasury than expected

- **Category:** Accounting flaws
- **Source:** LendingPool.sol
- **Status:** Open

Description

accrueInterest() is used to accrue interest on borrows and distribute the interest to the lender and treasury.

```
     function accrueInterest() public {
         uint _lastAccrued = lastAccrued;
         if (block.timestamp != _lastAccrued) {
             uint _totalDebt = totalDebt;
             uint _cash = cash;
             uint borrowRate_e18 = IIRM(interestRateModel).getBorrowRate_e18(_cash,
 _totalDebt);
```

```
            uint accruedInterest = (borrowRate_e18 * (block.timestamp - _lastAccrued)
* _totalDebt) / ONE_E18;
            uint reserve = (accruedInterest * reserveFactor_e18) / ONE_E18;
            // NOTE: no need to check totalSupply > 0, because if totalSupply == 0,
totalDebt == 0
            uint supply = totalSupply();
            if (supply > 0) _mint(treasury, (reserve * supply) / (_cash +
_totalDebt));

            totalDebt += accruedInterest;
            lastAccrued = block.timestamp;
        }
    }
```

The issue here is that the proportion of interest distributed to the treasury is greater than **reserveFactor_e18.**

Pool has 5% APR, has 10000 cash, 10000 borrowed, the total share is 20000

ReserveFee is 10%

One year later

accruedInterest = 10000 * 5% = 500

reserve = 500 * 10% = 50

treasuryShare = 50 * 20000/20000 = 50

treasuryAmount = 50 * 20500/20000 = 51.25

It is observed that the actual distributed **treasuryAmount** is larger than the expected **reserve.**

Recommended mitigation

It is recommended to change the shares minted to the treasure as follows, so that **treasuryShare = 50 * 20000/20450 = 48.9**, **treasuryAmount = 48.9 * 20500/20000 = 50** in the above calculation.

```
-           if (supply > 0) _mint(treasury, (reserve * supply) / (_cash +
_totalDebt));
+           if (supply > 0) _mint(treasury, (reserve * supply) / (_cash + _totalDebt +
accruedInterest - reserve));
```

Team response

TBD

TRST-M-10 coreCallback() flaw can lead to loss of ETH

- **Category:** Leak of value issues
- **Source:** WrapCenter.sol
- **Status:** Open

Description

The user can call *InitCore.callback()* for the remaining ETH and it may call *WrapCenter.coreCallback()*. When the user wants to call *WrapCenter.wrapNative()* in *WrapCenter.coreCallback()* to wrap the remaining ETH, since *WrapCenter.coreCallback()* doesn't send the ETH to *WrapCenter.wrapNative()*, these ETHs will be lost.

```
    function coreCallback(address, bytes calldata _data) external payable {
        _require(msg.sender == CORE, Errors.NOT_INIT_CORE);
        (bool success, ) = address(this).call(_data);
        _require(success, Errors.CALL_FAIL);
    }
```

Recommended mitigation

It is recommended to send the ETH in *WrapCenter.coreCallback()*.

```
    function coreCallback(address, bytes calldata _data) external payable {
        _require(msg.sender == CORE, Errors.NOT_INIT_CORE);
-       (bool success, ) = address(this).call(_data);
+       (bool success, ) = address(this).call{value: msg.value}(_data);
        _require(success, Errors.CALL_FAIL);
```

Team response

TBD

TRST-M-11 Race condition in liquidateWLp() can lead to loss for liquidator

- **Category:** Block sequencing flaws
- **Source:** InitCore.sol
- **Status:** Open

Description

When liquidating wLp, the wLp amount the liquidator receives is the minimum of the collateral balance and the liquidation incentive.

```
            uint wLpAmt = IPositionManager(POS_MANAGER).getCollateralWLpAmt(_posId,
_wLpToBurn, _tokenId);
            wLpAmtToBurn = IBaseOracle(oracle).getPrice_e36(vars.repayToken).mulDiv(
                vars.repayAmountWithLiqIncentive,
                IBaseWrapLp(_wLpToBurn).calculatePrice_e36(_tokenId, oracle)
            );
            wLpAmtToBurn = MathUpgradeable.min(wLpAmtToBurn, wLpAmt); // take min of
what's available
```

Consider the following scenario:

1. Alice has collateral of 1000 wLp.
2. Bob and Charlie initiate liquidation together.
3. Bob's transaction is executed first, the liquidation incentive is 900 wLP, and Bob receives 900 wLp.

4. Charlie's transaction is executed second, with a liquidation incentive of 300 wLP, but Charlie receives only 100 wLp.

Recommended mitigation

It is recommended to revert the transaction when the liquidation incentive is greater than the balance of the liquidated person, similarly to *liquidate()*.

Team response

TBD

TRST-M-12 Users can construct unbounded loops to prevent being liquidated

- **Category:** Unbounded gas spending issues
- **Source:** InitCore.sol
- **Status:** Open

Description

Take MockWLpUniV2 as reference, it will use new tokenIDs with each wrap. If the user wraps 1000 ETH/USD pair tokens, 1 token per wrap, 1,000 wraps, then the user will hold ERC1155 tokens with IDs from 0 to 1000, and _wLp.lp(ID) are all ETH/USD pair addresses.

```
    function wrap(address _lp, uint _amt, address _to, bytes calldata) external
returns (uint id) {
        IERC20(_lp).safeTransferFrom(msg.sender, address(this), _amt);
        id = nextId++;
        lps[nextId] = _lp;
        _mint(_to, id, _amt, '');
    }
```

The user can collateralize them, since they have same **_wLp** addresses and same **_wLp.lp()** addresses, so it bypasses the check in *collateralizeWLp()*.

```
    function collateralizeWLp(
        address _wLp,
        uint _tokenId,
        uint _posId
    ) public virtual onlyPosOwner(_posId) nonReentrant {
        IConfig _config = IConfig(config);
        uint16 mode = _getPositionMode(_posId);
        // check mode status
        _require(_config.getModeStatus(mode).canCollateralize,
Errors.COLLATERALIZE_PAUSED);
        // check if the wLp is whitelisted
        _require(_config.whitelistedWLps(_wLp), Errors.NOT_WHITELISTED_TOKEN);
        // check if the position mode supports _wLp
        _require(_config.isAllowedForCollateral(mode, IBaseWrapLp(_wLp).lp(_tokenId)),
Errors.INVALID_MODE);
        // update collateral on the position
        uint amtColl = IPositionManager(POS_MANAGER).addCollateralWLp(_posId, _wLp,
_tokenId);

        emit CollateralizeWLp(_wLp, _tokenId, _posId, amtColl);
    }
```

This will cause *getCollateralCredit_e36()* to revert due to the unbounded loop, and so the user prevents being liquidated.

```
        for (uint i; i < wLps.length; i = i.uinc()) {
            for (uint j; j < ids[i].length; j = j.uinc()) {
                uint wLpPrice_e36 = IBaseWrapLp(wLps[i]).calculatePrice_e36(ids[i][j],
_oracle);
                TokenFactors memory factors = _config.getTokenFactors(mode,
IBaseWrapLp(wLps[i]).lp(ids[i][j]));
                collateralCredit_e54 += wLpPrice_e36 * factors.collateralFactor_e18;
            }
        }
```

Recommended mitigation

It is recommended to limit the total number of pool and wLp user used as active market.

Team response

TBD

Low severity findings

TRST-L-1 Confidence interval validation could lead to issues

- **Category:** Oracle integration issues
- **Source:** PythOracleReader.sol
- **Status:** Open

Description

According to the doc from pyth oracle, "The aggregation algorithm itself is a simple two-step process. The first step computes the aggregate price by giving each publisher three votes — one vote at their price and one vote at each of their price +/- their confidence interval — then taking the median of all the votes. The second step computes distance from the aggregate price to the 25th and 75th percentiles of the votes, then selects the larger of the two as the aggregate confidence interval."

In the current code, the Pyth oracle is validating the confidence interval with price. The error message is **Errors.CONFIDENCE_TOO_HIGH**.

```
    // validate conf
    uint priceInUint = int(price).toUint256();
    _require(Math.mulDiv(conf, ONE_E18, priceInUint) <= config.maxConfDeviation_e18,
Errors.CONFIDENCE_TOO_HIGH);
```

However, when **conf * ONE_E18 / priceInUnit** is greater than **config.maxConfDeviation_e18** (it could be **priceInUnit** is too small or **conf * ONE_E18** too large) and confidence interval is in a reasonable range, oracle lookup will still revert, so it is not recommended to validate confidence interval along with the oracle price.

Recommended mitigation

The protocol can choose the validate confidence independently or drop the confidence internal validation.

Team response

TBD

Additional recommendations

TRST-R-1 Improve integration with the codebase by adhering to NFT semantics

All borrow / repay related actions are guarded by modifier *ensurePositionHealth().*

```
modifier onlyPosOwner(uint _posId) {
   _require(msg.sender == IERC721(POS_MANAGER).ownerOf(_posId), Errors.NOT_OWNER);
   _;
}
```

The modifier only validates if the msg.sender holds the current NFT, but the code should also enable the approved operator of NFT to borrow / repay on behalf of the NFT owner for third-party integrations.

TRST-R-2 Inconsistent behavior during decollateralization

When user decollateralizes the WLP, the WLP is transferred to the user.

However, when the WLP is liquidated, the smart contract burns the WLP for liquidator instead of transfering the WLP to liquidator.

```
// reduce and burn wLp to underlying for liquidator
(tokens, amts) = IPositionManager(POS_MANAGER).burnCollateralWLpTo(
   _posId,
   _wLpToBurn,
   _tokenId,
   wLpAmtToBurn,
   msg.sender
);
```

It is recommended to transfer the WLP to liquidator as well.

TRST-R-3 Whitelist behavior issue

When user calls *collateralizeWLP()* and *decollateralizeWLP(),* if the WLP is not whitelisted, the transaction will revert in this check.

```
// check wLp is whitelisted
_require(_config.whitelistedWLps(_wLp), Errors.NOT_WHITELISTED_TOKEN);
```

Removing a WLP from the whitelist should only block user from using a deprecated LP as collateral, but should not block the user from withdrawing the WLP fund.

TRST-R-4 Should document assumption of using Multicall

To create a collateralized position after user mints an NFT, they needs to complete multiple steps in one transaction:

1. Transfer the underlying token to *LendingPool.*
2. Call *mintTo()* in *InitCore* to mint *LendingPool* token.
3. Transfer the minted *LendingPool* token to *PositionManager.*
4. Collateralize to create collateralized position.

There should be documentation that informs users to carefully construct the payload to complete these step and execute the transaction via multicall.

if user does not execute this multiple step in one transaction, for example, if user transfer the underlying token to the *LendingPool*, and then in another transaction call *mintTo()*, attacker can frontrun the *mintTo()* and mint *LendingPool* token for themselves using victim's account balance.

TRST-R-5 Ensure pending fee is properly account towards account health

The code allows user to create collateral position with whitelisted LP token, but when evaluating LP worth, the LP price oracle should not only consider the LP worth, but also the pending fee. Otherwise, the user's collateral value is underpriced and can leads to false liquidation.

For example, if the underlying Lp token is Uniswap V3 LP token, the pending fee is separated from liquidity, so the protocol should count the pending fee towards the collateral credit.

TRST-R-6 Consider further protecting against bad debt risks

When users' collateral fails to cover the debt, bad debts are created, yet the protocol does not handle them.

Consider that Alice has collateral worth 10,000 USD, and the debt is worth 10,000 USD. The maximum liquidation incentive is 10%. After being liquidated, Alice will have a bad debt of 10,000-10,000/1.1 = **910 USD**.

In *LendingPool*, the **totalDebt** still contains 910 USD, however, due to the bad debt, those 910 USD will never be repaid. As the bad debts accumulate, the actual total assets of *LendingPool* are less than *totalAssets()*. Consider addressing bad debt through solutions like socializing bad debt.

Centralization risks

TRST-CR-1 Centralization Risks in Config.sol

In *Config.sol*, admin can call privileged functions to change the protocol configuration without any restrictions, which carries centralization risks.

For example, the admin can change the collateral factor and borrow factor arbitrarily, and their changes may cause the user to be liquidated. Additionally, admin can change the mode status so that users cannot collateralize/decollateralize/borrow/repay.

TRST-CR-2 Centralization Risks in IncentiveCalculator.sol

In *IncentiveCalculator.sol*, admin can call privileged functions to change the protocol configuration without any restrictions, which carries centralization risks.

For example, the admin can change the **maxIncentiveMultiplier** arbitrarily, and a very large **maxIncentiveMultiplier** can cause the liquidated person to lose excessive collateral.

TRST-CR-3 Centralization Risks in InitCore.sol

In *InitCore.sol*, admin can call privileged functions to change the protocol configuration without any restrictions, which carries centralization risks.

For example, the admin can change the config address to apply a new configuration that compromises users.

TRST-CR-4 Centralization Risks in LendingPool.sol

In *LendingPool.sol,* admin can call privileged functions to change the protocol configuration without any restrictions, which carries centralization risks.

For example, the admin can change the **reserveFactor** arbitrarily to distribute too much interest to the treasury.

TRST-CR-5 Centralization Risks in Api3OracleReader.sol/PythOracleReader.sol/InitOracle.sol

In *Api3OracleReader.sol/PythOracleReader.sol/InitOracle.sol,* admin can call privileged functions to change the protocol configuration without any restrictions, which carries centralization risks.

For example, the admin can change the oracle address arbitrarily to make the oracle return incorrect prices. These can then be exploited to drain the liquidity of the lending pool.

Systemic risks

TRST-SR-1 Oracles must be trusted to report correct prices

The protocol uses pyth and API3 oracles to get the prices of *LendingPool's* underlying tokens, which are used to calculate the value of collateral and debts. The protocol handles the case when oracles stop working due to timeout.

However, if the oracle reports incorrect price, this can cause the value of collateral and debts to be miscalculated, which can result in the healthy user being liquidated or the user taking what are effectively uncollateralized loans.

TRST-SR-2 Sharp drop in collateral leads to large bad debts

The protocol assumes that assets are liquid enough to be liquidated before bad debt, and there is no check for collateral cap in current implementation. If the cap is in use, it is assumed to be set conservatively.

In practice, users could use large amount of highly volatile tokens as collateral, and if that collateral depreciates rapidly, liquidation may result in large bad debts arising in the protocol. Since bad debt is not socialized, users who redeem later will not be able to redeem due to insufficient balances, which could lead to a run on the bank.