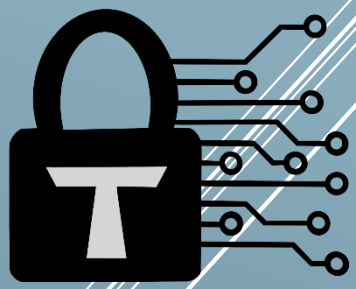
Trust Security

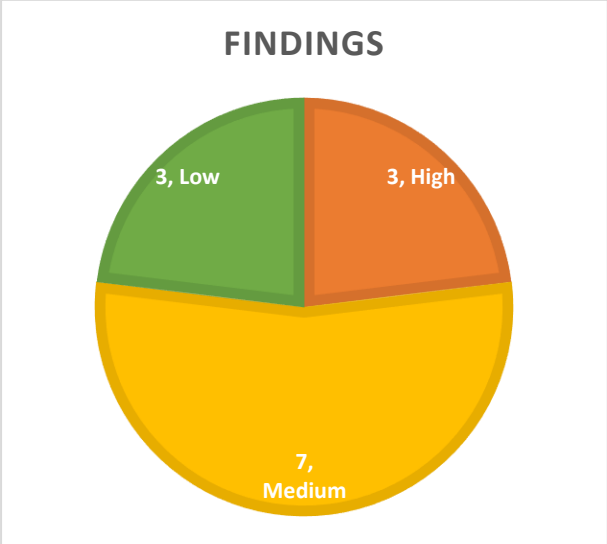


Smart Contract Audit

Beta Finance - Omni Protocol

25/10/2023

Executive summary



Category	Lending
Audited file count	7
Lines of Code	1044
Auditor	cccz
Time period	17/10/2023-25/10/2023

Findings

Severity	Total	Fixed	Acknowledged
High	3	3	-
Medium	7	1	6
Low	3	1	2

Centralization score

Signature

Digitally signed by Trust
Date: 2023-10-27 15:20:19

Document properties

Versioning

Version	Date	Description
0.1	25/10/2023	Client report
0.2	27/10/2023	Mitigation review
0.3	27/10/2023	Final Mitigation review

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

- src/OmniPool.sol
- src/OmniToken.sol
- src/OmniTokenNoBorrow.sol
- src/IRM.sol
- src/OmniOracle.sol
- src/WithUnderlying.sol
- src/SubAccount.sol

Repository details

- **Repository URL:** https://github.com/beta-finance/Omni_Trust
- **Commit hash:** de2327ac26cf5ca1a3b77b3d8ed8acc0918372dd
- **Mitigation review hash:** d932fe059dc928203236ce401e72191ce28b2fe2

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.

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 enterIsolatedMarket() can be called after borrowing to clear the debt

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

Description

When an account enters the **IsolatedMarket**, the **BorrowTier** of the account changes.

```
    function getAccountBorrowTier(AccountInfo memory _account) public view returns
(uint8) {
        address isolatedCollateralMarket = _account.isolatedCollateralMarket;
        if (_account.modeId == 0) {
            if (isolatedCollateralMarket == address(0)) {
                // Account has no isolated collateral market. Use tier 0 (lowest risk).
                return 0;
            } else {
                // Account has isolated collateral market. Use the market's risk
tranche.
                return marketConfigurations[isolatedCollateralMarket].riskTranche;
            }
        } else {
            // Account is in a mode. Use the mode's risk tranche.
            return modeConfigurations[_account.modeId].modeTranche;
        }
    }
```

The **BorrowTier** is used to get the user's debt.

```
            uint8 borrowTier = getAccountBorrowTier(_account);
            uint256                              borrowAmount                              =
IOmniToken(market).getAccountBorrowInUnderlying(_accountId, borrowTier);
            if (borrowAmount != 0) {
                ++eval.numBorrow;
                uint256 borrowValue = (borrowAmount * price) / PRICE_SCALE; // Rounds
down
                eval.borrowTrueValue += borrowValue;
                uint256 borrowFactor = marketCount == 1
                    ? SELF_COLLATERALIZATION_FACTOR
                    :            _account.modeId           ==           0           ?
uint256(marketConfiguration_.borrowFactor) : uint256(mode.borrowFactor);
                eval.borrowAdjValue   +=   (borrowValue   *   FACTOR_PRECISION_SCALE)   /
borrowFactor; // Rounds down

            }
```

Therefore, if the user enters the normal market first, borrows assets and then enters the **IsolatedMarket**, the user's **BorrowTier** will change, thus clearing the user's debt.

The POC that follows shows that when the user enters the **IsolatedMarket**, its debt will be cleared.

```
    function test_POC() public {
        uint256 amount1 = 1e2 * (10 ** uToken.decimals());
        oToken.deposit(0, 2, amount1);
        vm.startPrank(ALICE);
        uint256 amount2 = 1e2 * (10 ** uToken2.decimals());
        oToken2.deposit(0, 2, amount2);
        address[] memory markets = new address[](2);
        markets[0] = address(oToken);
        markets[1] = address(oToken2);
        pool.enterMarkets(0, markets);
        pool.borrow(0, address(oToken), amount1);

        OmniPool.Evaluation               memory               eval0               =
pool.evaluateAccount(address(ALICE).toAccount(0));
        assertEq(eval0.borrowTrueValue,amount1,"borrow value is 1e2");

        pool.enterIsolatedMarket(0, address(oToken3)); // change borrowTier

        OmniPool.Evaluation               memory               eval1               =
pool.evaluateAccount(address(ALICE).toAccount(0));
        assertEq(eval1.borrowTrueValue,0,"borrow value is 0");
        vm.stopPrank();
    }
```

Recommended mitigation

It is recommended to modify *enterIsolatedMarket()* as follows, to require that the account must not be in debt in *enterIsolatedMarket()*.

```
    function enterIsolatedMarket(uint96 _subId, address _isolatedMarket) external {
        bytes32 accountId = msg.sender.toAccount(_subId);
        AccountInfo memory account = accountInfos[accountId];
        require(account.modeId  ==  0,  "OmniPool::enterIsolatedMarket:  Already  in  a
mode.");
        require(
            account.isolatedCollateralMarket == address(0),
            "OmniPool::enterIsolatedMarket: Already has isolated collateral."
        );
        MarketConfiguration             memory             marketConfig             =
marketConfigurations[_isolatedMarket];
        if        (marketConfig.expirationTimestamp        <=        block.timestamp
|| !marketConfig.isIsolatedCollateral) {
            revert("OmniPool::enterIsolatedMarket: Isolated market invalid.");
        }
+       Evaluation memory eval = evaluateAccount(accountId);
+      require(eval.numBorrow == 0, "OmniPool::clearMarkets: Non-zero borrow count.");
        accountInfos[accountId].isolatedCollateralMarket = _isolatedMarket;
        emit EnteredIsolatedMarket(accountId, _isolatedMarket);
    }
```

Team response

Fixed.

Mitigation Review

The fix required the borrower to have no active borrows in *enterIsolatedMarket()*.

TRST-H-2 Underlying tokens logic does not normalize decimals

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

Description

Calculating **depositTrueValue** and **borrowTrueValue** in *_evaluateAccountInternal()* directly uses the cumulative value of the tokens amount multiplied by the price, which can result in the incorrect evaluation of the account due to the fact that different tokens have different decimals.

```
            address underlying = IWithUnderlying(market).underlying();
            uint256 price = IOmniOracle(oracle).getPrice(underlying); // Returns price
in 1e18
            uint256                          depositAmount                          =
IOmniTokenBase(market).getAccountDepositInUnderlying(_accountId);
            if (depositAmount != 0) {
                ++eval.numDeposit;
                uint256 depositValue = (depositAmount * price) / PRICE_SCALE; // Rounds
down
                eval.depositTrueValue += depositValue;
```

Consider USDC (6 decimals, 1 USD), WETH (18 decimals, 1600 USD).

Alice deposits 1 WETH, **depositTrueValue** = 1*1e18*1600.

Then Alice borrows 1e12 USDC, and the calculated **borrowTrueValue** = 1e12*1e6*1.

The value of the USDC borrowed by Alice is much larger than the value of the WETH deposited.

In addition, the issue also exists in the calculation of liquidation bonus on liquidation, where decimal normalization should also be applied.

```
        uint256 amount =
            IOmniToken(_params.liquidateMarket).repay(_params.targetAccountId,
msg.sender, borrowTier, _params.amount);
        (uint256       liquidationBonus,       uint256       softThreshold)       =
getLiquidationBonusAndThreshold(
            evalBefore.depositAdjValue,                     evalBefore.borrowAdjValue,
_params.collateralMarket
        );
        { // Avoid stack too deep
            uint256                           borrowPrice                           =
IOmniOracle(oracle).getPrice(IWithUnderlying(_params.liquidateMarket).underlying());
            uint256                           depositPrice                          =
IOmniOracle(oracle).getPrice(IWithUnderlying(_params.collateralMarket).underlying());
            uint256 seizeAmount = Math.ceilDiv(
                Math.ceilDiv(amount      *      borrowPrice,      depositPrice)      *
(LIQ_BONUS_PRECISION_SCALE   +   liquidationBonus),   //   Need   to   add   base   since
liquidationBonus < LIQ_BONUS_PRECISION_SCALE
                LIQ_BONUS_PRECISION_SCALE
            ); // round up
            seizedShares = IOmniTokenBase(_params.collateralMarket).seize(
                _params.targetAccountId, _params.liquidatorAccountId, seizeAmount
            );
```

Recommended mitigation

It is recommended to normalize decimals for the underlying tokens. For example, introducing the underlying token decimals to normalize the **price** directly in *OmniOracle.getPrice()*.

Team response

Fixed.

Mitigation Review

The fix normalized the returned **price** against the underlying token's decimals in *OmniOracle.getPrice()*.

TRST-H-3 enterMarkets() allows users to enter duplicate markets

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

Description

enterMarkets() does not duplicate check the **_markets** parameter, resulting in users being able to enter duplicate markets, which can lead to repeated counting of the user's deposits in *_evaluateAccountInternal()*.

The POC is as follows, it shows that when the user enters duplicate markets, the deposits will be counted repeatedly.

```
    function test_POC() public {
        address[] memory markets = new address[](5);
        markets[0] = address(oToken);
        markets[1] = address(oToken);
        markets[2] = address(oToken);
        markets[3] = address(oToken);
        markets[4] = address(oToken);
        pool.enterMarkets(0, markets);
        uint256 amount1 = 1e2 * (10 ** uToken.decimals());
        oToken.deposit(0, 0, amount1);
        OmniPool.Evaluation                memory                eval                =
pool.evaluateAccount(address(this).toAccount(0));
        assertEq(eval.depositTrueValue, 5 * amount1, "depositTrueValue is 5e2");
    }
```

Recommended mitigation

It is recommended to check the **_markets** parameter in *enterMarkets()* for duplicates.

Team response

Fixed.

Mitigation Review

The fix added duplicate checks for **_markets** in *enterMarkets()*.

Medium severity findings

TRST-M-1 Users can avoid losses by withdrawing assets before pausing due to bad debt

- **Category:** MEV attacks
- **Source:** OmniToken.sol
- **Status:** Acknowledged

Description

If bad debt arises after liquidation, the protocol will pause to prevent the user from withdrawing, and then call *OmniToken.socializeLoss()* to cover the bad debt using the user's deposits.

However, the user can front-run the liquidation transaction that pauses the protocol to withdraw assets before the protocol is paused, thus preventing losses from being caused by *OmniToken.socializeLoss()* later.

Recommended mitigation

It is recommended that instead of allowing the user to make immediate withdrawals, the user would be required to make a withdrawal request and would be allowed to make withdrawals after a period of time, and then the withdrawals would only be allowed within a limited time frame.

Team response

Reject w/ qualification, don't think it should be a Medium severity issue but acknowledge that MEV can happen in certain situations. It's fine to put this in the report, but we believe it shouldn't be a severity finding. We are aware of this problem, but feel there is no "good" way to mediate it. It would be impossible to determine when someone can and when someone cannot make withdrawals, as well as what the time frame should be, as different users deposit at different times. We also encourage that all liquidators use Flashbots RPC or something similar to prevent front running. This type of error would really occur if the liquidator submitted their transaction to the public mempool, which I believe most liquidators would not do, as then their liquidation transaction would also be front run. No changes to code made.

Trust Security response

We maintain that the issue should remain under medium severity and reflected to the users.

TRST-M-2 Whale can block other users from borrowing/withdrawing assets

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

Description

If a user borrows and repays in the same block, the user doesn't have to pay any interest since interest only grows over time. For *OmniToken*, the max borrowing is all deposits.

So, a whale can borrow all the deposits of a certain *OmniToken* in the first transaction of the block and repay it in the last transaction of the block, then all the borrowing/withdrawing transactions in between will fail due to insufficient assets.

Recommended mitigation

Consider incorporating anti-DOS measures to prevent long-term DOS of contract interactions. For example, a percentage of the total borrowed amount could be imposed as a fee.

Team response

Reject Medium, Accept Low or Non-Severity. This is a griefing vector, this could also happen on a protocol like Aave V3 (see hyperlink). There is a gas cost to the griefing attacker to do this already as well. The attacker would also need deposits >> total available to borrow, so there are also capital requirements, or they would need to flash loan which would incur more gas fees paid.

Our contracts are upgradeable, so if this becomes a serious problem then we will upgrade the contract with additional logic to stop this griefing vector, but we feel adding additional logic for this would not be necessary as there are high costs associated with it. If borrowing + repaying every single block, assuming $10 tx fee and 12 second block time, this would be $72,000 in gas fees per day for no gain to the griefer. No changes to code made.

Trust Security Response

It is true the attack is costly to run on every block for days. However, an attack on specific targets or a partial attack is still effective and unlocks temporary DOS or griefing impact. For AAVE the amount required for the attack is many millions of dollars, for Omni this would be much smaller to begin with, making it practical. Medium severity is justified.

TRST-M-3 If borrowFactor is updated to 0, the borrower will not be able to be liquidated

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

Description

In *_evaluateAccountInternal()*, if the **borrowFactor** of the borrowed asset is 0, the function will revert due to the divide by 0 error.

```
            if (borrowAmount != 0) {
                ++eval.numBorrow;
                uint256 borrowValue = (borrowAmount * price) / PRICE_SCALE; // Rounds
down
                eval.borrowTrueValue += borrowValue;
                uint256 borrowFactor = marketCount == 1
                    ? SELF_COLLATERALIZATION_FACTOR
                    :           _account.modeId           ==           0           ?
uint256(marketConfiguration_.borrowFactor) : uint256(mode.borrowFactor);
                eval.borrowAdjValue   +=   (borrowValue   *   FACTOR_PRECISION_SCALE)   /
borrowFactor; // Rounds down
            }
```

Although assets with **borrowFactor** of 0 will revert when borrowed, if the **borrowFactor** is updated to 0 by *setMarketConfiguration()*, the borrower will not be able to be liquidated due to the fact that *_evaluateAccountInternal()* will always revert.

Recommended mitigation

It is recommended to handle the case where **borrowFactor** is 0 in *_evaluateAccountInternal()* to prevent the function from reverting.

```
            if (borrowAmount != 0) {
                ++eval.numBorrow;
                uint256 borrowValue = (borrowAmount * price) / PRICE_SCALE; // Rounds
down
                eval.borrowTrueValue += borrowValue;
                uint256 borrowFactor = marketCount == 1
                    ? SELF_COLLATERALIZATION_FACTOR
                    :           _account.modeId           ==           0           ?
uint256(marketConfiguration_.borrowFactor) : uint256(mode.borrowFactor);
+               if(borrowFactor == 0) continue;
                eval.borrowAdjValue   +=   (borrowValue   *   FACTOR_PRECISION_SCALE)   /
borrowFactor; // Rounds down
            }
```

Also, to prevent users from borrowing assets with borrowFactor 0, require that the **borrowFactor** of the borrowed asset is not 0 in *borrow()*.

Team response

Acknowledged.

TRST-M-4 OmniToken is not compatible with rebase tokens

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

Description

OmniToken uses **totalDepositAmount** to store the token amount deposited by users. However, for rebasing tokens, the rebase token balance changes with the rebase event, which leads to inconsistency between **totalDepositAmount** and the actual token balance in *OmniToken*, and thus leads to the token amount withdrawn by the user to be different.

Recommended mitigation

It is recommended to document that rebase tokens are not supported.

Team response

Acknowledged.

TRST-M-5 Bad debt users can make small deposits to block socializeLoss() call

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

Description

The call to *socializeLoss()* requires the bad debtor's deposit to be less than 0.001% of the debt ($10 for $1M), in order to ensure that the user has been fully liquidated. However, a bad debtor can make a small deposit to prevent *socializeLoss()* from being called.

Consider the following scenario, Alice is currently fully liquidated but still has a bad debt of 10k USD, at this point Alice deposits 0.1 USD of collateral, which will prevent *socializeLoss()* from being called. Since Alice's balance is so small, the gas paid by the liquidator to liquidate Alice is much larger than 0.1 USD, which does not give the liquidator enough incentive to initiate liquidation. As a result, the Owner has to spend gas to liquidate Alice so that socializeLoss() can be called successfully.

Recommended mitigation

It is recommended to increase the required percentage in *socializeLoss()*, for example to 1%.

Team response

Reject. This issue doesn't get solved by adjusting percentage, as the bad actor could always just deposit slightly more or if the bad debt is small can post higher amounts. The reason we have the percentage there is more so to ensure for the administrator that the account has been fully liquidated before calling *socializeLoss()*. If the bad debt user decides to deposit more, we will just liquidate them again. Additionally, the *socializeLoss()* method is a privileged method, and in reality the administrator calling socializeLoss should often call liquidate() before socializeLoss() to make sure that the position is fully liquidated via a script.

Trust Security Response

While adjusting the percentage does not completely solve the griefing issue, it makes it expensive enough to definitely not be worthwhile. We believe the current impact justifies Medium severity, and off-chain setups should not be used to affect severity.

TRST-M-6 Band Oracle may return stale data

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

Description

According to Band docs, *getReferenceData()* will return **rate**,**lastUpdatedBase**,**lastUpdatedQuote**, where **rate** represents the exchange rate of base/quote, **lastUpdatedBase** represents the last update time of the base price, and **lastUpdatedQuote** represents the last update time of the quote price.

However, in *getPrice()*, only **lastUpdatedBase** is checked to prevent the price from being stale, not **lastUpdatedQuote**, which can only guarantee that the base price is fresh, but cannot guarantee that the quote price is fresh, and the stale quote price will lead to the **rate** being stale.

Recommended mitigation

It is recommended to check if **lastUpdatedQuote** is timed out in *getPrice()*. Since quote is always **USD**, a fixed **delay** can be used to check it.

Team response

Fixed.

Mitigation Review

In the fix, the same **delay** is used to check both **lastUpdatedQuote** and **lastUpdatedBase**, which results in the **delay** needing to be the greater of the two heartbeats for the function to work correctly, but also introduces the risk of returning stale data. We recommend using a different fixed **delay** for checking **lastUpdatedQuote.**

Mitigation Review #2

The fix added **delayQuote** for checking **lastUpdatedQuote**.

TRST-M-7 Liquidator may suffer loss due to being unable to seize sufficient collateral

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

Description

In liquidation, the protocol calculates the token amount seized from the liquidated person based on the token amount repaid by the liquidator.

```
        uint256 amount =
            IOmniToken(_params.liquidateMarket).repay(_params.targetAccountId,
msg.sender, borrowTier, _params.amount);
        (uint256       liquidationBonus,        uint256        softThreshold)        =
getLiquidationBonusAndThreshold(
            evalBefore.depositAdjValue,                     evalBefore.borrowAdjValue,
_params.collateralMarket
        );
        { // Avoid stack too deep
            uint256                            borrowPrice                            =
IOmniOracle(oracle).getPrice(IWithUnderlying(_params.liquidateMarket).underlying());
            uint256                            depositPrice                           =
IOmniOracle(oracle).getPrice(IWithUnderlying(_params.collateralMarket).underlying());
            uint256 seizeAmount = Math.ceilDiv(
                Math.ceilDiv(amount      *      borrowPrice,      depositPrice)      *
(LIQ_BONUS_PRECISION_SCALE   +   liquidationBonus),   //   Need   to   add   base   since
liquidationBonus < LIQ_BONUS_PRECISION_SCALE
                LIQ_BONUS_PRECISION_SCALE
            ); // round up
            seizedShares = IOmniTokenBase(_params.collateralMarket).seize(
                _params.targetAccountId, _params.liquidatorAccountId, seizeAmount
            );
        }
```

However, *OmniToken.seize()* does not guarantee that the token amount seized will be sufficient, i.e., the token amount seized will be the minimum of **_amount** and the balance of the liquidated person.

```
    function seize(bytes32 _account, bytes32 _to, uint256 _amount)
        external
        override
        nonReentrant
        returns (uint256[] memory)
    {
        require(msg.sender == omniPool, "OmniToken::seize: Bad caller");
        accrue();
        uint256 amount_ = _amount;
        uint256[] memory seizedShares = new uint256[](trancheCount);
        for (uint8 ti = 0; ti < trancheCount; ++ti) {
            uint256 totalShare = tranches[ti].totalDepositShare;
            uint256 totalAmount = tranches[ti].totalDepositAmount;
            uint256 share = trancheAccountDepositShares[ti][_account];
            uint256 amount = (share * totalAmount) / totalShare;
            if (amount_ > amount) {
                amount_ -= amount;
                trancheAccountDepositShares[ti][_account] = 0;
                trancheAccountDepositShares[ti][_to] += share;
                seizedShares[ti] = share;
            } else {
                uint256 transferShare = (share * amount_) / amount;
                trancheAccountDepositShares[ti][_account] = share - transferShare;
                trancheAccountDepositShares[ti][_to] += transferShare;
                seizedShares[ti] = transferShare;
                break;
            }
        }
        emit Seize(_account, _to, _amount, seizedShares);
        return seizedShares;
    }
```

Consider the following scenario.

Alice deposits 5000 USDC and 5000 USDT and borrows 8000 USD of WETH.

WETH appreciates to 9000 USD and Alice defaults.

Bob and Charlie both want to repay the WETH and receive USDC, but when the transaction is executed, Bob repays 4500 USD of WETH and receives 5000 USDC, Charlie repays 4500 USD of WETH but receives 0 USDC since Alice has 0 USDC balance.

Recommended mitigation

It is recommended to add **minOutAmount** parameter to *OmniToken.seize()* to allow the liquidator to perform slippage control.

Team response

Reject High, Accept Low. This shouldn't be listed as an issue as we intentionally did this. This is for the situations where someone can selflessly liquidate, i.e. liquidate without reward or for partial reward. We opt to greedily take as much as we can, rather than forcing an amount to exist. For example, the protocol or some other incentivized party would be able to liquidate

without claiming reward. Additionally, regarding using something like a **minOutAmount**, if a liquidator wants to ensure they get a correct amount of shares, we return the **seizedShares**, so that the liquidator can perform whatever necessary checks afterwards on the amount, such as checking that it exceeds a minimum, and revert in their custom contract if necessary. We have expectations that liquidators are capable of writing smart contracts, which is reasonable based on experience. We add additional comments in the docstring for *liquidate()*, and will also add this information in our documentation for further clarity to liquidators. No changes to code made.

Trust Security Response

The team documented this issue here. As for severity, we would reconsider it to be Medium, since *liquidate()* does return the share amounts, some amount of fault can be moved to the user.

Low severity findings

TRST-L-1 Inconsistent expiration state when expirationTimestamp == block.timestamp

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

Description

When **expirationTimestamp == block.timestamp**, it is treated as expired in some functions like *_evaluateAccountInternal()*, but it is treated as unexpired in some other functions like *liquidate()*.

Recommended mitigation

It is recommended to make the expiration state consistent when **expirationTimestamp == block.timestamp**.

Team response

Fixed.

Mitigation Review

The fix has been applied correctly.

TRST-L-2 Users depositing during pause state may suffer losses

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

Description

If bad debt arises after liquidation, the protocol pauses and waits for *socializeLoss()* to be called to cover the bad debt with the user's deposits. Withdrawing and borrowing are paused during the pause, but deposits are not paused, leading to the possibility that users who deposit during the pause may suffer losses due to the *socializeLoss()* call.

Recommended mitigation

It is recommended to disallow depositing during pause.

```
    function deposit(uint96 _subId, uint8 _trancheId, uint256 _amount) external
nonReentrant returns (uint256 share) {
+        require(_trancheId < IOmniPool(omniPool).pauseTranche(), "OmniToken::deposit:
Tranche paused.");
        require(_trancheId < trancheCount, "OmniToken::deposit: Invalid tranche id.");
        accrue();
```

Team response

Acknowledged.

TRST-L-3 No margin between max borrowing and liquidation thresholds

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

Description

When borrowing, the protocol requires that **eval.depositAdjValue >= eval.borrowAdjValue** after borrowing, and when **evalBefore.depositAdjValue < evalBefore.borrowAdjValue** the user can be liquidated.

Since there is no margin between the max borrow and the liquidation threshold, the user with the max borrow may be liquidated due to slight fluctuations in the asset price.

Recommended mitigation

It is recommended to keep a margin between the max borrowing and the liquidation thresholds. For example after borrowing require **eval.depositAdjValue * 0.95 >= eval.borrowAdjValue** instead of **eval.depositAdjValue >= eval.borrowAdjValue**.

Team response

Acknowledged

Additional recommendations

Limiting the updates to the old config in setMarketConfiguration()

setMarketConfiguration() is used to configure the market and can update old configurations. Currently *setMarketConfiguration()* does not have any restrictions on updates, which can lead to incorrect updates such as modifying the market's **isIsolatedCollateral**, modifying **riskTranche**, etc. It is recommended to limit the update in *setMarketConfiguration()*.

Centralization risks

CR-1 MARKET_CONFIGURATOR_ROLE can arbitrarily set the collateralFactor and borrowFactor of the market configuration

The **collateralFactor** and **borrowFactor** of the market configuration determines the health of the user's account. If **MARKET_CONFIGURATOR_ROLE** decreases the **collateralFactor** and **borrowFactor**, it will cause the healthy account to become unhealthy and be liquidated.

CR-2 setModeConfiguration() does not check for duplicate markets

setModeConfiguration() does not check for duplicate markets in **_modeConfiguration**, if there are duplicate markets in **_modeConfiguration**, it will result in repeated counting of user's deposits.

Systemic risks

Oracles must be trusted to report correct prices

The protocol uses Chainlink and Band oracles to get the prices of *OmniToken'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 borrowing more assets.