Security Audit Report for WenCore

Date: Feb 2, 2024

Version: 1.0

Contact: contact@blocksec.com

BlockSec
Audit Team

Digitally signed by
BlockSec Audit Team
Date: 2024.02.08
12:16:07 +08'00'

Contents

Report Manifest

Item	Description
Client	Wen-Protocol
Target	WenCore

Version History

Version	Date	Description
1.0	February 2, 2024	First Version

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and released detailed analysis reports of high-impact security incidents. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is the code repository of WenCore[1] of Wen-Protocol.

The auditing process is iterative. Specifically, we will audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following. Our audit report is responsible for the only initial version (i.e., `Version 1`), as well as new codes (in the following versions) to fix issues in the audit report.

Project		Commit SHA
WenCore	`Version 1`	`b4f1a412874924cfcc66fb0b084779f9de575ad1`
	`Version 2`	`cab02c83243b3b86aa1e239d4dd1c86c6370af83`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.

[1] https://github.com/WENProtocol/WenCore/

- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Access control
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we find **eighteen** potential issues. Besides, we also have **two** recommendations and **one** note as follows:

- High Risk: 7
- Medium Risk: 10
- Low Risk: 1
- Recommendations: 2
- Note: 1

ID	Severity	Description	Category	Status
1	High	Incorrect Calculation of Staking Rewards in esWenstaking	DeFi Security	Fixed
2	Medium	Front-Running of Reward Distribution in submit()	DeFi Security	Confirmed
3	Medium	Improper Check of Input in setRewardEndTime()	DeFi Security	Fixed
4	Medium	Precision Loss of Rewards in claim()	DeFi Security	Fixed
5	High	Transferable esWen Token	DeFi Security	Fixed
6	High	Incapable Collateral Token within Protocol	DeFi Security	Fixed
7	High	Losses of Stakers in Stability Pool due to Flash Loan Liquidation	DeFi Security	Fixed
8	High	Incorrect Update of System Variable lastCollateralError_Offset	DeFi Security	Fixed
9	Medium	Timely Redistribution of Liquidated Collateral and Debt among Troves	DeFi Security	Confirmed
10	Medium	Potential Centralization Issues	DeFi Security	Confirmed
11	Medium	The Last Trove with Bad Debt can Influence the TCR	DeFi Security	Fixed
12	Medium	Potential Revert in Batch Liquidation of Troves	DeFi Security	Fixed
13	Medium	Incorrect Rounding Direction in shareBurnt()	DeFi Security	Confirmed
14	Medium	Lack of Check in Function setMaxSystemDebt()	DeFi Security	Fixed
15	Low	Conflicts of Updating rewardEndTime During Initialization of LPStakingPool	DeFi Security	Confirmed
16	High	Inappropriate Parameter Settings in initLockSettings	DeFi Security	Fixed
17	Medium	Lack of Check in Function setMCR()	DeFi Security	Fixed
18	High	Incorrect Calculation of Debt Interest	DeFi Security	Fixed
19	-	Incorrect Function Name	Recommendation	Fixed
20	-	Inconsistency between Implementation and Comments	Recommendation	Fixed
21	-	Contract Supports Multiple Collateral Assets and Relies on Timely Updates from the Price Oracle	Note	Confirmed

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Incorrect Calculation of Staking Rewards in esWenstaking

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `esWenStaking`, the function `_updateSnapshot()` is used to update rewards. `F_Tokens` represent the current value per share. When the user stakes for the first time, since the current stake amount is zero, the current `F_Token` will not be assigned to `snapshots`. The calculation of rewards earned by the user is based on the difference between the global `F_Tokens` and the `F_Tokens` recorded in the user's `snapshots`. In this case, the calculated reward is wrong and the user can invoke the function `unstake()` to claim the incorrect reward immediately.

```
54    function stakeWithoutLock(uint256 amount) external {
55        uint256 id = _MaxId;
56        _updateSnapshot(msg.sender, _MaxId);
57        esWen.sendToken(msg.sender, amount);
58        stakes[msg.sender][id] += amount;
59        stakeSharesByTokenAmountWithoutLock(msg.sender, id, amount);
60        totalStakes += amount;
61        emit StakedWithoutLock(msg.sender, id, amount);
62    }
63
64
65    function earned(address user, address token, uint256 id) public view override returns (uint256
          ) {
66        return (shareOf(user, id) * (F_Tokens[token] - snapshots[user][token][id])) / PRECISION;
67    }
68
69
70    function _updateSnapshot(address user, uint256 id) internal {
71        for (uint256 i = 0; i < tokensLength(); i++) {
72            uint256 currentStakes = stakes[user][id];
73            (address token,) = tokenAt(i);
74            if (currentStakes > 0) {
75                uint256 amount = earned(user, token, id);
76                snapshots[user][token][id] = F_Tokens[token];
77                if (amount > 0) {
78                    IERC20(token).transfer(user, amount);
79                    emit Claimed(user, id, amount);
80                }
81            }
82        }
83    }
84
85
86    function unstake(uint256 id) external {
87        if (id != _MaxId) {
```

```
88            require(block.timestamp > unlockTime(msg.sender, id), "esWenStaking: token is in lock
                  period");
89        }
90        uint256 amount = stakeOf(msg.sender, id);
91        _updateSnapshot(msg.sender, id);
92        burnSharesByTokenAmount(msg.sender, id, amount);
93        stakes[msg.sender][id] -= amount;
94        totalStakes -= amount;
95        esWen.transfer(msg.sender, amount);
96        emit Withdrawn(msg.sender, id, amount);
97    }
```

Listing 2.1: esWenStaking.sol

Impact The rewards pool can be drained.

Suggestion Modify the corresponding logic in the function `_updateSnapshot()` to ensure that users' `snapshots` will be correctly updated during their first stake.

2.1.2 Front-Running of Reward Distribution in submit()

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description In the contract of `esWenStaking`, function `submit()` is used to distribute rewards for staking. These rewards are not directly distributed to users. Instead, they are first recorded in the corresponding `F_Tokens[token]`. In this case, malicious users only need to stake before the function `submit()` is invoked to instantly receive a certain proportion of the newly allocated rewards.

```
127   function submit(address token, uint256 amount) external override {
128       require(amount > 0, "esWenStaking: zero amount");
129       require(totalShares() > 0, "esWenStaking: zero stakes");
130       require(tokenExists(token), "esWenStaking: nonexistent token");
131       IERC20(token).transferFrom(msg.sender, address(this), amount);
132       F_Tokens[token] += (amount * PRECISION) / totalShares();
133       emit FeeIncreased(token, amount);
134   }
```

Listing 2.2: esWenStaking.sol

Impact Front running can be used to claim undeserved newly allocated rewards by executing transactions ahead of function `submit()`, which results in losses of other stakers.

Suggestion Staking without a lock-up period should not immediately gain rewards.

2.1.3 Improper Check of Input in setRewardEndTime()

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `LPStakingPool`, the privileged function `setRewardEndTime()` is used to set the end time of reward distribution. However, there is no verification for the input parameter `_rewardEndTime` to ensure the new rewardEndTime is larger than the current `timestamp`. Instead, the previous `rewardEndTime` is checked, which means the `rewardEndTime` can only be updated before it expires.

```
48    function setRewardEndTime(uint256 _rewardEndTime) external onlyOwner {
49        updatePool();
50        require(rewardEndTime > block.timestamp, "invalid rewardEndTime");
51        rewardEndTime = _rewardEndTime;
52        emit RewardEndTimeUpdated(_rewardEndTime);
53    }
```

Listing 2.3: LPStakingPool.sol

Impact The newly updated `rewardEndTime` is able to be lower than the current timestamp, which terminates the distribution of rewards instantly.

Suggestion Verify the input parameter `_rewardEndTime` in function `setRewardEndTime()`.

2.1.4 Precision Loss of Rewards in claim()

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract of `Vest`, function `claim()` allows the user to claim the linearly released rewards. Based on the staking duration, there are three states of reward distribution: `inCliff`, `inRelease`, and `outOfRelease`. In the `inCliff` state, rewards are not distributed. In the `inRelease` state, the distribution of rewards is calculated based on the elapsed time. In the `outOfRelease` state, all rewards are fully released.

However, in the `inRelease` state, if the staked amount and elapsed time are both relatively small, the reward calculation may result in a loss of precision, causing the rewards to be rounded down to zero. Nevertheless, the contract still considers this reward segment as released and records the elapsed time.

The same issue also exists in function `claimAll()`.

```
62    function claimAll(uint256[] memory ids) external {
63        address account = msg.sender;
64        for (uint256 i = 0; i < ids.length; i++) {
65            uint256 id = ids[i];
66            require(id < currentIds[account], "Vest: invalid id");
67            State state = stateOf(account, id);
68            uint64 total = orderInfos[account][id].releaseEnd - orderInfos[account][id].cliffEnd;
69            if (orderInfos[account][id].released == total) {
70                continue;
71            }
72            if (state == State.inCliff) {
73                continue;
74            } else if (state == State.inRelease) {
75                uint64 walked = uint64(block.timestamp - orderInfos[account][id].cliffEnd);
76                uint256 amount = ((walked - orderInfos[account][id].released) * orderInfos[account
                      ][id].amount) / total;
```

```
77              orderInfos[account][id].released = walked;
78              Wen.esWen2Wen(account, amount);
79              emit Claimed(account, id, amount);
80          } else {
81              uint64 leftWalk = total - orderInfos[account][id].released;
82              uint256 amount = (leftWalk * orderInfos[account][id].amount) / total;
83              orderInfos[account][id].released = total;
84              Wen.esWen2Wen(account, amount);
85              emit Claimed(account, id, amount);
86          }
87      }
88  }
89  function claim(uint256 id) external {
90      address account = msg.sender;
91      require(id < currentIds[account], "Vest: invalid id");
92      State state = stateOf(account, id);
93      uint64 total = orderInfos[account][id].releaseEnd - orderInfos[account][id].cliffEnd;
94      require(orderInfos[account][id].released < total, "Vest: order claimed");
95      if (state == State.inCliff) {
96          revert("Vest: in cliff");
97      } else if (state == State.inRelease) {
98          uint64 walked = uint64(block.timestamp - orderInfos[account][id].cliffEnd);
99          uint256 amount = ((walked - orderInfos[account][id].released) * orderInfos[account][id
                ].amount) / total;
100         orderInfos[account][id].released = walked;
101         Wen.esWen2Wen(account, amount);
102         emit Claimed(account, id, amount);
103     } else {
104         uint64 leftWalk = total - orderInfos[account][id].released;
105         uint256 amount = (leftWalk * orderInfos[account][id].amount) / total;
106         orderInfos[account][id].released = total;
107         Wen.esWen2Wen(account, amount);
108         emit Claimed(account, id, amount);
109     }
110 }
```

Listing 2.4: Vest.sol

Impact Claiming rewards may result in loss of rewards for users due to precision loss.

Suggestion Set a minimum vesting amount in `Vest`.

2.1.5 Transferable esWen Token

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description According to the design, `esWen token` is supposed to be staked and then linearly converted into `Wen Token`, and `esWen Token` should not be transferable. However, in the current implementation, the function `transferFrom()` can bypass the check of senders.

```
87  function transfer(address to, uint256 amount) public override(IERC20, ERC20) returns (bool) {
```

```
88        _requireCallerIsSender();
89        _transfer(msg.sender, to, amount);
90        return true;
91    }
92    function transferFrom(address from, address to, uint256 value) public virtual returns (bool) {
93        address spender = _msgSender();
94        _spendAllowance(from, spender, value);
95        _transfer(from, to, value);
96        return true;
97    }
```

Listing 2.5: esWen.sol

Impact From the protocol design, `esWen` should not be allowed to be transferred to anyone, but the current implementation still allows transfers via the function `transferFrom()`.

Suggestion Overwrite the internal function `_transfer()`, instead of the function `transfer()`.

2.1.6 Incapable Collateral Token within Protocol

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description According to the design, the protocol supports multiple tokens as collateral, but the protocol's implementation is not compatible with rebasing tokens such as `stETH`.

Specifically, the function `transferFrom()` in `stETH` takes the parameter `_amount` as the quantity of `ETH` by default. It converts this amount to the corresponding number of shares through the function `getSharesByPooledEth()` and internally records changes in shares between accounts. The contract uses the parameter `_amount`, which is not converted into shares, as the basis for collecting and returning `stETH` during the process of opening and closing `troves`. As a result, due to the nature of `stETH`, the amount of `ETH` that can be obtained per share is increasing over time, users will receive fewer shares when closing `trove` compared to the initial deposit shares.

```
237    function transferFrom(address _sender, address _recipient, uint256 _amount) external returns (
           bool) {
238        _spendAllowance(_sender, msg.sender, _amount);
239        _transfer(_sender, _recipient, _amount);
240        return true;
241    }
```

Listing 2.6: stETH.sol

```
375    function _updateBalances() private {
376        _updateRewardIntegral(totalActiveDebt);
377        _accrueActiveInterests();
378    }
379
380
381    function _transfer(address _sender, address _recipient, uint256 _amount) internal {
382        uint256 _sharesToTransfer = getSharesByPooledEth(_amount);
```

```
383         _transferShares(_sender, _recipient, _sharesToTransfer);
384         _emitTransferEvents(_sender, _recipient, _amount, _sharesToTransfer);
385     }
```

Listing 2.7: stETH.sol

Impact Users will receive less shares as expected when they close `troves` that use tokens like `stETH` as collateral.

Suggestion Implement relevant logic to support special collateral tokens such as `stETH`.

Feedback The project party ensures that `Rebase-Token` will not be used as collateral in subsequent operations.

2.1.7 Losses of Stakers in Stability Pool due to Flash Loan Liquidation

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `StabilityPool`, the functions `provideToSP()` and `withdrawFromSP()` are allowed to be executed in the same transaction. This means that a large amount of `WenUSD` can be borrowed using a `flashloan` to first stake (`provideToSP()`) in the `StabilityPool`, then to liquidate troves to gain collateral rewards distributed to the `StabilityPool`, and finally unstake (`withdrawFromSP()`) to repay the `flashloan`.

```
196     function provideToSP(uint256 _amount) external {
197         require(!WenCore.paused(), "Deposits are paused");
198         require(_amount > 0, "StabilityPool: Amount must be non-zero");
199 
200 
201         _triggerRewardIssuance();
202 
203 
204         _accrueDepositorCollateralGain(msg.sender);
205 
206 
207         uint256 compoundedDeposit = getCompoundedDeposit(msg.sender);
208 
209 
210         _accrueRewards(msg.sender);
211 
212 
213         WenUSD.sendToSP(msg.sender, _amount);
214         uint256 newTotalWenUSDDeposits = totalWenUSDDeposits + _amount;
215         totalWenUSDDeposits = newTotalWenUSDDeposits;
216         emit StabilityPoolWenUSDBalanceUpdated(newTotalWenUSDDeposits);
217 
218 
219         uint256 newDeposit = compoundedDeposit + _amount;
220         accountDeposits[msg.sender] = newDeposit;
221 
```

```
222
223         _updateSnapshots(msg.sender, newDeposit);
224         emit UserDepositChanged(msg.sender, newDeposit);
225     }
```

Listing 2.8: StabilityPool.sol

```
230     function withdrawFromSP(uint256 _amount) external {
231         uint256 initialDeposit = accountDeposits[msg.sender];
232         require(initialDeposit > 0, "StabilityPool: User must have a non-zero deposit");
233
234
235         _triggerRewardIssuance();
236
237
238         _accrueDepositorCollateralGain(msg.sender);
239
240
241         uint256 compoundedDeposit = getCompoundedDeposit(msg.sender);
242         uint256 debtToWithdraw = WenMath._min(_amount, compoundedDeposit);
243
244
245         _accrueRewards(msg.sender);
246
247
248         if (debtToWithdraw > 0) {
249             WenUSD.returnFromPool(address(this), msg.sender, debtToWithdraw);
250             _decreaseDebt(debtToWithdraw);
251         }
252
253
254         // Update deposit
255         uint256 newDeposit = compoundedDeposit - debtToWithdraw;
256         accountDeposits[msg.sender] = newDeposit;
257
258
259         _updateSnapshots(msg.sender, newDeposit);
260         emit UserDepositChanged(msg.sender, newDeposit);
261     }
```

Listing 2.9: StabilityPool.sol

Impact This could result in losses for users who provide liquidity in the contract `StabilityPool` over a long time, and decrease their motivation to stake into the contract `StabilityPool`.

Suggestion Add a check in the function `withdrawFromSP()` to ensure that `provideToSP()` and `withdrawFromSP()` cannot be executed in the same block.

2.1.8 Incorrect Update of System Variable lastCollateralError_Offset

Severity High

Status Fixed in `Version 2`

Introduced by Version 1

Description As mentioned, the protocol supports multiple tokens as collateral, and in the Stability Pool, there are various collateral rewards obtained from liquidations as well. These rewards are distributed based on the amount of WenUSD deposited by the staker. In the reward distribution calculation process, the division before multiplication is intentionally used, resulting in precision loss. This loss of precision will be calculated and recorded in the global variable lastCollateralError_Offset, which will be redistributed in the next calculation.

The problem arises when this variable is shared among different collateral tokens during reward calculation.

```
359    function _computeRewardsPerUnitStaked(uint256 _collToAdd, uint256 _debtToOffset, uint256
           _totalWenUSDDeposits) internal returns (uint256 collateralGainPerUnitStaked, uint256
           debtLossPerUnitStaked) {
360        /*
361         * Compute the Debt and collateral rewards. Uses a "feedback" error correction, to keep
362         * the cumulative error in the P and S state variables low:
363         *
364         * 1) Form numerators which compensate for the floor division errors that occurred the last
                time this
365         * function was called.
366         * 2) Calculate "per-unit-staked" ratios.
367         * 3) Multiply each ratio back by its denominator, to reveal the current floor division
               error.
368         * 4) Store these errors for use in the next correction when this function is called.
369         * 5) Note: static analysis tools complain about this "division before multiplication",
               however, it is intended.
370         */
371        uint256 collateralNumerator = (_collToAdd * DECIMAL_PRECISION) + lastCollateralError_Offset
               ;
372        if (_debtToOffset == _totalWenUSDDeposits) {
373            debtLossPerUnitStaked = DECIMAL_PRECISION; // When the Pool depletes to 0, so does each
                    deposit
374            lastDebtLossError_Offset = 0;
375        } else {
376            uint256 debtLossNumerator = (_debtToOffset * DECIMAL_PRECISION) -
                   lastDebtLossError_Offset;
377            /*
378             * Add 1 to make error in quotient positive. We want "slightly too much" Debt loss,
379             * which ensures the error in any given CompoundedDeposit favors the Stability Pool.
380             */
381            debtLossPerUnitStaked = (debtLossNumerator / _totalWenUSDDeposits) + 1;
382            lastDebtLossError_Offset = (debtLossPerUnitStaked * _totalWenUSDDeposits) -
                   debtLossNumerator;
383        }
384        collateralGainPerUnitStaked = collateralNumerator / _totalWenUSDDeposits;
385        lastCollateralError_Offset = collateralNumerator - (collateralGainPerUnitStaked *
               _totalWenUSDDeposits);
386        return (collateralGainPerUnitStaked, debtLossPerUnitStaked);
387    }
```

Listing 2.10: StabilityPool.sol

Impact The rewards distributed to stakers will be allocated incorrectly.

Suggestion Implement a proper differentiation and accurate reward allocation mechanism for each collateral token.

2.1.9 Timely Redistribution of Liquidated Collateral and Debt among Troves

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description In the implementation of function `liquidateTroves()`, multiple `troves` can be liquidated in a single invocation. If the collateral ratio of a liquidated `trove` is less than `minICR`, its collateral and debt will be redistributed to other `troves`.

However, when liquidating troves that meet the criteria (collateralization ratio below `maxICR`), the bad debt from the liquidation is not immediately allocated to those `troves`. In fact, this will be done at the end of the liquidation process with the function `finalizeLiquidation()`.

Due to the redistribution mechanism, the liquidation of bad debt will inevitably decrease the collateralization ratio of `troves` with initially higher ratios. In this case, `troves` that were originally above `maxICR` can potentially have their collateralization ratios fall below `maxICR`. In this case, liquidators may miss some `troves` whose collateralization ratio should fall below `maxICR` during the liquidation process.

```
134    function liquidateTroves(ITroveManager troveManager, uint256 maxTrovesToLiquidate, uint256
           maxICR) external {
135        require(_enabledTroveManagers[troveManager], "TroveManager not approved");
136        IStabilityPool stabilityPoolCached = stabilityPool;
137        troveManager.distributeInterestDebt();
138
139
140        ISortedTroves sortedTrovesCached = ISortedTroves(troveManager.sortedTroves());
141
142
143        LiquidationValues memory singleLiquidation;
144        LiquidationTotals memory totals;
145        TroveManagerValues memory troveManagerValues;
146
147
148        uint256 trovesRemaining = maxTrovesToLiquidate;
149        uint256 troveCount = troveManager.getTroveOwnersCount();
150        troveManagerValues.price = troveManager.fetchPrice();
151        troveManagerValues.sunsetting = troveManager.sunsetting();
152        troveManagerValues.MCR = troveManager.MCR();
153        uint debtInStabPool = stabilityPoolCached.getTotalWenUSDDeposits();
154
155
156        while (trovesRemaining > 0 && troveCount > 1) {
157            address account = sortedTrovesCached.getLast();
158            uint ICR = troveManager.getCurrentICR(account, troveManagerValues.price);
159            if (ICR > maxICR) {
160                // set to 0 to ensure the next if block evaluates false
161                trovesRemaining = 0;
```

```
162             break;
163         }
164         if (ICR <= _100pct) {
165             singleLiquidation = _liquidateWithoutSP(troveManager, account);
166             _applyLiquidationValuesToTotals(totals, singleLiquidation);
167         } else if (ICR < troveManagerValues.MCR) {
168             singleLiquidation = _liquidateNormalMode(troveManager, account, debtInStabPool,
                    troveManagerValues.sunsetting);
169             debtInStabPool -= singleLiquidation.debtToOffset;
170             _applyLiquidationValuesToTotals(totals, singleLiquidation);
171         } else break; // break if the loop reaches a Trove with ICR >= MCR
172         unchecked {
173             --trovesRemaining;
174             --troveCount;
175         }
176     }
177     if (trovesRemaining > 0 && troveCount > 1) {
178         (uint entireSystemColl, uint entireSystemDebt) = borrowerOperations.
                getGlobalSystemBalances();
179         entireSystemColl -= totals.totalCollToSendToSP * troveManagerValues.price;
180         entireSystemDebt -= totals.totalDebtToOffset;
181         address nextAccount = sortedTrovesCached.getLast();
182         ITroveManager _troveManager = troveManager; //stack too deep workaround
183         while (trovesRemaining > 0 && troveCount > 1) {
184             uint ICR = troveManager.getCurrentICR(nextAccount, troveManagerValues.price);
185             if (ICR > maxICR) break;
186             unchecked {
187                 --trovesRemaining;
188             }
189             address account = nextAccount;
190             nextAccount = sortedTrovesCached.getPrev(account);
191 
192 
193             uint256 TCR = WenMath._computeCR(entireSystemColl, entireSystemDebt);
194             if (TCR >= CCR || ICR >= TCR) break;
195 
196 
197             singleLiquidation = _tryLiquidateWithCap(_troveManager, account, debtInStabPool,
                    troveManagerValues.MCR, troveManagerValues.price);
198             if (singleLiquidation.debtToOffset == 0) continue;
199             debtInStabPool -= singleLiquidation.debtToOffset;
200             entireSystemColl -= (singleLiquidation.collToSendToSP + singleLiquidation.
                    collSurplus) * troveManagerValues.price;
201             entireSystemDebt -= singleLiquidation.debtToOffset;
202             _applyLiquidationValuesToTotals(totals, singleLiquidation);
203             unchecked {
204                 --troveCount;
205             }
206         }
207     }
208 
209 
210     require(totals.totalDebtInSequence > 0, "nothing to liquidate");
```

```
211         if (totals.totalDebtToOffset > 0 || totals.totalCollToSendToSP > 0) {
212             // Move liquidated collateral and Debt to the appropriate pools
213             stabilityPoolCached.offset(troveManager.collateralToken(), totals.totalDebtToOffset,
                    totals.totalCollToSendToSP);
214             troveManager.decreaseDebtAndSendCollateral(address(stabilityPoolCached), totals.
                    totalDebtToOffset, totals.totalCollToSendToSP);
215         }
216         troveManager.finalizeLiquidation(msg.sender, totals.totalDebtToRedistribute, totals.
                totalCollToRedistribute, totals.totalCollSurplus, totals.totalDebtGasCompensation,
                totals.totalCollGasCompensation, totals.totalInterest);
217         emit Liquidation(totals.totalDebtInSequence, totals.totalCollInSequence - totals.
                totalCollGasCompensation - totals.totalCollSurplus, totals.totalCollGasCompensation,
                totals.totalDebtGasCompensation, totals.totalInterest);
218     }
```

Listing 2.11: LiquidateManager.sol

```
223     function batchLiquidateTroves(ITroveManager troveManager, address[] memory _troveArray) public
            {
224         require(_enabledTroveManagers[troveManager], "TroveManager not approved");
225         require(_troveArray.length != 0, "TroveManager: Calldata address array must not be empty");
226         troveManager.distributeInterestDebt();
227
228
229         LiquidationValues memory singleLiquidation;
230         LiquidationTotals memory totals;
231         TroveManagerValues memory troveManagerValues;
232
233
234         IStabilityPool stabilityPoolCached = stabilityPool;
235         uint debtInStabPool = stabilityPoolCached.getTotalWenUSDDeposits();
236         troveManagerValues.price = troveManager.fetchPrice();
237         troveManagerValues.sunsetting = troveManager.sunsetting();
238         troveManagerValues.MCR = troveManager.MCR();
239         uint troveCount = troveManager.getTroveOwnersCount();
240         uint length = _troveArray.length;
241         uint troveIter;
242         while (troveIter < length && troveCount > 1) {
243             // first iteration round, when all liquidated troves have ICR < MCR we do not need to
                    track TCR
244             address account = _troveArray[troveIter];
245
246
247             // closed / non-existent troves return an ICR of type(uint).max and are ignored
248             uint ICR = troveManager.getCurrentICR(account, troveManagerValues.price);
249             if (ICR <= _100pct) {
250                 singleLiquidation = _liquidateWithoutSP(troveManager, account);
251             } else if (ICR < troveManagerValues.MCR) {
252                 singleLiquidation = _liquidateNormalMode(troveManager, account, debtInStabPool,
                        troveManagerValues.sunsetting);
253                 debtInStabPool -= singleLiquidation.debtToOffset;
254             } else {
255                 // As soon as we find a trove with ICR >= MCR we need to start tracking the global
```

```
                    TCR with the next loop
256                 break;
257             }
258             _applyLiquidationValuesToTotals(totals, singleLiquidation);
259             unchecked {
260                 ++troveIter;
261                 --troveCount;
262             }
263         }
264
265
266         if (troveIter < length && troveCount > 1) {
267             // second iteration round, if we receive a trove with ICR > MCR and need to track TCR
268             (uint256 entireSystemColl, uint256 entireSystemDebt) = borrowerOperations.
                    getGlobalSystemBalances();
269             entireSystemColl -= totals.totalCollToSendToSP * troveManagerValues.price;
270             entireSystemDebt -= totals.totalDebtToOffset;
271             while (troveIter < length && troveCount > 1) {
272                 address account = _troveArray[troveIter];
273                 uint ICR = troveManager.getCurrentICR(account, troveManagerValues.price);
274                 unchecked {
275                     ++troveIter;
276                 }
277                 if (ICR <= _100pct) {
278                     singleLiquidation = _liquidateWithoutSP(troveManager, account);
279                 } else if (ICR < troveManagerValues.MCR) {
280                     singleLiquidation = _liquidateNormalMode(troveManager, account, debtInStabPool,
                            troveManagerValues.sunsetting);
281                 } else {
282                     uint256 TCR = WenMath._computeCR(entireSystemColl, entireSystemDebt);
283                     if (TCR >= CCR || ICR >= TCR) continue;
284                     singleLiquidation = _tryLiquidateWithCap(troveManager, account, debtInStabPool,
                            troveManagerValues.MCR, troveManagerValues.price);
285                     if (singleLiquidation.debtToOffset == 0) continue;
286                 }
287
288
289                 debtInStabPool -= singleLiquidation.debtToOffset;
290                 entireSystemColl -= (singleLiquidation.collToSendToSP + singleLiquidation.
                        collSurplus) * troveManagerValues.price;
291                 entireSystemDebt -= singleLiquidation.debtToOffset;
292                 _applyLiquidationValuesToTotals(totals, singleLiquidation);
293                 unchecked {
294                     --troveCount;
295                 }
296             }
297         }
298
299
300         require(totals.totalDebtInSequence > 0, "TroveManager: nothing to liquidate");
301
302
303         if (totals.totalDebtToOffset > 0 || totals.totalCollToSendToSP > 0) {
```

```
304             // Move liquidated collateral and Debt to the appropriate pools
305             stabilityPoolCached.offset(troveManager.collateralToken(), totals.totalDebtToOffset,
                    totals.totalCollToSendToSP);
306             troveManager.decreaseDebtAndSendCollateral(address(stabilityPoolCached), totals.
                    totalDebtToOffset, totals.totalCollToSendToSP);
307         }
308         troveManager.finalizeLiquidation(msg.sender, totals.totalDebtToRedistribute, totals.
                totalCollToRedistribute, totals.totalCollSurplus, totals.totalDebtGasCompensation,
                totals.totalCollGasCompensation, totals.totalInterest);
309
310
311         emit Liquidation(totals.totalDebtInSequence, totals.totalCollInSequence - totals.
                totalCollGasCompensation - totals.totalCollSurplus, totals.totalCollGasCompensation,
                totals.totalDebtGasCompensation, totals.totalInterest);
312     }
```

Listing 2.12: LiquidateManager.sol

Impact Liquidators may receive less profit than expected while liquidating multiple `troves` via the function `liquidateTroves()`.

Suggestion Timely updates of collateralization ratio of relevant `troves` during liquidation.

2.1.10 Potential Centralization Issues

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description The protocol has potential centralization problems, the owner has privilege to conduct sensitive operations like minting `WEN` or `esWEN` to any accounts.

If the owner's private key is lost or maliciously exploited, it could lead to losses for the entire protocol.

```
145     function mint(address account, uint256 amount) external onlyOwner {
146         require(totalSupply() + esWen.totalSupply() + amount <= MaxCap, "Wen: exceeds MaxCap");
147         _mint(account, amount);
148     }
149     function mintesWen(address account, uint256 amount) external onlyOwner {
150         require(totalSupply() + esWen.totalSupply() + amount <= MaxCap, "Wen: exceeds MaxCap");
151         esWen.mint(account, amount);
152     }
```

Listing 2.13: TroveManager.sol

Impact If the owner's private key is lost or maliciously exploited, it will cause significant losses to the protocol.

Suggestion Safeguard the owner's private key, and remove unnecessary interfaces (e.g., `mint`, `mintesWen`) in contract `Wen`.

Feedback `esWen`'s incentives will be distributed in batches, and the time and quantity of each distribution need to be determined based on the actual situation of future operations.Therefore, it is necessary to introduce this interface to control the minting of `esWen`.

2.1.11 The Last Trove with Bad Debt can Influence the TCR

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description During the sunsetting of the `troveManager`, the liquidation process cannot liquidate all the `troves` and the function `redeemCollateral()` does not support the redemption of collateral from `troves` with bad debt. In this case, if the `trove` with bad debt is the last active one, unless the owner of the trove actively closes it, the entire system debt cannot be zero. This can prohibit the sunsetting `troveManager` being removed, which further influences the `TCR`.

```
134    function liquidateTroves(
135        ITroveManager troveManager,
136        uint256 maxTrovesToLiquidate,
137        uint256 maxICR
138    ) external {
139        require(
140            _enabledTroveManagers[troveManager],
141            "TroveManager not approved"
142        );
143        IStabilityPool stabilityPoolCached = stabilityPool;
144        troveManager.updateBalances();
145        ISortedTroves sortedTrovesCached = ISortedTroves(
146            troveManager.sortedTroves()
147        );
148        LiquidationValues memory singleLiquidation;
149        LiquidationTotals memory totals;
150        TroveManagerValues memory troveManagerValues;
151        uint256 trovesRemaining = maxTrovesToLiquidate;
152        uint256 troveCount = troveManager.getTroveOwnersCount();
153        troveManagerValues.price = troveManager.fetchPrice();
154        troveManagerValues.sunsetting = troveManager.sunsetting();
155        troveManagerValues.MCR = troveManager.MCR();
156        uint debtInStabPool = stabilityPoolCached.getTotalDebtTokenDeposits();
157        while (trovesRemaining > 0 && troveCount > 1) {
158            address account = sortedTrovesCached.getLast();
159            uint ICR = troveManager.getCurrentICR(
160                account,
161                troveManagerValues.price
162            );
163            if (ICR > maxICR) {
164                // set to 0 to ensure the next if block evaluates false
165                trovesRemaining = 0;
166                break;
167            }
168            if (ICR <= _100pct) {
169                singleLiquidation = _liquidateWithoutSP(troveManager, account);
170                _applyLiquidationValuesToTotals(totals, singleLiquidation);
171            } else if (ICR < troveManagerValues.MCR) {
172                singleLiquidation = _liquidateNormalMode(
173                    troveManager,
```

```
174                 account,
175                 debtInStabPool,
176                 troveManagerValues.sunsetting
177             );
178             debtInStabPool -= singleLiquidation.debtToOffset;
179             _applyLiquidationValuesToTotals(totals, singleLiquidation);
180         } else break; // break if the loop reaches a Trove with ICR >= MCR
181         unchecked {
182             --trovesRemaining;
183             --troveCount;
184         }
185     }
186     if (
187         trovesRemaining > 0 &&
188         !troveManagerValues.sunsetting &&
189         troveCount > 1
190     ) {
191         (uint entireSystemColl, uint entireSystemDebt) = borrowerOperations
192             .getGlobalSystemBalances();
193         entireSystemColl -=
194             totals.totalCollToSendToSP *
195             troveManagerValues.price;
196         entireSystemDebt -= totals.totalDebtToOffset;
197         address nextAccount = sortedTrovesCached.getLast();
198         ITroveManager _troveManager = troveManager; //stack too deep workaround
199         while (trovesRemaining > 0 && troveCount > 1) {
200             uint ICR = troveManager.getCurrentICR(
201                 nextAccount,
202                 troveManagerValues.price
203             );
204             if (ICR > maxICR) break;
205             unchecked {
206                 --trovesRemaining;
207             }
208             address account = nextAccount;
209             nextAccount = sortedTrovesCached.getPrev(account);
210             uint256 TCR = ListaMath._computeCR(
211                 entireSystemColl,
212                 entireSystemDebt
213             );
214             if (TCR >= CCR || ICR >= TCR) break;
215             singleLiquidation = _tryLiquidateWithCap(
216                 _troveManager,
217                 account,
218                 debtInStabPool,
219                 troveManagerValues.MCR,
220                 troveManagerValues.price
221             );
222             if (singleLiquidation.debtToOffset == 0) continue;
223             debtInStabPool -= singleLiquidation.debtToOffset;
224             entireSystemColl -=
225                 (singleLiquidation.collToSendToSP +
226                     singleLiquidation.collSurplus) *
```

```
227                 troveManagerValues.price;
228             entireSystemDebt -= singleLiquidation.debtToOffset;
229             _applyLiquidationValuesToTotals(totals, singleLiquidation);
230             unchecked {
231                 --troveCount;
232             }
233         }
234     }
235     require(
236         totals.totalDebtInSequence > 0,
237         "TroveManager: nothing to liquidate"
238     );
239     if (totals.totalDebtToOffset > 0 || totals.totalCollToSendToSP > 0) {
240         // Move liquidated collateral and Debt to the appropriate pools
241         stabilityPoolCached.offset(
242             troveManager.collateralToken(),
243             totals.totalDebtToOffset,
244             totals.totalCollToSendToSP
245         );
246         troveManager.decreaseDebtAndSendCollateral(
247             address(stabilityPoolCached),
248             totals.totalDebtToOffset,
249             totals.totalCollToSendToSP
250         );
251     }
252     troveManager.finalizeLiquidation(
253         msg.sender,
254         totals.totalDebtToRedistribute,
255         totals.totalCollToRedistribute,
256         totals.totalCollSurplus,
257         totals.totalDebtGasCompensation,
258         totals.totalCollGasCompensation
259     );
260     emit Liquidation(
261         totals.totalDebtInSequence,
262         totals.totalCollInSequence -
263             totals.totalCollGasCompensation -
264             totals.totalCollSurplus,
265         totals.totalCollGasCompensation,
266         totals.totalDebtGasCompensation
267     );
268 }
```

Listing 2.14: LiquidateManager.sol

```
476   function redeemCollateral(uint256 _debtAmount, address _firstRedemptionHint, address
          _upperPartialRedemptionHint, address _lowerPartialRedemptionHint, uint256
          _partialRedemptionHintNICR, uint256 _maxIterations, uint256 _maxFeePercentage) external {
477       ISortedTroves _sortedTrovesCached = sortedTroves;
478       RedemptionTotals memory totals;
479       require(_maxFeePercentage >= redemptionFeeFloor && _maxFeePercentage <= maxRedemptionFee, "
              Max fee 0.5% to 100%");
480       require(block.timestamp >= systemDeploymentTime + BOOTSTRAP_PERIOD, "BOOTSTRAP_PERIOD");
```

```
481         _distributeInterestDebt();
482         totals.price = fetchPrice();
483         uint256 _MCR = MCR;
484         require(IBorrowerOperations(borrowerOperationsAddress).getTCR() >= _MCR, "Cannot redeem
                when TCR < MCR");
485         require(_debtAmount > 0, "Amount must be greater than zero");
486         require(WenUSD.balanceOf(msg.sender) >= _debtAmount, "Insufficient balance");
487         totals.totalDebtSupplyAtStart = getGlobalSystemDebt();
488         totals.remainingDebt = _debtAmount;
489         address currentBorrower;
490         if (_isValidFirstRedemptionHint(_sortedTrovesCached, _firstRedemptionHint, totals.price,
                _MCR)) {
491             currentBorrower = _firstRedemptionHint;
492         } else {
493             currentBorrower = _sortedTrovesCached.getLast();
494             // Find the first trove with ICR >= MCR
495             while (currentBorrower != address(0) && getRedemptionICR(currentBorrower, totals.price)
                    < _MCR) {
496                 currentBorrower = _sortedTrovesCached.getPrev(currentBorrower);
497             }
498         }
499         // Loop through the Troves starting from the one with lowest collateral ratio until _amount
                of debt is exchanged for collateral
500         if (_maxIterations == 0) {
501             _maxIterations = type(uint256).max;
502         }
503         while (currentBorrower != address(0) && totals.remainingDebt > 0 && _maxIterations > 0) {
504             _maxIterations--;
505             // Save the address of the Trove preceding the current one, before potentially
                    modifying the list
506             address nextUserToCheck = _sortedTrovesCached.getPrev(currentBorrower);
507             _applyPendingRewards(currentBorrower);
508             SingleRedemptionValues memory singleRedemption = _redeemCollateralFromTrove(
                    _sortedTrovesCached, currentBorrower, totals.remainingDebt, totals.price,
                    _upperPartialRedemptionHint, _lowerPartialRedemptionHint,
                    _partialRedemptionHintNICR);
509             if (singleRedemption.cancelledPartial) break; // Partial redemption was cancelled (out-
                    of-date hint, or new net debt < minimum), therefore we could not redeem from the
                    last Trove
510             totals.totalDebtToRedeem = totals.totalDebtToRedeem + singleRedemption.debtLot;
511             totals.totalCollateralDrawn = totals.totalCollateralDrawn + singleRedemption.
                    collateralLot;
512             totals.totalInterest = totals.totalInterest + singleRedemption.interestLot;
513             totals.remainingDebt = totals.remainingDebt - singleRedemption.debtLot;
514             currentBorrower = nextUserToCheck;
515         }
516         require(totals.totalCollateralDrawn > 0, "Unable to redeem any amount");
517         // Decay the baseRate due to time passed, and then increase it according to the size of
                this redemption.
518         // Use the saved total debt supply value, from before it was reduced by the redemption.
519         _updateBaseRateFromRedemption(totals.totalCollateralDrawn, totals.price, totals.
                totalDebtSupplyAtStart);
520         // Calculate the collateral fee
```

```
521         totals.collateralFee = sunsetting ? 0 : _calcRedemptionFee(getRedemptionRate(), totals.
                totalCollateralDrawn);
522         _requireUserAcceptsFee(totals.collateralFee, totals.totalCollateralDrawn, _maxFeePercentage
                );
523         _sendCollateral(feeReceiver(), totals.collateralFee);
524         totals.collateralToSendToRedeemer = totals.totalCollateralDrawn - totals.collateralFee;
525         emit Redemption(_debtAmount, totals.totalDebtToRedeem, totals.totalCollateralDrawn, totals.
                totalInterest, totals.collateralFee);
526         // Burn the total debt that is cancelled with debt, and send the redeemed collateral to msg
                .sender
527         WenUSD.burn(msg.sender, totals.totalDebtToRedeem);
528         // Update Trove Manager debt, and send collateral to account
529         totalActiveDebt = totalActiveDebt - totals.totalDebtToRedeem;
530         decreaseOutstandingInterestDebt(totals.totalInterest);
531         _sendCollateral(msg.sender, totals.collateralToSendToRedeemer);
532         _resetState();
533     }
```

Listing 2.15: TroveManager.sol

```
549     function _redeemCollateralFromTrove(
550         ISortedTroves _sortedTrovesCached,
551         address _borrower,
552         uint256 _maxDebtAmount,
553         uint256 _price,
554         address _upperPartialRedemptionHint,
555         address _lowerPartialRedemptionHint,
556         uint256 _partialRedemptionHintNICR
557     ) internal returns (SingleRedemptionValues memory singleRedemption) {
558         Trove storage t = Troves[_borrower];
559         uint256 interest = getTroveInterest(_borrower, t.debt);
560         if (_maxDebtAmount < interest) {
561             singleRedemption.cancelledPartial = true;
562             return singleRedemption;
563         }
564         singleRedemption.interestLot = interest;
565         // Determine the remaining amount (lot) to be redeemed, capped by the entire debt of the
                Trove minus the liquidation reserve
566         singleRedemption.debtLot = WenMath._min(_maxDebtAmount - singleRedemption.interestLot, t.
                debt - DEBT_GAS_COMPENSATION);
567         // Get the CollateralLot of equivalent value in USD
568         singleRedemption.collateralLot = ((singleRedemption.debtLot + singleRedemption.interestLot)
                 * DECIMAL_PRECISION) / _price;
569         // Decrease the debt and collateral of the current Trove according to the debt lot and
                corresponding collateral to send
570         uint256 newDebt = (t.debt) - singleRedemption.debtLot;
571         uint256 newColl = (t.coll) - singleRedemption.collateralLot;
572         if (newDebt == DEBT_GAS_COMPENSATION) {
573             // No debt left in the Trove (except for the liquidation reserve), therefore the trove
                    gets closed
574             _removeStake(_borrower);
575             _closeTrove(_borrower, Status.closedByRedemption);
576             _redeemCloseTrove(_borrower, DEBT_GAS_COMPENSATION, newColl);
```

```
577             emit TroveUpdated(_borrower, 0, 0, 0, TroveManagerOperation.redeemCollateral);
578         } else {
579             uint256 newNICR = WenMath._computeNominalCR(newColl, newDebt);
580             /*
581              * If the provided hint is out of date, we bail since trying to reinsert without a good
                     hint will almost
582              * certainly result in running out of gas.
583              *
584              * If the resultant net debt of the partial is less than the minimum, net debt we bail.
585              */
586             {
587                 // We check if the ICR hint is reasonable up to date, with continuous interest
                        there might be slight differences (<1bps)
588                 uint256 icrError = _partialRedemptionHintNICR > newNICR ?
                        _partialRedemptionHintNICR - newNICR : newNICR - _partialRedemptionHintNICR;
589                 if (icrError > 5e14 || _getNetDebt(newDebt) < IBorrowerOperations(
                        borrowerOperationsAddress).minNetDebt()) {
590                     singleRedemption.cancelledPartial = true;
591                     return singleRedemption;
592                 }
593             }
594             _sortedTrovesCached.reInsert(_borrower, newNICR, _upperPartialRedemptionHint,
                    _lowerPartialRedemptionHint);
595             t.debt = newDebt;
596             t.coll = newColl;
597             _updateStakeAndTotalStakes(t);
598             _updateTroveRewardSnapshots(_borrower);
599             emit TroveUpdated(_borrower, newDebt, newColl, t.stake, TroveManagerOperation.
                    redeemCollateral);
600         }
601         return singleRedemption;
602     }
```

Listing 2.16: TroveManager.sol

Impact `Troves` that cannot be closed and remain as bad debt will accumulate debt interests over time. This accumulation of debt interests can indeed have an impact on the system's `Total Collateralization Ratio` (`TCR`).

Suggestion Implement corresponding logic for the `admin` to handle the last bad debt `trove`.

2.1.12 Potential Revert in Batch Liquidation of Troves

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description During the liquidation process, if a trove's `ICR` is above `MCR`, the protocol will try to use the liquidity from the `StabilityPool` for the liquidation, and the collateral rewards obtained from the liquidation will be sent to the `StabilityPool`, rewarding the liquidity providers. If one collateral is sunsetting, the `indexByCollateral[collateral]` will be set to 0, which disables the liquidity in the `StabilityPool` from being used in the liquidation process.

In this case, when the protocol is in `Recovery Mode` and the specified collateral is sunsetting, if a user tries to liquidate a batch of `troves` , including a `trove` whose `ICR` is between `MCR` and `CCR`, it will revert. In this case, the `troves` whose `ICR` is lower than `MCR` can not be liquidated, which is not friendly for liquidators.

```
223    function batchLiquidateTroves(ITroveManager troveManager, address[] memory _troveArray) public
           {
224        ......
225        while (troveIter < length && troveCount > 1) {
226            // first iteration round, when all liquidated troves have ICR < MCR we do not need to
                   track TCR
227            address account = _troveArray[troveIter];
228
229
230            // closed / non-existent troves return an ICR of type(uint).max and are ignored
231            uint ICR = troveManager.getCurrentICR(account, troveManagerValues.price);
232            if (ICR <= _100pct) {
233                singleLiquidation = _liquidateWithoutSP(troveManager, account);
234            } else if (ICR < troveManagerValues.MCR) {
235                singleLiquidation = _liquidateNormalMode(troveManager, account, debtInStabPool,
                       troveManagerValues.sunsetting);
236                debtInStabPool -= singleLiquidation.debtToOffset;
237            } else {
238                // As soon as we find a trove with ICR >= MCR we need to start tracking the global
                       TCR with the next loop
239                break;
240            }
241            _applyLiquidationValuesToTotals(totals, singleLiquidation);
242            unchecked {
243                ++troveIter;
244                --troveCount;
245            }
246        }
247
248
249        if (troveIter < length && troveCount > 1) {
250            // second iteration round, if we receive a trove with ICR > MCR and need to track TCR
251            (uint256 entireSystemColl, uint256 entireSystemDebt) = borrowerOperations.
                   getGlobalSystemBalances();
252            entireSystemColl -= totals.totalCollToSendToSP * troveManagerValues.price;
253            entireSystemDebt -= totals.totalDebtToOffset;
254            while (troveIter < length && troveCount > 1) {
255                address account = _troveArray[troveIter];
256                uint ICR = troveManager.getCurrentICR(account, troveManagerValues.price);
257                unchecked {
258                    ++troveIter;
259                }
260                if (ICR <= _100pct) {
261                    singleLiquidation = _liquidateWithoutSP(troveManager, account);
262                } else if (ICR < troveManagerValues.MCR) {
263                    singleLiquidation = _liquidateNormalMode(troveManager, account, debtInStabPool,
                           troveManagerValues.sunsetting);
264                } else {
265                    uint256 TCR = WenMath._computeCR(entireSystemColl, entireSystemDebt);
```

```
266                 if (TCR >= CCR || ICR >= TCR) continue;
267                 singleLiquidation = _tryLiquidateWithCap(troveManager, account, debtInStabPool,
                        troveManagerValues.MCR, troveManagerValues.price);
268                 if (singleLiquidation.debtToOffset == 0) continue;
269             }
270
271
272             debtInStabPool -= singleLiquidation.debtToOffset;
273             entireSystemColl -= (singleLiquidation.collToSendToSP + singleLiquidation.
                    collSurplus) * troveManagerValues.price;
274             entireSystemDebt -= singleLiquidation.debtToOffset;
275             _applyLiquidationValuesToTotals(totals, singleLiquidation);
276             unchecked {
277                 --troveCount;
278             }
279         }
280     }
281
282
283     require(totals.totalDebtInSequence > 0, "TroveManager: nothing to liquidate");
284
285
286     if (totals.totalDebtToOffset > 0 || totals.totalCollToSendToSP > 0) {
287         // Move liquidated collateral and Debt to the appropriate pools
288         stabilityPoolCached.offset(troveManager.collateralToken(), totals.totalDebtToOffset,
                totals.totalCollToSendToSP);
289         troveManager.decreaseDebtAndSendCollateral(address(stabilityPoolCached), totals.
                totalDebtToOffset, totals.totalCollToSendToSP);
290     }
291     troveManager.finalizeLiquidation(msg.sender, totals.totalDebtToRedistribute, totals.
            totalCollToRedistribute, totals.totalCollSurplus, totals.totalDebtGasCompensation,
            totals.totalCollGasCompensation, totals.totalInterest);
292
293
294     emit Liquidation(totals.totalDebtInSequence, totals.totalCollInSequence - totals.
            totalCollGasCompensation - totals.totalCollSurplus, totals.totalCollGasCompensation,
            totals.totalDebtGasCompensation, totals.totalInterest);
295 }
296 function offset(IERC20 collateral, uint256 _debtToOffset, uint256 _collToAdd) external virtual
        {
297     _offset(collateral, _debtToOffset, _collToAdd);
298 }
299
300
301 function _offset(IERC20 collateral, uint256 _debtToOffset, uint256 _collToAdd) internal {
302     require(msg.sender == liquidationManager, "StabilityPool: Caller is not Liquidation Manager
            ");
303     uint256 idx = indexByCollateral[collateral];
304     idx -= 1;
305
306
307     uint256 totalDebt = totalWenUSDDeposits; // cached to save an SLOAD
308     if (totalDebt == 0 || _debtToOffset == 0) {
```

```
309             return;
310         }
311
312
313         _triggerRewardIssuance();
314
315
316         (uint256 collateralGainPerUnitStaked, uint256 debtLossPerUnitStaked) =
                _computeRewardsPerUnitStaked(_collToAdd, _debtToOffset, totalDebt);
317
318
319         _updateRewardSumAndProduct(collateralGainPerUnitStaked, debtLossPerUnitStaked, idx); //
                updates S and P
320
321
322         // Cancel the liquidated Debt debt with the Debt in the stability pool
323         _decreaseDebt(_debtToOffset);
324     }
```

Listing 2.17: LiquidationManager.sol

```
223     function offset(IERC20 collateral, uint256 _debtToOffset, uint256 _collToAdd) external virtual
            {
224         _offset(collateral, _debtToOffset, _collToAdd);
225     }
226
227
228     function _offset(IERC20 collateral, uint256 _debtToOffset, uint256 _collToAdd) internal {
229         require(msg.sender == liquidationManager, "StabilityPool: Caller is not Liquidation Manager
                ");
230         uint256 idx = indexByCollateral[collateral];
231         idx -= 1;
232
233
234         uint256 totalDebt = totalWenUSDDeposits; // cached to save an SLOAD
235         if (totalDebt == 0 || _debtToOffset == 0) {
236             return;
237         }
238
239
240         _triggerRewardIssuance();
241
242
243         (uint256 collateralGainPerUnitStaked, uint256 debtLossPerUnitStaked) =
                _computeRewardsPerUnitStaked(_collToAdd, _debtToOffset, totalDebt);
244
245
246         _updateRewardSumAndProduct(collateralGainPerUnitStaked, debtLossPerUnitStaked, idx); //
                updates S and P
247
248
249         // Cancel the liquidated Debt debt with the Debt in the stability pool
250         _decreaseDebt(_debtToOffset);
```

```
251     }
```

Listing 2.18: StabilityPool.sol

```
223     function startCollateralSunset(IERC20 collateral) external onlyOwner {
224         require(indexByCollateral[collateral] > 0, "Collateral already sunsetting");
225         _sunsetIndexes[queue.nextSunsetIndexKey++] = SunsetIndex(uint128(indexByCollateral[
                collateral] - 1), uint128(block.timestamp + SUNSET_DURATION));
226         delete indexByCollateral[collateral]; //This will prevent calls to the SP in case of
                liquidations
227     }
```

Listing 2.19: LiquidationManager.sol

```
134     function liquidateTroves(ITroveManager troveManager, uint256 maxTrovesToLiquidate, uint256
            maxICR) external {
135         require(_enabledTroveManagers[troveManager], "TroveManager not approved");
136         IStabilityPool stabilityPoolCached = stabilityPool;
137         troveManager.distributeInterestDebt();
138         ISortedTroves sortedTrovesCached = ISortedTroves(troveManager.sortedTroves());
139         LiquidationValues memory singleLiquidation;
140         LiquidationTotals memory totals;
141         TroveManagerValues memory troveManagerValues;
142         uint256 trovesRemaining = maxTrovesToLiquidate;
143         uint256 troveCount = troveManager.getTroveOwnersCount();
144         troveManagerValues.price = troveManager.fetchPrice();
145         troveManagerValues.sunsetting = troveManager.sunsetting();
146         troveManagerValues.MCR = troveManager.MCR();
147         uint debtInStabPool = stabilityPoolCached.getTotalWenUSDDeposits();
148         while (trovesRemaining > 0 && troveCount > 1) {
149             address account = sortedTrovesCached.getLast();
150             uint ICR = troveManager.getCurrentICR(account, troveManagerValues.price);
151             if (ICR > maxICR) {
152                 // set to 0 to ensure the next if block evaluates false
153                 trovesRemaining = 0;
154                 break;
155             }
156             if (ICR <= _100pct) {
157                 singleLiquidation = _liquidateWithoutSP(troveManager, account);
158                 _applyLiquidationValuesToTotals(totals, singleLiquidation);
159             } else if (ICR < troveManagerValues.MCR) {
160                 singleLiquidation = _liquidateNormalMode(troveManager, account, debtInStabPool,
                        troveManagerValues.sunsetting);
161                 debtInStabPool -= singleLiquidation.debtToOffset;
162                 _applyLiquidationValuesToTotals(totals, singleLiquidation);
163             } else break; // break if the loop reaches a Trove with ICR >= MCR
164             unchecked {
165                 --trovesRemaining;
166                 --troveCount;
167             }
168         }
169         if (trovesRemaining > 0 && troveCount > 1) {
170             (uint entireSystemColl, uint entireSystemDebt) = borrowerOperations.
                    getGlobalSystemBalances();
```

```
171             entireSystemColl -= totals.totalCollToSendToSP * troveManagerValues.price;
172             entireSystemDebt -= totals.totalDebtToOffset;
173             address nextAccount = sortedTrovesCached.getLast();
174             ITroveManager _troveManager = troveManager; //stack too deep workaround
175             while (trovesRemaining > 0 && troveCount > 1) {
176                 uint ICR = troveManager.getCurrentICR(nextAccount, troveManagerValues.price);
177                 if (ICR > maxICR) break;
178                 unchecked {
179                     --trovesRemaining;
180                 }
181                 address account = nextAccount;
182                 nextAccount = sortedTrovesCached.getPrev(account);
183                 uint256 TCR = WenMath._computeCR(entireSystemColl, entireSystemDebt);
184                 if (TCR >= CCR || ICR >= TCR) break;
185                 singleLiquidation = _tryLiquidateWithCap(_troveManager, account, debtInStabPool,
                        troveManagerValues.MCR, troveManagerValues.price);
186                 if (singleLiquidation.debtToOffset == 0) continue;
187                 debtInStabPool -= singleLiquidation.debtToOffset;
188                 entireSystemColl -= (singleLiquidation.collToSendToSP + singleLiquidation.
                        collSurplus) * troveManagerValues.price;
189                 entireSystemDebt -= singleLiquidation.debtToOffset;
190                 _applyLiquidationValuesToTotals(totals, singleLiquidation);
191                 unchecked {
192                     --troveCount;
193                 }
194             }
195         }
196         require(totals.totalDebtInSequence > 0, "nothing to liquidate");
197         if (totals.totalDebtToOffset > 0 || totals.totalCollToSendToSP > 0) {
198             // Move liquidated collateral and Debt to the appropriate pools
199             stabilityPoolCached.offset(troveManager.collateralToken(), totals.totalDebtToOffset,
                    totals.totalCollToSendToSP);
200             troveManager.decreaseDebtAndSendCollateral(address(stabilityPoolCached), totals.
                    totalDebtToOffset, totals.totalCollToSendToSP);
201         }
202         troveManager.finalizeLiquidation(msg.sender, totals.totalDebtToRedistribute, totals.
                totalCollToRedistribute, totals.totalCollSurplus, totals.totalDebtGasCompensation,
                totals.totalCollGasCompensation, totals.totalInterest);
203         emit Liquidation(totals.totalDebtInSequence, totals.totalCollInSequence - totals.
                totalCollGasCompensation - totals.totalCollSurplus, totals.totalCollGasCompensation,
                totals.totalDebtGasCompensation, totals.totalInterest);
204     }
```

Listing 2.20: LiquidationManager.sol

Impact When liquidating a batch of troves, if it contains a trove whose `ICR` is higher than `MCR` but less than `CCR`, the whole liquidation process will revert.

Suggestion When the protocol is in `Recovery Mode` and the specified collateral is sunsetting, skip the liquidation for troves whose `ICR` is between `MCR` and `CCR`.

2.1.13 Incorrect Rounding Direction in shareBurnt()

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description In the `StakeBoost` contract, the function `shareBurnt()` calculates the number of shares to be burned based on the input parameter `amount`. The default downward rounding in division calculations will cause precision loss, which in fact burns less shares of the user.

```
72    function shareBurnt(address user, uint256 id, uint256 amount, uint256 stakes) public view
          returns (uint256) {
73        if (stakes == 0)
74        {
75            return 0;
76        }
77            return (amount * shareOf(user, id)) / stakes;
78    }
```

Listing 2.21: StakingBoost.sol

Impact The precision loss caused by rounding down could result in users burning less shares, resulting in losses for the protocol.

Suggestion Use upward rounding in the function `shareBurnt()`.

2.1.14 Lack of Check in Function setMaxSystemDebt()

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `TroveManager` contract, the function `setMaxSystemDebt()` is used to set maximum system debt in a single `TroveManager` pool. However, there is no check to ensure the newly updated `maxSystemDebt` is lower than the current accumulated debt.

```
184    function setMaxSystemDebt(uint256 _maxSystemDebt) external onlyOwner {
185        maxSystemDebt = _maxSystemDebt;
186    }
```

Listing 2.22: TroveManager.sol

Impact The contract may suffer a denial of service problem.

Suggestion Add a check to ensure `maxSystemDebt` is larger than current total debt.

2.1.15 Conflicts of Updating rewardEndTime During Initialization of LPStakingPool

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the contract of `LPStakingPool`, global variable `rewardEndTime` can either be updated by internal function `updatePool()` or by privileged function `setRewardEndTime()`. However, the function `setRewardEndTime()` will invoke the `updatePool()` during the execution, which is inconsistent for the update of `rewardEndTime`.

```
48    function setRewardEndTime(uint256 _rewardEndTime) external onlyOwner {
49        updatePool();
50        require(rewardEndTime > block.timestamp, "invalid rewardEndTime");
51        rewardEndTime = _rewardEndTime;
52        emit RewardEndTimeUpdated(_rewardEndTime);
53    }
54    function updatePool() internal {
55        if (lastUpdateTime == 0) {
56            lastUpdateTime = block.timestamp;
57            rewardEndTime = block.timestamp + duration;
58        }
59        if (totalShares() == 0) {
60            return;
61        }
62        if (lastTimeRewardApplicable() == lastUpdateTime) {
63            return;
64        }
65        if (lastTimeRewardApplicable() > lastUpdateTime) {
66            uint256 pending = (lastTimeRewardApplicable() - lastUpdateTime) * rewardPerSec;
67            accRewardPerShare += pending * PRECISION / totalShares();
68            lastUpdateTime = lastTimeRewardApplicable();
69        }
70    }
```

Listing 2.23: LPStakingPool.sol

Impact Conflicts may arise when updating the `rewardEndTime` during the initialization of the `LPStakingPool`.

Suggestion Add a check to ensure the consistency of `rewardEndTime` between `updatePool()` and `setRewardEndTime()`.

2.1.16 Inappropriate Parameter Settings in initLockSettings

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract of `StakingBoost`, the internal function `initLockSettings()` will be invoked during the initialization process. However, the settings for these lock times are not consistent with the documentation.

```
26    function initLockSettings() internal {
27        _MaxId = type(uint256).max;
28        LockSettings.push(LockSetting(30 seconds, 10));
29        LockSettings.push(LockSetting(90 seconds, 20));
30        LockSettings.push(LockSetting(183 seconds, 50));
31        LockSettings.push(LockSetting(365 seconds, 100));
```

```
32    }
```

Listing 2.24: StakingBoost.sol

Impact The locking period is too short in the time unit of seconds, which is against the design of the documentation.

Suggestion Change the time unit from `seconds` to `days`.

2.1.17 Lack of Check in Function setMCR()

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `TroveManager` contract, the function `setMCR()` is used to set `Minimum Collateral Ratio` for individual troves `(MCR)`. However, there is no check to ensure the newly updated `MCR` is lower than `Critical System Collateral Ratio (CCR)` and larger than `110%`.

```
180    function setMCR(uint256 _MCR) external onlyOwner {
181        MCR = _MCR;
182    }
```

Listing 2.25: TroveManager.sol

Impact The contract may disobey the design purpose, which allows users to liquidate `troves` that are not supposed to be liquidated.

Suggestion Implement the checks mentioned above.

2.1.18 Incorrect Calculation of Debt Interest

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `TroveManager`, any user can assist a specified `borrower` in repaying interest through the function `repayInterestDebt()`. This function retrieves the `borrower`'s `debt`, followed by invoking the internal function `_repayInterest()`. Within function `_repayInterest()`, the function `getTroveInterest()` calculates interest based on the `_account`'s `rewardSnapshots` and the `borrower`'s `debt`, where `_account` can be different from the `borrower`.

```
832    function repayInterestDebt(address _borrower) external {
833        _distributeInterestDebt();
834        _applyPendingRewards(_borrower);
835        (uint256 debt, , , ) = getEntireDebtAndColl(_borrower);
836        _repayInterest(msg.sender, _borrower, debt);
837    }
```

Listing 2.26: TroveManager.sol

```
837    function _repayInterest(address _account, address _borrower, uint256 _debt) internal {
838        uint256 interest = getTroveInterest(_account, _debt);
839        if (WenUSD.balanceOf(_account) >= interest) {
840            WenUSD.burn(_account, interest);
841        } else {
842            totalActiveDebt += interest;
843            Troves[_borrower].debt += interest;
844        }
845        decreaseOutstandingInterestDebt(interest);
846        _updateTroveRewardSnapshots(_borrower);
847        emit InsterstPaid(_account, _borrower, interest);
848    }
```

Listing 2.27: TroveManager.sol

```
320    function getTroveInterest(address _borrower, uint256 debt) public view returns (uint256) {
321        return (debt * (getPendingInterestDebt() - rewardSnapshots[_borrower].interest)) /
               DECIMAL_PRECISION;
322    }
```

Listing 2.28: TroveManager.sol

Impact The calculation of interest is incorrect.

Suggestion In the function `getTroveInterest()`, replace parameter `_account` with the parameter `_borrower`.

2.2 Additional Recommendation

2.2.1 Incorrect Function Name

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `TroveManager`, the function `repayInterestDetb()` is named with a spelling error. It should be '`Debt`', instead of '`Detb`'.

```
823    function repayInterestDetb(address _borrower) external {
824       _distributeInterestDebt();
825       _applyPendingRewards(_borrower);
826       (uint256 debt, , , ) = getEntireDebtAndColl(_borrower);
827       _repayInterest(msg.sender, _borrower, debt);
828    }
```

Listing 2.29: TroveManager.sol

Suggestion I The function name should be changed to `repayInterestDebt`.

2.2.2 Inconsistency between Implementation and Comments

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `WenMath`, the comments mention that the function `_calcDecayedBaseRate()` from the contract `TroveManager` and the function `_getCumulativeIssuanceFraction()` from the contract `CommunityIssuance` both use the function `_decPow()` for calculations. However, the implementation of function `_getCumulativeIssuanceFraction()` in the contract CommunityIssuance cannot be found.

```
594    * Called by two functions that represent time in units of minutes:
595    * 1) TroveManager._calcDecayedBaseRate
596    * 2) CommunityIssuance._getCumulativeIssuanceFraction
597    function _decPow(uint256 _base, uint256 _minutes) internal pure returns {
598       if (_minutes > 525600000) {
599           _minutes = 525600000;
600       } // cap to avoid overflow
601
602
603       if (_minutes == 0) {
604           return DECIMAL_PRECISION;
605       }
606       uint256 y = DECIMAL_PRECISION;
607       uint256 x = _base;
608       uint256 n = _minutes;
609       // Exponentiation-by-squaring
610       while (n > 1) {
611           if (n % 2 == 0) {
612               x = decMul(x, x);
613               n = n / 2;
614           } else {
615               // if (n % 2 != 0)
616               y = decMul(x, y);
617               x = decMul(x, x);
618               n = (n - 1) / 2;
619           }
620       }
621       return decMul(x, y);
622   }
```

Listing 2.30: BorrowerOperations.sol

Suggestion Modify the implementation logic to align the code with the comments.

2.3 Note

2.3.1 Contract Supports Multiple Collateral Assets and Relies on Timely Updates from the Price Oracle

Introduced by `version 1`

Description According to the current implementation, the protocol supports several collateral tokens, and each collateral token corresponds to one market pool (i.e, `troveManager`). However, when measuring the health of the market, the `Total Collateralization Ratio` (`TCR`) is calculated based on the total collateral and debt in all market pools. This means that if a collateral token experiences a sharp price drop, the overall market health will also decline. However, due to the larger volume of other markets compared to a

single market, the overall decline in health may not be that large. It is possible that the system's protective mechanism (`Recovery Mode`) may not activate in a timely manner. If the `Oracle` fails to provide timely price feeds for the rapidly declining price, significant arbitrage opportunities can arise, further reducing the market's health. This can lead to a significant sell pressure of minted `WenUSD`, causing them to deviate from their peg and resulting in a series of other issues. Therefore, the protocol needs to carefully select tokens that can be used as collateral assets and ensure that the `Oracle` can provide timely and accurate price feeds.