TaoBank Security Audit

Report Version 1.0.0

KukerLabs

Contents

1. About

1.1 Kuker Labs

At Kuker Labs, we're a dedicated group of independent researchers specializing in smart contract security. With a wealth of experience conducting security reviews and uncovering numerous vulnerabilities in live smart contracts, we've safeguarded millions in Total Value Locked. Our commitment is to provide DeFi protocols with high-quality security services. For inquiries about security audits, feel free to contact us via Telegram or Twitter at @kukerlabs.

1.2 TaoBank

TaoBank, under the leadership of Taolord and Athena Nodes, specializes in providing a lending platform exclusively for $TAO holders, enabling them to use their digital assets as collateral for loans without selling them. This approach provides immediate liquidity while maintaining ownership, enhancing the utility and value of $TAO. The protocol is known for its high security, efficiency, and a community-centric governance model that involves users in its development. Learn more at https://docs.taobank.ai.

2. Disclaimer

Audits require a significant investment of time, resources, and specialized expertise. Trained professionals employ a mix of automated tools and manual inspection to thoroughly evaluate smart contracts, aiming to uncover as many vulnerabilities as possible. While audits can uncover existing vulnerabilities, they cannot assure their complete absence.

3. Risk Classification

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

3.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - involves a small loss of funds or affects a core functionality of the protocol.
- **Low** - encompasses any unexpected behavior that's non-critical.

3.2 Likelihood

- **High** - a direct attack vector; the cost is relatively low compared to the potential loss of funds.
- **Medium** - only a conditionally incentivized attack vector, with a moderate likelihood.
- **Low** - involves too many or unlikely assumptions; offers little to no incentive.

3.3 Actions required by severity level

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

4. Executive Summary

The team at Kuker Labs was engaged with reviewing TaoBank's smart contracts from March 30, 2024, to April 7, 2024.

4.1 Overview

Project Name	TaoBank
Repository	https://github.com/TaoBank/tbank-protocol-v2
Commit Hash	6346590f814f44119b2f6642aeadc83c67f19a95
Methods	Manual Review

4.2 Scope

All files in the `contracts` directory were reviewed.

4.3 Issues Found

Severity	Count
High Risk	1
Medium Risk	9
Low Risk	12
Informational	12
Gas Optimization	4

5. Findings

5.1 High Risk

[H-01] Risk of auction duration modification in `AuctionManager`

Impact
Severity: High
Likelihood: Medium

Context
AuctionManager::bid()
AuctionManager::bidInfo()
AuctionManager::setAuctionDuration()

Description
The `AuctionManager::setAuctionDuration()` function enables an admin to alter the duration of Dutch auctions. However, this flexibility poses risks to existing auctions. Modifying the auction duration mid-process could cause auctions to revert, execute at unfavorable prices, or conclude at higher prices, potentially discouraging users from participating in liquidations.

Recommendation
To mitigate these risks, implement caching of the `auctionDuration` at the time of auction creation in the `auctionData` struct. Utilize this cached variable within `AuctionManager::bidInfo()`

and AuctionManager::bid(). This approach ensures that changes to the global auctionData won't impact existing auctions.

```
struct auctionData {
+   uint256 auctionDuration;
    uint256 originalDebt;
    uint256 lowestDebtToAuction;
    uint256 highestDebtToAuction;
    uint256 collateralsLength;
    address[] collateral;
    uint256[] collateralAmount;
    uint256 auctionStartTime;
    uint256 auctionEndTime;
    bool auctionEnded;
}

auctions.push(
    auctionData({
+       auctionDuration: auctionDuration,
        originalDebt: _debtToAuction,
        lowestDebtToAuction: _lowestDebtToAuction,
        highestDebtToAuction: _highestDebtToAuction,
        collateralsLength: _collateralsLength,
        collateral: _collaterals,
        collateralAmount: _collateralAmounts,
        auctionStartTime: _auctionStartTime,
        auctionEndTime: _auctionEndTime,
        auctionEnded: false
    })
);

uint256 _debtToAuctionAtCurrentTime = _highestDebtToAuction -
    ((_highestDebtToAuction - _lowestDebtToAuction) *
        (block.timestamp - _auction.auctionStartTime)) /
-   auctionDuration;
+  _auction.auctionDuration;
```

PoC

To verify the issue, follow this guide and include this test in the codebase. The test demonstrates how the protocol accrues bad debt by the inability to liquidate a vault when there is a huge change in the auction duration as well as by executing a liquidation at prices lower than the lowestDebtToAuction.

5.2 Medium Risk

[M-01] Possibility of a reorg attack

Impact
Severity: High
Likelihood: Low

Context
VaultDeployer::deployVault()

Description
The `VaultFactory::createVault()` function creates a new vault and returns its address. However, the address of the newly created vault is determined using the `VaultDeployer::deployVault()` function, which deploys the vault using the `CREATE` opcode. This opcode generates the contract address based on the contract creator's address and nonce.

```
function deployVault(
    address _factory,
    address _vaultOwner,
    string memory _name
) external returns (address) {
    // Deploy a new instance of the Vault contract
@>  Vault vault = new Vault( // @audit CREATE opcode
        _factory,
        _vaultOwner,
        _name,
        vaultExtraSettings
    );
    return address(vault);
}
```

The predictable nature of this address generation method (based on nonce) poses a risk, particularly in the event of blockchain reorganizations (reorgs).

During a reorg, transactions (including contract creations) might be rolled back, resetting the nonce. An attacker observing the network could exploit this by deploying their own contract with the same address during a reorg, especially in networks susceptible to reorgs, and intercept funds or interactions meant for the original contract.

Arbitrum rollups (Optimism/Arbitrum/Polygon) are suspect to reorgs since if someone finds a fraud the blocks will be reverted, even though the user receives a confirmation and already created a vault.

Attack scenario:

1. Alice deploys a vault and sends funds to it via `VaultFactory::addCollateralNative()`
2. Bob observes a network block reorg and calls `VaultFactory::createVault()`, creating a vault with an address to which Alice sends funds.
3. Alice's transactions are executed, and she transfers funds to Bob's controlled vault.

Recommendation

To mitigate this risk, it is recommended to use the `CREATE2` opcode for deploying new vault contracts. The `CREATE2` opcode allows for a more unpredictable and safer contract address generation by including a user-provided salt value along with the deployer's address and nonce.

[M-02] User can lose money paying someone elses debt

Impact

Severity: High

Likelihood: Medium

Context

VaultFactory::repay()

Description

The repay function lacks checks to ensure that the caller is the vault owner, allowing any user to repay any vault's debt. This can result in users paying off debts for others without receiving any collateral in return.

```
function repay(address _vault, uint256 _amount) external {
    require(containsVault(_vault), 'vault-not-found');
    totalDebt -= _amount;
    Vault(_vault).repay(_amount);

    IMintableToken(stable).safeTransferFrom(
        _msgSender(),
        address(this),
        _amount
    );
    IMintableToken(stable).burn(_amount);
}
```

Recommendation

Add an `onlyVaultOwnerOrOperator(_vault)` modifier to restrict repayments to the owner or authorized users only.

```
- function repay(address _vault, uint256 _amount) external {
+ function repay(address _vault, uint256 _amount) external
    onlyVaultOwnerOrOperator(_vault) {
     require(containsVault(_vault), 'vault-not-found');
     totalDebt -= _amount;
     Vault(_vault).repay(_amount);

     IMintableToken(stable).safeTransferFrom(
         _msgSender(),
         address(this),
         _amount
     );
     IMintableToken(stable).burn(_amount);
  }
```

[M-03] Chainlink price feed good practices

Impact
Severity: High
Likelihood: Low

Context
ChainlinkPriceOracle::price()

Description
Given that the protocol will be deployed to Arbitrum, it's essential to implement checks for the sequencer being down. Consider the scenario where the sequencer experiences downtime — this has occurred in the past and may happen again in the future. When the sequencer resumes operation and oracles update their prices, all price movements that occurred during the downtime are applied simultaneously. If these movements are significant, they may lead to chaos, with borrowers rushing to save their positions while liquidators rush to liquidate them. Since liquidations are primarily handled by bots, borrowers are likely to face mass liquidations, which is unfair to them as they couldn't act on their positions during the L2 downtime. Hence, it would be ideal if the protocol provides borrowers with a grace period once the sequencer returns.

Chainlink provides a dedicated feed for sequencer uptimes. This feed continuously updates sequencer uptime from L1 to L2. When the sequencer goes down, the Chainlink sequencer update is pushed to the delayed inbox. Since all transactions in the delayed inbox are executed before any others once the sequencer resumes operation, your contract would consider a grace period.

Aave V3 follows a similar approach as described in their technical paper, bearing in mind that heavily undercollateralized (0.95 < HF < 1) positions may still experience liquidations even during the grace period.

Recommendation

Familiarize yourself with the Aave V3 solution to the problem. Example code snippet:

```
(
    /*uint80 roundID*/,
    int256 answer,
    uint256 startedAt,
    /*uint256 updatedAt*/,
    /*uint80 answeredInRound*/
) = sequencerUptimeFeed.latestRoundData();

// Answer == 0: Sequencer is up
// Answer == 1: Sequencer is down
bool isSequencerUp = answer == 0;
if (!isSequencerUp) {
    revert SequencerDown();
}

// Ensure the grace period has passed after the
// sequencer is back up.
uint256 timeSinceUp = block.timestamp - startedAt;
if (timeSinceUp <= GRACE_PERIOD_TIME) {
    revert GracePeriodNotOver();
}
```

Furthermore, there is a possibility in the future that access to Chainlink oracles will be restricted to paying customers (Chainlink is currently subsidized and has no monetization model in place for its most used feeds). Hence, it's advisable to surround the `latestRoundData()` call in a try-catch statement to gracefully recover from errors. Additionally, having a fallback option for oracles is recommended to avoid a single point of failure. Secondary oracle options such as Uniswap TWAP for assets with good liquidity on Uniswap or another off-chain oracle like Tellor can be used.

[M-04] No check for sequencer uptime can lead to dutch auctions executing at bad prices

Impact

Severity: Medium

Likelihood: Low

Context

AuctionManager::bid()

Description

There is no check for sequencer uptime, which could lead to auctions executing at unfavorable prices for the protocol. While the auctions are automatically closed after 2 hours (`auctionDuration`), a dutch auction which starts at `_highestDebtToAuction` and ends at `_lowestDebtToAuction`, in case of a sequencer outage, would never had a chance to fill at a higher price during the outage. While this doesn't result in bad debt for the protocol, it's beneficial for auctions to execute at higher prices during stable market conditions, as more stable tokens are removed from circulation.

1. Alice borrows 2500 stable coins (maximum borrowable) in exchange for 1 WETH worth $3000.
2. A small price movement decreases her health factor to just below 1e18 (approximately ~0.99e18), making her position liquidatable.
3. A 2-hour Dutch auction is initiated for her collateral, but a sequencer outage occurs simultaneously. During the 1.5-hour outage, the price decreases.
4. When the sequencer comes online, Bob can redeem Alice's collateral for a lower price.
5. Fewer stable coins are burnt compared to if the sequencer had not gone offline.

Recommendation Implement a check for sequencer uptime and invalidate the auction if the sequencer was down during the auction period, provided the position's health factor hasn't significantly decreased. In volatile times, it's preferable to liquidate positions quickly rather than restart auctions to avoid the protocol incurring underwater debt.

[M-05] Minimal borrowed amounts can grief protocol, leading to underwater debt

Impact

Severity: Medium

Likelihood: Medium

Context

StabilityPool::_computeRewardsPerUnitStaked()

Description

While calculating the `stableCoinLossNumerator` on L833, `lastStableCoinLossErrorOffset` is subtracted due to the fact that the result of `stableCoinLossPerUnitStaked` is rounded up, but `lastStableCoinLossErrorOffset` being scaled to 1e18 precision can lead to underflows for vaults with low borrowed amount, thus making the vault non-liquidatable and yielding underwater debt for the protocol. This opens the door for malicious users to grief the protocol by opening really small positions.

```
/* code */

if (_debtToOffset == _totalStableCoinDeposits) {
    stableCoinLossPerUnitStaked = DECIMAL_PRECISION; // When the Pool
        depletes to 0, so does each deposit
    lastStableCoinLossErrorOffset = 0;
} else {
    uint256 stableCoinLossNumerator = _debtToOffset *
@>      DECIMAL_PRECISION - lastStableCoinLossErrorOffset; // @audit
    underflow
    /*
     * Add 1 to make error in quotient positive. We want "slightly too
         much" StableCoin loss,
     * which ensures the error in any given compoundedStableCoinDeposit
         favors the Stability Pool.
     */
    stableCoinLossPerUnitStaked =
        stableCoinLossNumerator /
        _totalStableCoinDeposits +
        1;
    lastStableCoinLossErrorOffset =
        stableCoinLossPerUnitStaked *
        _totalStableCoinDeposits -
        stableCoinLossNumerator;
}

/* code */
```

Recommendation

Consider enforcing a minimum borrowable amount in `VaultFactory::borrow()`

```
function borrow(
    address _vault,
    uint256 _amount,
    address _to
) external onlyVaultOwnerOrOperator(_vault) {
    require(containsVault(_vault), 'vault-not-found');
    require(_to != address(0x0), 'to-is-0');
+   require(_amount >= MIN_BORROWABLE_AMOUNT, 'below-minimum-amount');

/ * code */
}
```

PoC

To verify the issue, follow this guide and include this test in the codebase. The test demonstrates the impossibility of liquidating a vault with small debt.

[M-06] The liquidation auction doesnt allow bidders to specify maximum bid price

Impact
Severity: High
Likelihood: Low

Context
AuctionManager::bid()

Description
`AuctionManager::bid()` doesn't allow the liquidator to specify a max price they are willing to pay for the collateral they are liquidating. On the surface, this doesn't seem like an issue because the price is always decreasing due to the Dutch auction. However, this can be problematic if the chain the contracts are deployed on suffers a reorg attack. This can place the transaction earlier than anticipated and therefore charge the user more than they meant to pay. While this scenario is unlikely on Ethereum, the protocol is intended to be deployed on Arbitrum, which is more susceptible to reorganization.

Recommendation

```diff
- function bid(uint256 _auctionId) external nonReentrant {
+ function bid(uint256 _auctionId, uint256 _maxBid) external
   nonReentrant {
    /* code */

    uint256 _debtToAuctionAtCurrentTime = _highestDebtToAuction -
        ((_highestDebtToAuction - _lowestDebtToAuction) *
            (block.timestamp - _auction.auctionStartTime)) /
        auctionDuration;

+   require(_debtToAuctionAtCurrentTime <= _maxBid, 'max-bid-exceeded')
   ;

    /* code */
```

[M-07] Dutch auction poses disadvantages for bidders

Impact
Severity: Medium
Likelihood: Medium

Context

AuctionManager::setLowestHealthFactor()

AuctionManager::newAuction()

Description

During the initiation of a new Dutch auction for liquidation, the highest debt to be auctioned is set to be equal to the vault's stable coin debt:

```
uint256 _highestDebtToAuction = _debtToAuction;
```

and the lowest debt to be auctioned is equal to a percentage of the value of the collateral, calculated as follows:

```
uint256 _lowestDebtToAuction = (_totalCollateralValue * lowestHF) /
    DECIMAL_PRECISION;
```

When the health factor of a vault becomes `< 1e18`, the vault is liquidateable. However as the vault is overcollateralized (MCR > 100%) and there aren't significant price swings, the collateral remains worth more than the debt.

The `lowestHF`, representing a percentage of the collateral value, is initialized with a value of `1.05e18` (105%). The setter function setLowestHealthFactor() for this variable only checks whether the value is more than 0. Allowing `lowestHF` to be `>= 1e18` (>= 100%), combined with an if-statement that reverses the price bounds (underflow prevention) and collateral already being worth more than the debt, causes the auction price to be more expensive than the collateral, resulting in bidders incurring losses instead of profit.

```
if (_highestDebtToAuction < _lowestDebtToAuction) {
    uint256 _debtToAuctionTmp = _lowestDebtToAuction;
    _lowestDebtToAuction = _highestDebtToAuction;
    _highestDebtToAuction = _debtToAuctionTmp; // @audit this is bigger
        than the actual debt
}
```

In scenarios where there are small price movements and a non-volatile market, and the health factor just dips under `1e18`, the auction becomes unprofitable at the start and gradually becomes cheaper than the collateral value over time. However, in significant price swings and volatile markets, the lowest auction price remains higher than the collateral value, leading to bad debt.

Additionally, when `lowestHF` is `== 1e18`, the auction's price remains constant and does not change, rendering it ineffective as an auction mechanism.

Recommendation

Initialize `lowestHF` with a value < `1e18`, for example:

```
uint256 public lowestHF = 0.95 ether; // 95%
```

Modify `setLowestHealthFactor()` as follows:

```
function setLowestHealthFactor(uint256 _lowestHF) external onlyOwner {
    require(_lowestHF > 0 && _lowestHF < 1e18, 'lowest-hf-is-0');
    lowestHF = _lowestHF;
}
```

PoC

To verify the issue, follow this guide and include this test in the codebase. This test demonstrates how even with a small price movement, the auction remains unprofitable for approximately 40 minutes out of a 2-hour-long auction, potentially resulting in bad debt if the price drops further during these 40 minutes.

[M-08] Incorrect borrow rate calculation leads to user funds loss

Impact

Severity: Medium

Likelihood: High

Context

VaultBorrowRate::getBorrowRate()

Description

The `VaultBorrowRate::getBorrowRate()` function is accountable for calculating the weighted borrow rate, but has a flaw when calculating the collateral value on L37 which arises when there are multiple collateral tokens, including one with less than 18 decimals (e.g., `USDC` with 6 decimals) and another with 18 decimals (e.g., `WETH`). The calculation on L35 performs normalization of the collateral amount to 18 decimal places. However, the subsequent calculation on L37 does not appropriately adjust for tokens with fewer decimals, resulting in skewed borrow rate calculations due to having `10 ** _priceFeed.decimals(_collateralAddress)` in the denominator instead of `1e18`.

Consider this scenario:

1. Alice deposits 1 USDC (9.9% BR) and 50 WETH (1% BR), worth $1 and $150,000, respectively. It's obvious that WETH's weight is magnitudes bigger than the USDC's.
2. But due to dividing the collateral value by 1e6 instead of 1e18, the USDC's weight appears to be much bigger, thus leading to the overall borrow rate being ~9.9% rather than ~1%.
3. Alice borrows stable tokens but receives less than the expected amount as the fee is much bigger now.

```
function getBorrowRate(
    address _vaultAddress
) external view returns (uint256) {
 // ** code **
        uint256 _normalizedCollateralAmount = _collateralAmount *
            (10 ** (18 - _priceFeed.decimals(_collateralAddress)));
        uint256 _collateralValue = (_normalizedCollateralAmount *
            _price) /
 @>         (10 ** _priceFeed.decimals(_collateralAddress)); // @audit
    should divide by 1e18
        uint256 _weightedFee = (_collateralValue * _borrowRate) / 1e18;
// ** code **
}
```

Recommendation

Change the denominator on L37 to be `1e18` as follows:

```
uint256 _collateralValue = (_normalizedCollateralAmount * _price) / 1
    e18;
```

PoC

To verify the issue, follow this guide and include this test in the codebase. This test highlights discrepancies in received funds due to the inaccurate borrowing rate calculation.

[M-09] First staker can drain `TBANKStaking` contract

Impact

Severity: High

Likelihood: Low

Context

TBANKStaking::stake()

Description

In the TBANKStaking::stake() function, there is a vulnerability that arises when TBANKStaking::takeFees() is called with an amount of stable coins to be distributed, and the staking contract has no TBANK staked. In this scenario, the F_StableCoin variable is incremented. However, upon a subsequent stake, the totalTBANKStaked is checked in order to give the entire previously accumulated F_StableCoin to the first staker. The issue lies in the fact that stableCoinUserGains[msg.sender] is incremented, allowing a staker to unstake their funds and stake them again to accumulate the same amount repeatedly. This opens the door for malicious users to repeatedly unstake and restake their funds, accumulating a significant value in stableCoinUserGains[msg.sender]. Subsequently, when there are new calls to TBANKStaking::takeFees(), thus enough taoUSD in the contract, the malicious user can drain the entire contract, leaving other users without any accumulated rewards.

```
function stake(uint256 _tbankAmount) external {
    // ** code **
    // Grab and record accumulated StableCoin gains from the current
       stake and update Snapshot.
    uint256 currentTotalTBANKStaked = totalTBANKStaked;
    if (currentTotalTBANKStaked == 0)
@>      stableCoinUserGains[msg.sender] += F_StableCoin; // @audit can
   be exploited by unstaking
    _updateUserSnapshot(msg.sender);
   // ** code **
}
```

Recommendation

Adjust L109 accordingly:

```
stableCoinUserGains[msg.sender] += (F_StableCoin -
   F_StableCoinSnapshots[msg.sender]);
```

Utilizing this adjustment prevents the accumulation of rewards by restaking and avoids losing already accrued rewards when a staker has unstaked all of their funds and then restaked again.

PoC

To verify the issue, follow this guide and include this test in the codebase. The test demonstrates how a malicious user is able to drain the staking contracts, leaving others with no way to retrieve their funds.

5.3 Low Risk

[L-01] Incompliance With ERC-20 Standard

Context

Stabilizer::constructor()
TokenToPriceFeed::setTokenPriceFeed()

Description

The `decimals()` and `symbol()` function are part of the OpenZeppelin's IERC20Metadata, but aren't originally included in the official ERC-20 standard. Therefore, indiscriminately casting all tokens to this interface and subsequently invoking this function can fail.

[L-02] OpenZeppelin ERC20 `safeApprove()` is deprecated

Context

Vault::liquidate()
VaultFactoryZapper::createVault()
LiquidationRouter::addSeizedCollateral()
AuctionManager::_transferToLastResortLiquidation()

Description

The use of `safeApprove()` has been marked as deprecated, thus is recommended to be replaced with `safeIncreaseAllowance()` and `safeDecreaseAllowance()`.

[L-03] `Ownable2Step` is preferred over `Ownable`

Context

TokenToPriceFeed::transferOwnership()
MintableTokenOwner::transferTokenOwnership()

Description

Ownable2Step places some important checks, such as a 2-step ownership transfer procedure and should be preferred over Ownable.

[L-04] ERC20 tokens with more than 18 decimals cause reverts

Context

AuctionManager::getTotalCollateralValue()
Stabilizer::constructor()
Vault::calcRedeem()

Description

While the protocol is compatible with tokens containing 18 decimals or fewer, it may encounter issues with tokens having more than 18 decimals, such as `YAMv2` with 24 decimals.

[L-05] Use of abi.encodePacked() with dynamic types

Context

OwnerProxy::execute()
OwnerProxy::addPermission()

Description

Using `abi.encodePacked()` with dynamic types isn't advisable when feeding the outcome into a hashing function like `keccak256()`, due to hash collisions. Use `abi.encode()` instead.

[L-06] `StabilityPool` `stableCoin` variable not updated alongside factory

Context

StabilityPool::setVaultFactory()

Description

When changing the vault factory in `StabilityPool::setVaultFactory()`, the stableCoin variable is not updated alongside.

Recommendation

```
function setVaultFactory(address _factory) external onlyOwner {
    require(_factory != address(0x0), 'factory-is-0');
    factory = IVaultFactory(_factory);
+   stableCoin = IMintableToken(address(IVaultFactory(_factory).stable
   ()));
}
```

[L-07] Risk of locked funds inside `VaultFactoryZapper`

Context

VaultFactoryZapper::receive()

Description

The `receive` function allows users to send funds to the contract. However, there is no functionality provided to withdraw mistakenly sent funds. Therefore, having this function could lead to funds being permanently locked in the contract. As a precautionary measure, it's advisable to remove the `receive` function altogether.

[L-08] Missing events

Context

StabilityPool::setVaultFactory()
VaultExtraSettings::setRedemptionKickback()
VaultExtraSettings::setMaxRedeemablePercentage()
VaultFactoryZapper::setVaultFactory()
AuctionManager::setLowestHealthFactor()

Description Events must be emitted for all relevant state changes.

[L-09] Owner can renounce ownership

Description

If the project's roadmap doesn't anticipate relinquishing total ownership control, consider overriding OpenZeppelin's `Ownable` `renounceOwnership()` function to deactivate it. Otherwise, important functions can be left inaccessible in case of owner relinquishing.

[L-10] Known vulnerabilities in current version of OpenZeppelin contracts

Description

The current version of the contracts by OpenZeppelin (@openzeppelin/contracts) contains known vulnerabilities.

```
"@openzeppelin/contracts": "^4.9.0",
```

Recommendation

To address these issues and enhance the contract's security, it is advisable to update to at least version 5.0.1 of @openzeppelin/contracts.

[L-11] USDC blacklist functionality can lock depositor collateral rewards

Context

StabilityPool::_sendCollateralRewardsToDepositor()

Description

The `_sendCollateralRewardsToDepositor` function in the `StabilityPool` contract transfers collateral rewards to depositors. However, if a depositor has been blacklisted by a token with such functionality as `USDC` and one of the collateral rewards is `USDC`, the depositor won't be able to withdraw any rewards from the protocol. This is because the accrued rewards are stored in the `_depositorCollateralGains` array and are transferred in a for-loop.

Recommendation

To address this issue, add a destination `to` address parameter to the `_sendCollateralRewardsToDepositor` function. Then, modify the function to transfer the rewards to the specified `to` address instead of `msg.sender`.

```
function _sendCollateralRewardsToDepositor(
-    TokenToUint256[] memory _depositorCollateralGains
+    TokenToUint256[] memory _depositorCollateralGains,
+    address to
) internal {
    for (uint256 i = 0; i < _depositorCollateralGains.length; i++) {
        if (_depositorCollateralGains[i].value == 0) {
            continue;
        }
        IERC20 collateralToken = IERC20(
            _depositorCollateralGains[i].tokenAddress
        );
        collateralToken.safeTransfer(
-           msg.sender,
+           to,
            _depositorCollateralGains[i].value
        );
        emit CollateralRewardRedeemed(
-           msg.sender,
+           to,
            _depositorCollateralGains[i].tokenAddress,
            _depositorCollateralGains[i].value
        );
    }
}
```

[L-12] Unbounded view functions can fail

Context

VaultFactoryHelper::getProtocolTvl()
VaultFactoryHelper::getLiquidatableVaults()
VaultFactoryHelper::getRedeemableVaults()

Description

Some of the view functions in `VaultFactoryHelper` iterate over the vault count which can become a large number.

There are two potential limits to how much data a smart contract can return at once: gas limits and execution time. For a function call that's part of a transaction (e.g. from another smart contract), gas is often a limiting factor. Each byte of data returned from the call consumes gas, as does iterating through the data set. For view functions being called from outside the EVM (e.g. from JavaScript in a web app), gas is not a limiting factor because there is no transaction being executed. The node processing the call does the computation locally and returns the result. Each node gets to set its own processing limits - typically limiting execution time. If the call takes too long, it will fail.

Recommendation

Use cursor/pagination pattern.

5.5 Informational

[I-01] 1:1 ratio is assumed when minting and burning in `Stabilizer.sol`

Context

Stabilizer::mint()
Stabilizer::burn()

Description

The `Stabilizer` contract is designed to maintain the peg of the stable coin by minting and burning stable coins in exchange for collateral. However, these functions assume a 1:1 ratio between the value of the two tokens without validating their prices against an oracle. This vulnerability enables potential exploiters to steal significant amounts of funds from the protocol when volatile collateral is used. Consider the following scenario:

1. The `Stabilizer` contract is deployed with WETH as the collateral token, which has 18 decimals.
2. Alice holds $500 worth of stable coins (500e18), while the `Stabilizer` contract holds $3M worth of WETH (1000e18) (assuming 1 ETH = $3000).
3. Alice calls `Stabilizer.sol::burn(500e18)`, which transfers 475 WETH to her, resulting in a profit of approximately $1.42M for just $500 worth of stable coins (after considering a 5% fee to the protocol).

```
function burn(uint256 _amount) external {
    // mintableToken is 18 decimals, scale accordingly
@>  uint256 collateralAmount = _amount / scalingFactor; // @audit
   Assumes 1:1 ratio with both tokens having 18 decimals
    require(_amount >= scalingFactor, "amount-too-small");
    uint256 fee = (collateralAmount * feeBps) / 10000;

    stableToken.safeTransferFrom(msg.sender, address(this), _amount);
    stableToken.burn(_amount);

@>  collateralToken.safeTransfer(msg.sender, collateralAmount - fee);
   // @audit Transfers much more collateral than intended
    collateralToken.safeTransfer(feeRecipient, fee);

    emit StabilizerBurn(msg.sender, _amount, fee);
}
```

Recommendation

Implement an oracle to validate the prices of the provided tokens during both minting and burning operations and adjust accordingly based on their values.

PoC

To verify the issue, follow this guide and include this test in the codebase. The test demonstrates the issues encountered during minting and burning.

[I-02] The risks of relying on assert() in solidity contracts

Context

StabilityPool.sol

Description

In versions after 0.8.0, it's prudent to minimize the use of assert(). The Solidity documentation suggests that code should never reach a state where assert() is triggered. Instead, for better clarity and understanding of contract failures, developers are advised to utilize require() statements or custom error messages.

[I-03] Recommendation to use `uint48` for time-centric variables

Context

AuctionManager::auctionData

Description

For time-relevant variables, `uint32` might be restrictive as it ends in 2106, potentially posing challenges in the far future. On the other hand, using excessively large types like `uint256` is unnecessary. Thus, `uint48` offers a balanced choice for such use cases.

[I-04] Unresolved TODO and stale comments in the code

Context

StabilityPool::deposit()
StabilityPool::_updateDepositAndSnapshots()
StabilityPool::_getCollateralGainsArrayFromSnapshots()

Description
The presence of TODO and stale comments suggests that certain features might be incomplete or not ready for audit.

[I-05] Use of `override` is unnecessary

Context
TokenToPriceFeed::owner()
TokenToPriceFeed::transferOwnership()
VaultExtraSettings::setMaxRedeemablePercentage()

Description
Starting with Solidity version 0.8.8, using the override keyword when the function solely overrides an interface function, and the function doesn't exist in multiple base contracts, is unnecessary.

[I-06] Adopt named mappings for clarity

Description
To enhance code clarity, consider upgrading to Solidity version 0.8.18 or newer and employing named mappings. This approach clearly conveys the intent and usage of each mapping, making the codebase more comprehensible.

[I-07] Floating compiler version

Description
Contracts should be deployed with the same compiler version and flags that they have been tested the most with. Locking the pragma helps ensure that contracts do not accidentally get deployed using, for example, the latest compiler which may have higher risks of undiscovered bugs. Contracts may also be deployed by others and the pragma indicates the compiler version intended by the original authors.

[I-08] Use of named imports is advisable

Description
Using explicitly named imports is advisable for several reasons:

- Provides control over what is imported into the current scope.
- Helps avoid naming conflicts by importing only specific items.

- Promotes consistency and enforces coding style guidelines.
- Makes it easier to locate where a particular name is defined in the codebase.

[I-09] Non-compliance with function ordering in Solidity Style Guide

Description

For clarity and consistency, it's beneficial to adhere to the Solidity Style Guide which prescribes a specific order for function declarations: starting with `constructor()`, followed by `receive()`, `fallback()`, `external`, `public`, `internal`, and ending with `private`.

[I-10] If-statement never reached in `StabilityPool::_updateP()`

Context

StabilityPool::_updateP()

Description

The `_updateP` function is always called with its `loss` parameter being `true`, which means that `else` block is never reached.

```
  if (loss) {
      newProductFactor = uint256(
          DECIMAL_PRECISION - _stableCoinChangePerUnitStaked
      );
@> } else { // @audit this case is never hit
      newProductFactor = uint256(
          DECIMAL_PRECISION + _stableCoinChangePerUnitStaked
      );
  }
```

[I-11] `TBANKStaking::takeFees()` always returns `true`

Context

TBANKStaking::takeFees()

Description

The NatSpec of the function states that the returned results indicate whether the operation was successful, but the returned result is always `true`. Otherwise, the function will revert instead of returning `false`.

[I-12] Unused imports

Context

Stabilizer.sol

Description

Remove unused imports such as `import "hardhat/console.sol";`

5.4 Gas Optimization

[G-01] Missing `contributorDeposit` check in `StabilityPool::redeemReward()`

Context

StabilityPool::redeemReward()

Recommendation

Add this check in order to fail early and return unused gas to the user.

```
function redeemReward() external {
    Snapshots memory snapshots = depositSnapshots[msg.sender];
    uint256 contributorDeposit = deposits[msg.sender];
+   require(contributorDeposit > 0, "deposit-is-0");

    uint256 compoundedDeposit = _getCompoundedDepositFromSnapshots(
        contributorDeposit,
        snapshots
    );
    _redeemReward();
    _updateDepositAndSnapshots(msg.sender, compoundedDeposit);
}
```

[G-02] Store array length outside loops for efficiency

Context

Vault::collaterals()

LiquidationRouter::collaterals()

VaultFactoryHelper::getVaultTvl()

VaultFactoryHelper::getRedeemableVaults()

Description

When array length isn't cached, the Solidity compiler repeatedly reads it in every loop iteration. For storage arrays, this means an added sload operation costing 100 extra gas for each subsequent iteration. For memory arrays, it's an additional mload operation costing 3 extra gas post the first iteration.

[G-03] Revert strings are less gas-efficient than custom errors

Description

Consider opting for custom errors as they offer a more gas-efficient method to explain to users why an operation failed (~ 50 gas saved for each instance). Utilizing custom errors is also more cost-effective during deployment and allows for the inclusion of values like address, tokenID, and msg.value within its parentheses. Furthermore, custom errors enhance debugging capabilities and facilitate a detailed analysis of DApp reverts, especially on platforms like Tenderly.

[G-04] Increments/decrements can be unchecked in for-loops

Context

VaultBorrowRate::getBorrowRate()
LiquidationRouter::collaterals()

Description

In Solidity 0.8+, there's a default overflow check on unsigned integers. It's possible to uncheck this in for-loops and save some gas at each iteration, but at the cost of some code readability, as this uncheck cannot be made inline.

ethereum/solidity#10695

The change would be:

```
- for (uint256 i; i <  numIterations; i++) {
+ for (uint256 i; i <  numIterations;) {
 // ...
+   unchecked { ++i; }
}
```

These save around **25 gas saved** per instance. The same can be applied with decrements (which should use `break` when `i == 0`). The risk of overflow is non-existent for `uint256`.