BadgerDAO
Security Review

Cantina Managed review by:
hyh, Lead Security Researcher
Emanuele Ricci, Security Researcher

August 20, 2023

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Directly* exploitable security vulnerabilities that need to be fixed.
High	Security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All high issues should be addressed.
Medium	Objective in nature but are not security vulnerabilities. Should be addressed unless there is a clear reason not to.
Low	Subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. When determining the severity one first needs to determine whether the finding is subjective or objective. All subjective findings are considered of Minor severity.

Next it is determined whether the finding can be regarded as a security vulnerability. Some findings might be objective improvements that need to be fixed, but do not impact the project's security overall (Medium).

Finally, objective findings of security vulnerabilities are classified as either critical or major. Critical findings should be directly vulnerable and have a high likelihood of being exploited. Major findings on the other hand may require specific conditions that need to be met before the vulnerability becomes exploitable.

2 Security Review Summary

eBTC is a collateralized crypto asset soft pegged to the price of Bitcoin and built on the Ethereum network. It is backed exclusively by Staked Ether (stTEH) and powered by immutable smart contracts with minimized counterparty reliance. It's designed to be the most decentralized synthetic BTC in DeFi and offers the ability for anyone in the world to borrow BTC at no cost.

From August 2nd to August 11th the Cantina team conducted a review of ebtc on commit hash b2f641aa. The team identified a total of **14** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 3
- Medium Risk: 3
- Low Risk: 3
- Gas Optimizations: 0
- Informational: 5

3 Findings

3.1 High Risk

3.1.1 [BCCR] Many TCR increasing operations will be blocked when TCR is between 125 and 135

Severity: High Risk

Context: BorrowerOperations.sol#L404, BorrowerOperations.sol#L650

Description: It looks like after BCCR introduction some healthy TCR increasing openings and adjustments are now blocked.

If `oldTCR < newTCR`, but `ICR <= BCCR`, the operation can be denied if `TCR < BCCR`:

- BorrowerOperations.sol#L398-L413

```
        if (isRecoveryMode) {
            _requireICRisAboveCCR(vars.ICR);
        } else {
            _requireICRisAboveMCR(vars.ICR);
            uint newTCR = _getNewTCRFromCdpChange(vars.netColl, true, vars.debt, true, vars.price); // bools:
            ↪  coll increase, debt increase

>>          if (vars.ICR <= BUFFERED_CCR) {
                // Any open CDP is a debt increase so this check is safe

                // If you're dragging TCR toward buffer or RM, we add an extra check for TCR
                // Which forces you to raise TCR to 135+
                _requireNewTCRisAboveBufferedCCR(newTCR);
            } else {
                _requireNewTCRisAboveCCR(newTCR);
            }
        }
```

Similarly, when it's ICR increasing collateral withdrawal coupled with debt repayment, or debt increasing coupled with collateral posting, i.e. when one adjusts, making position less risky in the process, it will be blocked when `newTCR < BCCR`:

- BorrowerOperations.sol#L634-L660

```
        if (_isRecoveryMode) {
            ...
        } else {
            // if Normal Mode
            _requireICRisAboveMCR(_vars.newICR);
            _vars.newTCR = _getNewTCRFromCdpChange(
                ...
            );
>>          if ((_isDebtIncrease || _collWithdrawal > 0) && _vars.newICR <= BUFFERED_CCR) {
                // Adding debt or reducing coll has negative impact on TCR, do a stricter check

                // If you're dragging TCR toward buffer or RM, we add an extra check for TCR
                // Which forces you to raise TCR to 135+
                _requireNewTCRisAboveBufferedCCR(_vars.newTCR);
            } else {
                // Other cases have a laxer check
                _requireNewTCRisAboveCCR(_vars.newTCR);
            }
        }
```

Note, error message in these cases will incorrectly say about *TCR decreasing*:

- BorrowerOperations.sol#L670-L675

```
    function _requireNewTCRisAboveBufferedCCR(uint _newTCR) internal pure {
        require(
            _newTCR >= BUFFERED_CCR,
>>          "BorrowerOps: A TCR decreasing operation that would result in TCR < BUFFERED_CCR is not permitted"
        );
    }
```

Previous to BCCR introduction this was true as both checks happen in non-RM state and if now it's RM the TCR has to be decreased indeed. But once logic becomes 2-tiered it not the case.

It looks like both conditions should involve `&& newTCR < oldTCR`, as if it's not then nothing bad is happening, the system health increases, just not by that much that it's desired, but it might be impossible to achieve for small CDPs in big TVL conditions, which isn't a good reason to block those as system health would have strictly improved.

Impact: many CPD opening and, mostly importantly, CDP adjustment operations that makes the system healthier by increasing TCR will be denied. This is an issue both from UX perspective and protocol stability as the cumulative impact here is that such operations, mostly coming from small CDPs (which will constitute the bigger percentage of all accounts over time along with TVL growth) will not be carried out, so there will be a downward pressure on TCR compared to the situation before the change. I.e. some operations will be carried over with bigger funds brought in, but this will be less and less possible for an average CDP owner over time, and the bigger share of operations will be just cancelled. This will result in lower TCR.

Per high likelihood and medium impact setting the severity to be high.

Recommendation: Consider requiring the BCCR only when where was a decrease of TCR as a result of the operation, which was the initial rationale for buffer introduction, for example:

- BorrowerOperations.sol#L375-L413

```
-       bool isRecoveryMode = _checkRecoveryModeForTCR(_getTCR(vars.price));
+       uint oldTCR = _getTCR(vars.price);
+       bool isRecoveryMode = _checkRecoveryModeForTCR(oldTCR);

        vars.debt = _EBTCAmount;

        // Sanity check
        require(vars.netColl > 0, "BorrowerOperations: zero collateral for openCdp()!");

        uint _netCollAsShares = collateral.getSharesByPooledEth(vars.netColl);
        uint _liquidatorRewardShares = collateral.getSharesByPooledEth(LIQUIDATOR_REWARD);

        // ICR is based on the net coll, i.e. the requested coll amount - fixed liquidator incentive gas comp.
        vars.ICR = LiquityMath._computeCR(vars.netColl, vars.debt, vars.price);

        // NICR uses shares to normalize NICR across CDPs opened at different pooled ETH / shares ratios
        vars.NICR = LiquityMath._computeNominalCR(_netCollAsShares, vars.debt);

        /**
            In recovery move, ICR must be greater than CCR
            CCR > MCR (125% vs 110%)

            In normal mode, ICR must be greater thatn MCR
            Additionally, the new system TCR after the CDPs addition must be >CCR
        */
        if (isRecoveryMode) {
            _requireICRisAboveCCR(vars.ICR);
        } else {
            _requireICRisAboveMCR(vars.ICR);
            uint newTCR = _getNewTCRFromCdpChange(vars.netColl, true, vars.debt, true, vars.price); // bools:
            ↪  coll increase, debt increase

-           if (vars.ICR <= BUFFERED_CCR) {
+           // When new TCR is worse than before, it has to be above the buffer
+           if (vars.ICR <= BUFFERED_CCR && newTCR < oldTCR) {
                // Any open CDP is a debt increase so this check is safe

                // If you're dragging TCR toward buffer or RM, we add an extra check for TCR
                // Which forces you to raise TCR to 135+
                _requireNewTCRisAboveBufferedCCR(newTCR);
            } else {
                _requireNewTCRisAboveCCR(newTCR);
            }
        }
```

Similarly for `_requireValidAdjustmentInCurrentMode()`, assuming that `_vars.oldTCR` was populated before just as above via `_vars.oldTCR = _getTCR(vars.price)`:

- BorrowerOperations.sol#L634-L660

```
        if (_isRecoveryMode) {
            ...
        } else {
            // if Normal Mode
            _requireICRisAboveMCR(_vars.newICR);
            _vars.newTCR = _getNewTCRFromCdpChange(
                ...
            );
-           if ((_isDebtIncrease || _collWithdrawal > 0) && _vars.newICR <= BUFFERED_CCR) {
+           if ((_isDebtIncrease || _collWithdrawal > 0) && _vars.newTCR < _vars.oldTCR && _vars.newICR <=
↪   BUFFERED_CCR) {
                // Adding debt or reducing coll has negative impact on TCR, do a stricter check

                // If you're dragging TCR toward buffer or RM, we add an extra check for TCR
                // Which forces you to raise TCR to 135+
                _requireNewTCRisAboveBufferedCCR(_vars.newTCR);
            } else {
                // Other cases have a laxer check
                _requireNewTCRisAboveCCR(_vars.newTCR);
            }
        }
```

3.1.2 [BCCR] CDP closing can be used for whale sniping

Severity: High Risk

Context: BorrowerOperations.sol#L451-L475

Description: After BCCR introduction on CDP closing only `newTCR > CCR` is required, a whale sniper like scenario looks to be possible with an attacker preliminary opening two big CDPs (each can be a set of CDPs for granularity): first a `good` well capitalized one, at `goodICR >> BCCR`, and then one `bad`, at `badICR = MCR + epsilon1`, which is possible as `good` first one drags TCR up above BCCR.

Now, on observing any event that decreases the TCR, e.g. slashing (rare) or big enough oracle reading downtick (frequent enough), the attacker can front run the event transaction and remove `good` CDP (some combination of them), so that `newTCR = CCR + epsilon2`, and the event transaction triggers RM, which attacker back-runs with target liquidation.

Overall this requires CDP prepositioning and index/price update transaction sandwiching. Since keeping the said CDPs has low risk, the total cost for the attacker is gas and the cost of the capital involved. It can be viable given big enough expected profit up to `10%` of the whale collateral.

- BorrowerOperations.sol#L451-L475

```
    function closeCdp(bytes32 _cdpId) external override {
        ...

        uint newTCR = _getNewTCRFromCdpChange(
            collateral.getPooledEthByShares(coll),
            false,
            debt,
            false,
            price
        );
>>      _requireNewTCRisAboveCCR(newTCR);

        cdpManager.removeStake(_cdpId);
```

Impact: protocol manipulation in order to liquidate a specific CDP that isn't liquidatable by itself is possible via `closeCdp()`.

Per medium likelihood and high principal funds loss impact for CDP owner setting the severity to be high.

Recommendation: Since after BCCR introduction CDP adjusting is controlled at `135` TCR, while closing is at `125` TCR anyway (i.e. it's not a free exit), it might be reasonably to consider controlling it at `130` as a part of BCCR introduction. I.e. on the one hand it's not desirable to leave the surface open, on the another adding the very same control as for opening/adjusting might be too tight as closing is both more user sensitive operation and the cost and complexity of the attack is higher in this case.

3.1.3 Allowing the creation of "dust CDPs" could lead redeemers/liquidators to be not profitable or not wanting to perform the operation

Severity: High Risk

Context: BorrowerOperations.sol#L324-L329, LiquidationLibrary.sol#L365-L370

Description: When `collateral.getPooledEthByShares(DECIMAL_PRECISION) < DECIMAL_PRECISION` both the `BorrowerOperations._adjustCdpInternal` and `LiquidationLibrary._liquidateCDPPartially` allows the `collateral` balance of the CDP to go below the constant `MIN_NET_COLL` value of `2 stETH`.

This allows the possibility (voluntarily or involuntarily) to generate (by adjusting or partially liquidating CDPs) "dust CDPs" where the collateral and debt amount of the CDP is very low (dust level).

When those "dust CDPs" exists inside the protocol, the following scenarios could happen

- Redeemers (that, unlike liquidators, will not get the `LIQUIDATOR_REWARD` and can't directly choose the CDPs from which they want to redeem from) could incur in loss (collateral receives is less valuable than the gas cost of the operation) or simply decide that the operation is not worth the gas cost
- Liquidators, depending on `stETH share value` and gas cost, could incur in a loss or simply decide that the operation is not worth

Let's review how it's possible to create such "dust CDPs" and why redemptions and liquidations operations could not be profitable.

Create "dust CDP" by leveraging `BorrowerOperations._adjustCdpInternal`

In this scenario, the user/attacker creates a "dust CDP" by using adjusting the CDP coll/debt via the `BorrowerOperations._adjustCdpInternal` operation.

Let's set up an initial scenario just to showcase the test

- `1 BTC = 15 stETH <> 1 stETH = ~0,066666666 BTC`
- `1 stETH = 1 ETH`
- `1 stETH share = 1 stETH`

1) The protocol has some whale CDP opened with coll/debt ratio very high, this allows us to have `TCR >> CCR` just to be sure that we are in "normal mode" and that the next events do not trigger the "recovery mode"
2) `Alice` creates a CDP with a coll/debt ratio equal to `~MMCR`. In our example, she creates a CDP with
    1) `collateral = 10 stETH` (that is equal to `10 ether` of `stETH shares`)
    2) `debt = 0,606060606060606054 ether` of `eBTC`
3) Lido validators get slashed/incur penalties and `1 stETH shares = 1 stETH - 1 wei`. The share value has decreased of just `1 wei`
4) At this point `Alice` would be liquidable, but it is not relevant because she promptly performs the following action
    1) She repays most of her debt, leaving only `0.0000001 ether` of debt in her CDP. This was just an example number, she could go further down as much as she wants
    2) She withdraws as much of collateral as possible. The amount of collateral she can withdraw depends mostly on the debt she still has in her CDP because at the end of the operation the `ICR` must always be `> MCR`.

At the end of the scenario setup, Alice's CDP will be as following

- `coll = 0,000001650000000002` of `stETH shares`
- `debt = 0,0000001` of eBTC
- `ICR = 1100000000000666655` that is still `~> MCR`
- Her CDP is **not liquidable** at this time

Note that it is also possible to create "dust CDPs" by executing a partial liquidation.

Consequences on Redemption

When user call `CdpManager.redeemCollateral` they specify the amount of `eBTC` that they want to redeem for `collateral`. The execution will iterate over all the CDPs that are **not liquidable** starting from the CDP with **lower** ICR but still `ICR >= MCR`.

For each CDP, the redeemer will receive an amount of collateral (in `stETH shares`) equal to `collateral.getSharesByPooledEth( (singleRedemption.eBtcToRedeem * DECIMAL_PRECISION ) / _redeemColFromCdp._price )`

The redemption of the CDP can be **full** (CDP is closed) or **partial**, depending on the `CDP.debt` and the amount of remaining `eBTC` provided to redeem from the protocol.

Note 1: As opposed to a liquidation operation, the redeemer will **not receive** the gas stipend and will need to **pay** a **fee** on top of the collateral received (min `0.5%`, max `100%`). Note 2: As opposed to a liquidation operation, the redeemer **cannot** specify which CDPs they want to redeem from, they will always redeem from the CDP with lower ICR that is still above MCR.

If the amount of `stETH shares` received by the redeemer is less than the gas spent to execute the redeem operation, there are two consequences:

1) The redeemer will lose money on the operation
2) The redeemer won't perform the operation because it will lose money. The redeem operation is part of the soft-peg system of eBTC.

Can the redeem operation not be profitable?

@GalloDaSballo stated that the `redeemCollateral` operation should cost

- `138k` gas units on avg
- `256k` gas units on max

Let's assume the user calls `CdpManager.redeemCollateral` with an amount of eBTC enough just to redeem from the first available CDP to be redeemed from (we can generalize that more or less the cost to redeem fully 100 CDPs is equal to the same amount multiplied by 100)

The operation would cost:

- avg gas used `138k`, `150 gwei` gas cost -> `0,0207 ETH`
- max gas used `256k`, `150 gwei` gas cost -> `0,0384 ETH`
- avg gas used `138k`, `300 gwei` gas cost -> `0,0414 ETH`
- max gas used `256k`, `300 gwei` gas cost -> `0,0768 ETH`

Let's assume that

- `1 BTC = 15 stETH <> 1 stETH = ~0,066666666 BTC`
- `1 stETH = 1 ETH`
- `stETH share` drops to `1 ether - 1 wei` just to simulate the scenario
- assume that Badger does not have a fee on redemption (just to simplify things). The fee would just make the situation for the redeemer worst, if we are not profitable without the fee, we would not be profitable with the fee as well.

Let's use the data from the first scenario where to perform the operation we will use `138k gas` and that gas costs `~150 gwei`.

To be profitable (and probably it's not enough because after the redemption you want to do something with those shares like swapping them, repaying a flashloan and so on) the amount of `stETH` shares that you need to receive must be greater than `0,0207 ETH`.

This means that `collateral.getSharesByPooledEth(cdp.debt * DECIMAL_PRECISION / price)` -> `collateral.getSharesByPooledEth(cdp.debt * 1 ether / ~0,066666666 ether)` must be `>= ~0,0207 ETH` of `stETH shares`

Just to simply things, let's assume that Badger does not take a cut on that collateral (as we said, it takes a fee that goes from a min of `0.5%` to `100%`).

To be profitable (collateral received is greater than gas spent) the redeemer should need to receive at least `0,0207 ETH` so it would have to be able to redeem at least `0,0207 * 0,066666666` BTC of debt == `0,00138 BTC`.

Is it possible to adjust a CDP in a way that

- ICR is very near MCR (not liquidable, but will be selected first when redemptions happen)
- Very low (dust level) amount of debt

In the setup of the scenario, we have shown (and demonstrated in the foundry test at the very end) that it is possible to generate a "dust CDP" by executing adjusting the CDP balance. In the example, `Alice` was able to bring down the debt (and withdraw as much collateral as possible) to `0,0000001` of `eBTC` (note that she could have reduced even more the debt, aggravating the "dust level" of the CDP)

The result will be that the redeemer will pay (when gas is at `~150 gwei`) `~0,0207 ETH` to redeem `0,0000001 ether` of eBTC for `0,000001492499981999 ether` of `stETH shares`.

Consequences on Liquidation

The liquidation process is less affected by the "dust CDP" situation for different reason, but it can anyway lead to situations where

- The liquidator does not yield any profit from the liquidation (neutral result or loss of funds because of gas cost)
- The liquidator does not execute liquidation at all because there would be no profit (this is bad for the protocol)

The Liquidation scenario is less affected because

1) Liquidator does get a `GAS STIPEND` of `0.2 stETH` by fully liquidating the CDP. This `GAS STIPEND` is not rewarded if the liquidator performs a **partial liquidation**, and at this point we return to a situation similar to the redemption one
2) Liquidators can choose which CDP they want to liquidate by specifying a single CDP or an array of CDPs

Can the liquidation operation not be profitable?

Because the "dust CDP" has a tiny amount of collateral, it's safe to assume that the premium that the liquidator gets back from the liquidation process will not influence in the overall profitability of the operation.

We can state that the operation is not profitable when the `GAS STIPEND` is **not enough** to cover the gas cost of the transaction. This is influenced by two factors:

1) The **real** value of the `GAS STIPEND`. When a user opens a CDP, he needs to "pay" an additional `GAS STIPEND` that is equal to `0.2 stETH`. Those `0.2 stETH` are converted to `stETH shares` when the CDP is opened. The **real** value of those shares could end up not being `0.2 stETH` when it's returned to the liquidator because it depends on the current (at liquidation time) value of those shares. If the value has decreased (because of slashing/penalties on Lido) compared to the value at the time of the creation of the CDP (involved in the liquidation process) the liquidator will receive less than `0.2 stETH`. We also need to take in consideration that `stETH` is not always perfectly pegged to `ETH` and there have been cases where `stETH` was worth less than `ETH` (~7% less)
2) Gas price. Another factor to take in consideration is the gas price to execute the "direct" liquidation or the "MEV" liquidation (that could involve more complex operations like flashloaning, pre/post swaps, transfers and so on).

Liquidation Scenario 1: `stETH:BTC` **price decrease,** `stETH share` **value remain the same**

In the first scenario, we reduce the `stETH:BTC` price by a tiny fraction just to make `Alice` CDP be liquidable.

`Alice` CDP

- `coll = 0,000001650000000002` shares equal to `0,000001650000000001` of `stETH`
- `debt = 0,0000001` of `eBTC`

After the liquidation, the liquidator has collected `0,20000165 stETH`.

In this scenario, the `GAS STIPEND` rewarded to the liquidator is mostly the same as the one supplied by `Alice` at the CDP opening (share value has decreased by just `1 wei` in the meanwhile). This means that in this very specific case, the profitability of the operation depends on mostly by the gas cost of the execution itself.

From an analysis made by the Badger team, the gas consumption of a "MEV Liquidation" is around `650325 units of gas`.

- with gas cost of `~150 gwei` the liquidation would cost `~0.1 ETH`
- with gas cost of `~300 gwei` the liquidation would cost `~0.2 ETH`

With the gas cost of `~300 gwei` we are at the very limit of not being cost neutral, and we are not taking in consideration that the `stETH:ETH` could not be pegged or that the general value of `stETH` in the market has decreased

Liquidation Scenario 2: `stETH:BTC` **remains the same,** `stETH share` **value decreases**

In this scenario, the `stETH share` value has decreased because of a slash/penalty event on Lido. `1 stETH share` is now worth `0.9 stETH`. The price of `stETH:BTC` remain the same.

Because of the decrease in the share value, we are in this situation:

- `Alice` CDP `liquidatorRewardShares` **before** the slash was worth `~0.2 stETH`
- `Alice` CDP `liquidatorRewardShares` **after** the slash is worth `~0,18 stETH`

At the end of the liquidation process, the `liquidator` has received `~0,180001485` of `stETH` (gas stipend + collateral liquidated)

Compared to Scenario 1, in the Scenario 2 the gas price influences even more the profitability of the liquidation operation. If the gas cost to liquidate Alice's CDP is higher than `~0,180001485 stETH` the operation will only be a cost for the liquidator.

Test

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.17;
import "forge-std/Test.sol";
import {eBTCBaseInvariants} from "./BaseInvariants.sol";

contract FAdjustDustCDPTest is eBTCBaseInvariants {
    uint256 public constant MCR = 1100000000000000000; // 110%
    uint256 public constant DECIMAL_PRECISION = 1 ether;
    uint256 public constant MIN_COLLAERAL_SIZE = 2 ether;
    uint256 public constant GAS_STIPEND = 0.2 ether;

    address user1;
    address user2;
    bytes32 cdp1;
    bytes32 cdp2;

    function setUp() public override {
        super.setUp();

        // current value of the shares on lido (more or less)
        collateral.setEthPerShare(1 ether);
        priceFeedMock.setPrice((1 ether * DECIMAL_PRECISION) / 15 ether);

        connectCoreContracts();
        connectLQTYContractsToCore();
    }

    /// @notice the scope of this funcion is to prepare the test scenario
    /// 1) Warp at least 14 days to enable the redeem operation
    /// 2) Create two users with funds to open CDPs
    /// 3) First user open a CDP as a whale providing a lot of collateral to bring up TCR
    /// 4) Second user open a CDP with ICR ~= MCR
    /// 5) Eth Per Share goes from 1 ether -> 1 ether - 1 wei just to trigger the scenario
    /// 6) Second user adjust it's CDP to create a "dust CDP"
    function _createDustCDPFromAdjust(uint256 finalDustDebt) public {
        vm.warp(block.timestamp + 30 days);
```

```
        // let's say that I open a cdp with the minimum amount
        address payable[] memory users;
        users = _utils.createUsers(2);
        user1 = users[0];
        user2 = users[1];

        vm.label(user1, "user_1");
        vm.label(user2, "user_2");

        _dealCollateralAndPrepForUse(user1);
        _dealCollateralAndPrepForUse(user2);

        // let's say that we have a big whale that open a CDP
        // to bring the TCR high
        vm.startPrank(user1);
        uint256 borrowedAmount = _utils.calculateBorrowAmount(
            100 ether,
            priceFeedMock.fetchPrice(),
            COLLATERAL_RATIO
        );
        cdp1 = borrowerOperations.openCdp(borrowedAmount, "hint", "hint", 400 ether + GAS_STIPEND);
        vm.stopPrank();

        // user2 open the CDP just abobe MCR
        vm.startPrank(user2);
        borrowedAmount = _utils.calculateBorrowAmount(
            10 ether,
            priceFeedMock.fetchPrice(),
            MINIMAL_COLLATERAL_RATIO
        );
        cdp2 = borrowerOperations.openCdp(borrowedAmount, "hint", "hint", 10 ether + GAS_STIPEND);

        // share value decrease sub 1 ether
        collateral.setEthPerShare(1 ether - 1);

        // redeem and withdraw the max possible amount
        // how much do we want to leave as debt? the min possible?
        (uint256 cdp2Debt, uint256 cdp2Coll, ) = cdpManager.getEntireDebtAndColl(cdp2);
        uint256 removedDebt = cdp2Debt - finalDustDebt;
        borrowerOperations.repayEBTC(cdp2, removedDebt, "", "");

        (cdp2Debt, cdp2Coll, ) = cdpManager.getEntireDebtAndColl(cdp2);
        uint256 maxWithdrawColl = maxWithdraw(cdp2Coll, cdp2Debt);
        borrowerOperations.withdrawColl(cdp2, maxWithdrawColl, "", "");

        vm.stopPrank();
    }

    function testLiquidationNotProfitable() public {
        _createDustCDPFromAdjust(0.0000001 ether);

        // let's try to just fully liquidate it
        address liquidator = makeAddr("liquidator");

        (uint256 cdp2Debt, uint256 cdp2Coll, ) = cdpManager.getEntireDebtAndColl(cdp2);
        // send to redeemer just what's needed to fully redeem cdp2
        vm.prank(user1);
        eBTCToken.transfer(liquidator, cdp2Debt);

        ////////////////////////////////////////////////////////////////////////
        ////////////////////////////////////////////////////////////////////////
        // SUBSCENARIO 1: oracle price decrease, stETH share value does not decrease anymore
        // In this scenario as long as gas price are lower than 310 gwei the
        // operation should be still neutral/positive from a profit prospective
        // This depends also on the stETH price and peg to ETH
        // The Liquidator receive more or less ONLY the GAS STIPEND of ~0.2 stETH
        ////////////////////////////////////////////////////////////////////////
        ////////////////////////////////////////////////////////////////////////

        uint256 snapshot = vm.snapshot();

        // let's say that the price decrese just to make it liquidable
        priceFeedMock.setPrice(priceFeedMock.fetchPrice() - 0.0000000000001 ether);

        // In this case the liquidator is still profittable just because
        CdpInfo memory cdp2Before = getCdpInfo(cdp2);
```

```
        vm.prank(liquidator);
        cdpManager.liquidate(cdp2);

        CdpInfo memory cdp2After = getCdpInfo(cdp2);
        UserBalance memory liquidatorAfter = getUserBalance(liquidator);

        assertEq(liquidatorAfter.debt, 0);

        // note that cdp2 stETH collateral was just ~0,000001650000000001 stETH
        // that more or less are worth ~€0,002768007
        assertApproxEqAbs(liquidatorAfter.coll, GAS_STIPEND + cdp2Before.collStETH, 1);

        // Assert that cdp2 has been fully liquidated
        assertEq(cdp2After.status, 3); // closedByLiquidation

        vm.revertTo(snapshot);

        ////////////////////////////////////////////////////////////////////////
        ////////////////////////////////////////////////////////////////////////
        // SUBSCENARIO 2: oracle price decrease, stETH share value decrease even more
        // The main difference in this scenario is that the stETH shares
        // (used by protocol for accounting) are worth less (compared to scenario 1)
        // This influence not only how much is worth the collateral you are gaining from the liquidation
        // but also the GAS STIPEND that you gain from the full liquidation
        // The GAS STIPEND is equal to `0.2 stETH` but at the time that the CDP (liquidated) was opened
        // if during this period the `stETH share` has decreased in value, also the GAS STIPEND
        // (that is denominated in converted shares) has decreased in value
        // This mean that the liquidator will receive less than `0.2 stETH` for the GAS STIPEND
        ////////////////////////////////////////////////////////////////////////
        ////////////////////////////////////////////////////////////////////////

        // bring the stETH share value down more to prove our scenario
        collateral.setEthPerShare(0.9 ether);

        // Liquidation is much less profittable and gas cost to execute the transaction could
        // be above the gas stipend (because now stETH share is less valuable)
        cdp2Before = getCdpInfo(cdp2);
        vm.prank(liquidator);
        cdpManager.liquidate(cdp2);

        cdp2After = getCdpInfo(cdp2);
        liquidatorAfter = getUserBalance(liquidator);

        // Assert that the profit from the liquidation is less than the GAS STIPEND
        assertLt(liquidatorAfter.coll, GAS_STIPEND);

        // Assert that cdp2 has been fully liquidated
        assertEq(cdp2After.status, 3); // closedByLiquidation
    }

    function testRedeemNotProfitable() public {
        _createDustCDPFromAdjust(0.0000001 ether);

        // let's make things easier to track by having a separate redeemer user
        // that holds 0 collateral and just the amount of eBTC needed to fully redeem CDP2
        (uint256 cdp2Debt, uint256 cdp2Coll, ) = cdpManager.getEntireDebtAndColl(cdp2);

        address redeemer = makeAddr("redeemer");

        // send to redeemer just what's needed to fully redeem cdp2
        vm.prank(user1);
        eBTCToken.transfer(redeemer, cdp2Debt);

        // prepare variables for later asserts
        UserBalance memory redeemerBefore = getUserBalance(redeemer);
        CdpInfo memory cdp1Before = getCdpInfo(cdp1);

        // perform the redeem of CDP2
        vm.prank(redeemer);
        cdpManager.redeemCollateral(cdp2Debt, bytes32(0), bytes32(0), bytes32(0), 0, 0, 1e18);

        UserBalance memory redeemerAfter = getUserBalance(redeemer);
        CdpInfo memory cdp1After = getCdpInfo(cdp1);

        // CDP has not changed (redeem operation has only fullly redeemed CDP2)
        assertEq(cdp1Before.debt, cdp1After.debt);
```

```
        assertEq(cdp1Before.coll, cdp1After.coll);

        // redeemer had only the amount of eBTC needed for the redemption
        assertEq(redeemerBefore.debt, cdp2Debt);

        // redeemer now has 0 eBTC (all of them used)
        assertEq(redeemerAfter.debt, 0);

        // redeemer had no collateral before the operation
        assertEq(redeemerBefore.coll, 0);

        // CDP2 has been closed
        assertEq(cdpManager.getCdpStatus(cdp2), 4); // closedByRedemption

        // print the final amount of stETH gained by the reedeemer
        console.log("redeemerAfter.coll", redeemerAfter.coll);

        // in our test scenario gas cost for the operation is ~0,0207 ETH

        // assert that it was not profitable
        assertLt(redeemerAfter.coll, 0.0207 ether);
    }

    function calcStake(bytes32 cdpId) public view returns (uint256) {
        (uint realDebt, uint realColl, uint pendingEBTCDebtReward) = cdpManager.getEntireDebtAndColl(
            cdpId
        );

        return (realColl * cdpManager.totalStakesSnapshot()) / cdpManager.totalCollateralSnapshot();
    }

    function camputeICR(uint256 cdpCollShares, uint256 cdpDebt) public returns (uint256) {
        uint256 price = priceFeedMock.fetchPrice();
        uint256 coll = collateral.getPooledEthByShares(cdpCollShares);
        return (coll * price) / cdpDebt;
    }

    function getMaxLiquidableDebt(uint256 cdpDebt) public returns (uint256) {
        // needed for the check they are doing during partial liquidation
        uint256 price = priceFeedMock.fetchPrice();
        return cdpDebt - ((MIN_COLLAERAL_SIZE * price) / DECIMAL_PRECISION);
    }

    function maxWithdraw(uint256 cdpCollShares, uint256 cdpDebt) public returns (uint256) {
        // how much collateral can I withdraw to still be above MCR with the new ICR?
        uint256 price = priceFeedMock.fetchPrice();
        uint256 coll = collateral.getPooledEthByShares(cdpCollShares);
        uint256 maxWithdrawColl = coll - ((MCR * cdpDebt) / price);

        // remove 1 wei for rounding errors
        return maxWithdrawColl - 1;
    }

    function printSystem() public {
        console.log("=== PRINT SYSTEM ===");
        uint256 price = priceFeedMock.fetchPrice();
        uint256 tcr = cdpManager.getTCR(price);
        console.log("L_EBTCDebt     ->", cdpManager.L_EBTCDebt());
        console.log("stFeePerUnitg  ->", cdpManager.stFeePerUnitg());
        console.log("TCR            ->", tcr);
        console.log("RM ?           ->", tcr < CCR);

        console.log("");
    }

    function printCdp(bytes32 cdpId) public {
        (uint256 realDebt, uint256 realColl, ) = cdpManager.getEntireDebtAndColl(cdpId);

        uint256 price = priceFeedMock.fetchPrice();
        console.log("=== PRINT CDP ===");
        uint256 tcr = cdpManager.getTCR(price);
        uint256 icr = cdpManager.getCurrentICR(cdpId, price);
        bool liquidable = icr < MINIMAL_COLLATERAL_RATIO || (tcr < CCR && icr < tcr);
        uint256 coll = cdpManager.getCdpColl(cdpId);
        uint256 debt = cdpManager.getCdpDebt(cdpId);
```

```
        // try this
        coll = realColl;
        debt = realDebt;
        // try this

        uint collSt = collateral.getPooledEthByShares(coll);

        console.log("status   ->", cdpManager.getCdpStatus(cdpId));
        console.log("stake    ->", cdpManager.getCdpStake(cdpId));
        console.log("coll     ->", coll);
        console.log("coll st  ->", collSt);
        console.log("debt     ->", debt);
        console.log("ICR      ->", icr);
        console.log("liq?     -> ", liquidable);
        console.log("minColl? -> ", collSt < MIN_COLLAERAL_SIZE);
        console.log("");
    }

    function getCdpInfo(bytes32 cdpId) public view returns (CdpInfo memory) {
        (uint realDebt, uint realColl, ) = cdpManager.getEntireDebtAndColl(cdpId);
        return
            CdpInfo(
                cdpManager.getCdpStatus(cdpId),
                realColl,
                collateral.getPooledEthByShares(realColl),
                realDebt
            );
    }

    function getUserBalance(address user) public view returns (UserBalance memory) {
        uint256 debt = eBTCToken.balanceOf(user);
        uint256 coll = collateral.balanceOf(user);
        return UserBalance(coll, debt);
    }

    struct CdpInfo {
        uint status;
        uint256 coll;
        uint256 collStETH;
        uint256 debt;
    }

    struct UserBalance {
        uint256 coll;
        uint256 debt;
    }
}
```

Recommendation:

1. Badger should consider to **always** require, no matter what, that the CDP's collateral balance is **always** above the `MIN_NET_COLL` after the execution of `BorrowerOperations._adjustCdpInternal` and `LiquidationLibrary._liquidateCDPPartially`.

It looks like `_requirePartialLiqDebtSize()` and `_convertDebtDenominationToBtc()` functions will be no longer needed:

-LiquidationLibrary.sol#L960-L969

```
    function _requirePartialLiqDebtSize(
        uint _partialDebt,
        uint _entireDebt,
        uint _price
    ) internal view {
        require(
            (_partialDebt + _convertDebtDenominationToBtc(MIN_NET_COLL, _price)) <= _entireDebt,
            "LiquidationLibrary: Partial debt liquidated must be less than total debt"
        );
    }
```

- LiquityBase.sol#L106-L111

```
    // Convert debt denominated in ETH to debt denominated in BTC given that _price is ETH/BTC
    // _debt is denominated in ETH
    // _price is ETH/BTC
    function _convertDebtDenominationToBtc(uint _debt, uint _price) internal pure returns (uint) {
        return (_debt * _price) / DECIMAL_PRECISION;
    }
```

and can be removed (there are no other usages):

- LiquidationLibrary.sol#L340-L348

```
    function _liquidateCDPPartially(
        LocalVar_InternalLiquidate memory _partialState
    ) private returns (uint256, uint256) {
        bytes32 _cdpId = _partialState._cdpId;
        uint _partialDebt = _partialState._partialAmount;

        // calculate entire debt to repay
        LocalVar_CdpDebtColl memory _debtAndColl = _getEntireDebtAndColl(_cdpId);
-       _requirePartialLiqDebtSize(_partialDebt, _debtAndColl.entireDebt, _partialState._price);
```

2. Badger should also consider removing the dependency to `stETH` from the `LIQUIDATOR_REWARD` (gas stipend). By doing so, the reward for the liquidator will always be `0.2 ETH` without any influence from

- `stETH share value` at the moment of the liquidation
- `stETH` peg to `ETH`
- `stETH` value in the market

3.2 Medium Risk

3.2.1 `EBTCToken.transferFrom` decrease the allowance of `(owner, spender)` even when the allowance is set to `type(uint256).max`

Severity: Medium Risk

Context: EBTCToken.sol#L142-L144

Description: While it's not defined in the EIP-20, it's a common implementation (see both OpenZeppelin and Solmate) that the `transferFrom` function of an `ERC20` token does not decrease the allowance of the spender when such allowance has been set to the max value `type(uint256).max`.

The current implementation of `EBTCToken` does not follow this logic and decrease it by the `amount` transferred from the `sender` to the `recipient`

```
        unchecked {
            _approve(sender, msg.sender, cachedAllowances - amount);
        }
```

This behavior could create problems in contracts that are used to a more common behavior like the one used in OpenZeppelin/Solmate and have approved only once (without a way to update such value) the `EBTCToken`. The result is that at some point in the future, the `transferFrom` operation will revert because the `spender` would not have enough allowance anymore.

A contract that is already keen to this problem is `LeverageMacroReference`, an immutable contract that executes `ebtcToken.approve(_borrowerOperationsAddress, type(uint256).max);` only once when the constructor is executed.

At some point, an instance of the contract could not be able to perform operations like

- Adjusting the CDP (by repaying some `eBTC` debt)
- Close the CDP (by repaying all the CDP debt)
- Repaying the `eBTC` flashloaned `amount + fee`

Recommendation: BadgerDAO should follow the same logic used by the OpenZeppelin/Solmate implementation of the ERC20 token: the `transferFrom` function should not update the `spender` allowance if such allowance is equal to `type(uint256).max`

3.2.2 LeverageMacroBase's _doCheckValueType() condition looks to be incorrectly reverted

Severity: Medium Risk

Context: LeverageMacroBase.sol#L236-L247

Description: Given the usage, say for the `debt >= minDebt` check in

- LeverageMacroBase.sol#L175-L176

```
            _doCheckValueType(cdpInfo.debt, checkParams.expectedDebt);
            _doCheckValueType(cdpInfo.coll, checkParams.expectedCollateral);
```

it looks like `_doCheckValueType()` needs to compare first argument against the second according to the operator type.

Recommendation: Consider making the conditions reverted:

- LeverageMacroBase.sol#L236-L247

```
    /// @dev Assumes that
    ///     >= you prob use this one
    ///     <= if you don't need >= you go for lte
    ///     And if you really need eq, it's third
    function _doCheckValueType(uint256 valueToCheck, CheckValueAndType memory check) internal {
        if (check.operator == Operator.skip) {
            // Early return
            return;
        } else if (check.operator == Operator.gte) {
-           require(check.value >= valueToCheck, "!LeverageMacroReference: gte post check");
+           require(valueToCheck >= check.value, "!LeverageMacroBase: gte post check");
        } else if (check.operator == Operator.lte) {
-           require(check.value <= valueToCheck, "!LeverageMacroReference: let post check");
+           require(valueToCheck <= check.value, "!LeverageMacroBase: lte post check");
```

3.2.3 EIP-3156 requires flashFee() and maxFlashLoan() to accommodate their logic to flashLoansPaused flag

Severity: Medium Risk

Context: BorrowerOperations.sol#L816-L828, ActivePool.sol#L311-L332

Description: Per ERC-3156 `flashFee()` can revert, while `maxFlashLoan()` can return `0` when `flashLoansPaused == true`:

```
The maxFlashLoan function MUST return the maximum loan possible for token.
If a token is not currently supported maxFlashLoan MUST return 0, instead of reverting.

The flashFee function MUST return the fee charged for a loan of amount token.
If the token is not supported flashFee MUST revert.
```

Now the flag is ignored in the logic:

- BorrowerOperations.sol#L816-L828

```
    function flashFee(address token, uint256 amount) public view override returns (uint256) {
        require(token == address(ebtcToken), "BorrowerOperations: EBTC Only");

        return (amount * feeBps) / MAX_BPS;
    }

    /// @dev Max flashloan, exclusively in ETH equals to the current balance
    function maxFlashLoan(address token) public view override returns (uint256) {
        if (token != address(ebtcToken)) {
            return 0;
        }
        return type(uint112).max;
    }
```

On the same grounds as eBTC, `flashFee()` and `maxFlashLoan()` need to change their behavior when `flashLoansPaused == true` for ActivePool's stETH flash loans.

Impact: in both cases flash loan logic do not comply with EIP-3156 when `flashLoansPaused` is on, which can break the integrations.

Recommendation: `flashFee()` can revert, while `maxFlashLoan()` can return `0` when `flashLoansPaused == true` in both cases.

3.3 Low Risk

3.3.1 Do not allow the `LeverageMacroBase` to override the `approval` for `eBTC/stETH` during the `_doSwap` execution

Severity: Low Risk

Context: LeverageMacroBase.sol#L395-L398, LeverageMacroBase.sol#L417

Description: The `LeverageMacroReference` contracts that inherits from `LeverageMacroBase` executes some crucial approvals to allow the contract to repay the flashloan operations

```
// set allowance for flashloan lender/CDP open
ebtcToken.approve(_borrowerOperationsAddress, type(uint256).max);
stETH.approve(_borrowerOperationsAddress, type(uint256).max);
stETH.approve(_activePool, type(uint256).max);
```

Those approvals are done once and cannot be re-done by the `LeverageMacroReference`.

The `LeverageMacroBase._doSwap` allows the caller to specify both an arbitrary `tokenForSwap` token, `addressForApprove` address and `exactApproveAmount` that could allow the caller to override those approvals done during the `LeverageMacroReference` 'constructor.

If such an event happens, the `LeverageMacroReference` contract would not be able to repay any flashloan operation. Given that there is no way to re-do those initial approvals and that the flashloan operation is a hard requirement for the `doOperation` execution, it would mean that the `LeverageMacroReference` would probably not be able to execute any operation that involves the overridden token approval.

Recommendation: BadgerDAO should consider reverting the `_doSwap` operation when

- `tokenForSwap == ebtcToken` and `addressForApprove == borrowerOperationsAddress`
- `tokenForSwap == stETH` and `addressForApprove == borrowerOperationsAddress`
- `tokenForSwap == stETH` and `addressForApprove == activePool`

3.3.2 Outdated comments, error messages, naming across the codebase

Severity: Low Risk

Context: CdpManagerStorage.sol#L160-L163, CdpManagerStorage.sol#L328-L331, CdpManagerStorage.sol#L500-L502, PriceFeed.sol#L783-L789, LeverageMacroBase.sol#L26, PriceFeed.sol#L789-L806, BorrowerOperations.sol#L439-L442, BorrowerOperations.sol#L81-L92), BorrowerOperations.sol#L162-L165. BorrowerOperations.sol#L176-L178, BorrowerOperations.sol#L189-L191, BorrowerOperations.sol#L219-L223, BorrowerOperations.sol#L605-L624

Description: Comments, error messages, naming can be corrected, per list below.

Recommendation: `_closeCdpWithoutRemovingSortedCdps()` error message:

- CdpManagerStorage.sol#L160-L163

```
        require(
            closedStatus != Status.nonExistent && closedStatus != Status.active,
-           "CdpManagerStorage: close non-exist or non-active CDP!"
+           "CdpManagerStorage: close non-exist or active CDP!"
        );
```

Check is already done in the only caller above, `_closeCdpWithoutRemovingSortedCdps()`:

- CdpManagerStorage.sol#L328-L331

```
-       require(
-            cdpStatus != Status.nonExistent && cdpStatus != Status.active,
-           "CdpManagerStorage: remove non-exist or non-active CDP!"
-       );
```

Naming:

- CdpManagerStorage.sol#L500-L502

```
-   function _requireMoreThanOneCdpInSystem(uint CdpOwnersArrayLength) internal view {
+   function _requireMoreThanOneCdpInSystem(uint CdpIdsArrayLength) internal view {
        require(
-          CdpOwnersArrayLength > 1 && sortedCdps.getSize() > 1,
+          CdpIdsArrayLength > 1 && sortedCdps.getSize() > 1,
```

`_formatClAggregateAnswer()` description can substitute `stETH:BTC` feed that aren't used with `stETH:ETH` one in `_stEthEthAnswer` and `_stEthEthDecimals` params:

- PriceFeed.sol#L783-L789

```
    // @notice Returns the price of stETH:BTC in 18 decimals denomination
    // @param _ethBtcAnswer CL price retrieve from ETH:BTC feed
    // @param _stEthEthAnswer CL price retrieve from stETH:BTC feed
    // @param _ethBtcDecimals ETH:BTC feed decimals
    // @param _stEthEthDecimals stETH:BTC feed decimalss
    // @return The aggregated calculated price for stETH:BTC
    function _formatClAggregateAnswer(
```

LeverageMacroBase can implement `IERC3156FlashBorrower`:

- LeverageMacroBase.sol#L26

```
contract LeverageMacroBase {
```

`_formatClAggregateAnswer()` logic can be simplified to:

```
    (uint256(_ethBtcAnswer) * uint256(_stEthEthAnswer) * LiquityMath.DECIMAL_PRECISION) /
    10 ** (_stEthEthDecimals + _ethBtcDecimals)
```

Current implementation is a bit more prone to overflows (can break when bigger decimals number exceeds 30):

- PriceFeed.sol#L789-L806

```
    function _formatClAggregateAnswer(
        int256 _ethBtcAnswer,
        int256 _stEthEthAnswer,
        uint8 _ethBtcDecimals,
        uint8 _stEthEthDecimals
    ) internal view returns (uint256) {
        uint256 _decimalDenominator = _stEthEthDecimals > _ethBtcDecimals
            ? _stEthEthDecimals
            : _ethBtcDecimals;
        uint256 _scaledDecimal = _stEthEthDecimals > _ethBtcDecimals
            ? 10 ** (_stEthEthDecimals - _ethBtcDecimals)
            : 10 ** (_ethBtcDecimals - _stEthEthDecimals);
        return
            (_scaledDecimal *
                uint256(_ethBtcAnswer) *
                uint256(_stEthEthAnswer) *
                LiquityMath.DECIMAL_PRECISION) / 10 ** (_decimalDenominator * 2);
    }
```

These comments need to be removed or rewritten:

`closeCdp()` description:

- BorrowerOperations.sol#L439-L442

```
    /**
-   allows a borrower to repay all debt, withdraw all their collateral, and close their Cdp. Requires the
↪   borrower have a eBTC balance sufficient to repay their cdp's debt, excluding gas compensation - i.e.
↪   `(debt - 50)` eBTC.
+   allows a borrower to repay all debt, withdraw all their collateral, and close their Cdp.
    */
    function closeCdp(bytes32 _cdpId) external override {
```

- BorrowerOperations.sol#L81-L92

```
    constructor(
        ...
    ) LiquityBase(_activePoolAddress, _priceFeedAddress, _collTokenAddress) {
        // This makes impossible to open a cdp with zero withdrawn EBTC
        // TODO: Re-evaluate this
```

- BorrowerOperations.sol#L162-L165

```
    /**
    Withdraws `_collWithdrawal` amount of collateral from the callers Cdp. Executes only if the user has an
    ↪   active Cdp, the withdrawal would not pull the users Cdp below the minimum collateralization ratio, and
    ↪   the resulting total collateralization ratio of the system is above 150%.
    */
    function withdrawColl(
```

- BorrowerOperations.sol#L176-L178

```
    Issues `_amount` of eBTC from the callers Cdp to the caller. Executes only if the Cdp's collateralization
    ↪   ratio would remain above the minimum, and the resulting total collateralization ratio is above 150%.
     */
    function withdrawEBTC(
```

- BorrowerOperations.sol#L189-L191

```
    repay `_amount` of eBTC to the callers Cdp, subject to leaving 50 debt in the Cdp (which corresponds to
    ↪   the 50 eBTC gas compensation).
    */
    function repayEBTC(
```

- BorrowerOperations.sol#L219-L223

```
    /**
    enables a borrower to simultaneously change both their collateral and debt, subject to all the
    ↪   restrictions that apply to individual increases/decreases of each quantity with the following
    ↪   particularity: if the adjustment reduces the collateralization ratio of the Cdp, the function only
    ↪   executes if the resulting total collateralization ratio is above 150%. The borrower has to provide a
    ↪   `_maxFeePercentage` that he/she is willing to accept in case of a fee slippage, i.e. when a redemption
    ↪   transaction is processed first, driving up the issuance fee. The parameter is ignored if the debt is
    ↪   not increased with the transaction.
    */
    // TODO optimization candidate
    function adjustCdpWithColl(
```

- BorrowerOperations.sol#L605-L624

```
    function _requireValidAdjustmentInCurrentMode(
        bool _isRecoveryMode,
        uint _collWithdrawal,
        bool _isDebtIncrease,
        LocalVariables_adjustCdp memory _vars
    ) internal view {
        /*
         *In Recovery Mode, only allow:
         *
         * - Pure collateral top-up
         * - Pure debt repayment
         * - Collateral top-up with debt repayment
         * - A debt increase combined with a collateral top-up which makes the
         * ICR >= 150% and improves the ICR (and by extension improves the TCR).
         *
         * In Normal Mode, ensure:
         *
         * - The new ICR is above MCR
         * - The adjustment won't pull the TCR below CCR
         */
```

3.3.3 Last CDP cannot be closed and user cannot recover ~2.2 stETH from the CDP

Severity: Low Risk

Context: CdpManagerStorage.sol#L500-L505

Description: The system is designed to **not allow** the closure and removal of the last active CDP. This required and applied each time an operation (close, redeem, liquidate) tries to close a CDP.

Because of this check, that the last CDP can only be "adjusted" by the user. The user will be able to

- Repay at max `cdp.debt - 1 wei` of debt (repaying all the debt will revert the adjust operation)
- Withdraw at max `cdp.coll - 2 ether` of collateral (withdrawing more than that would revert because of the TCR check or because of the `MIN_NET_COLL` check)

The result is that the user will not be able to withdraw the `MIN_NET_COLL` amount of collateral and the `LIQUIDATOR_REWARD` amount of collateral provided at the creation of the CDP for a total of `~2.2 stETH` (the final amount of `stETH` depends on the conversion between `stETH shares` and `stETH` at the moment of the operation)

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.17;
import "forge-std/Test.sol";
import {eBTCBaseInvariants} from "./BaseInvariants.sol";

contract FastCloseTest is eBTCBaseInvariants {
    uint256 public constant MCR = 1100000000000000000; // 110%
    uint256 public constant DECIMAL_PRECISION = 1 ether;
    uint256 public constant MIN_COLLAERAL_SIZE = 2 ether;
    uint256 public constant GAS_STIPEND = 0.2 ether;

    address user1;
    address user2;
    bytes32 cdp1;
    bytes32 cdp2;

    function setUp() public override {
        super.setUp();

        // current value of the shares on lido (more or less)
        // collateral.setEthPerShare(1 ether);
        // priceFeedMock.setPrice((1 ether * DECIMAL_PRECISION) / 15 ether);

        connectCoreContracts();
        connectLQTYContractsToCore();
    }

    function testCloseLastCdpRevert() public {
        // Assume we are at a point where some users have opened the CDP, redempted, liquidated, closed and so
        ↪  on
        // We are at a point where there is only a user with an open CDP
```

```
    // The system is designed to revert always when a CDP is being closed but it's the last one active
    // This mean that the last CDP cannot be closed and the user won't be able to remove the MIN_NET_COLL
    // and LIQUIDATOR_REWARD for a total of ~2.2stETH (it depends on the share value at closing time)

    // setup the test
    address payable[] memory users;
    users = _utils.createUsers(2);
    user1 = users[0];
    vm.label(user1, "user_1");
    _dealCollateralAndPrepForUse(user1);

    // user1 creates a CDP
    uint256 borrowedAmount = _utils.calculateBorrowAmount(
        100 ether,
        priceFeedMock.fetchPrice(),
        CCR
    );
    vm.prank(user1);
    cdp1 = borrowerOperations.openCdp(borrowedAmount, "hint", "hint", 100 ether + GAS_STIPEND);

    // user1 tries to close the CDP but it reverts
    vm.prank(user1);
    vm.expectRevert("CdpManager: Only one cdp in the system");
    borrowerOperations.closeCdp(cdp1);
}

function testAdjustLastCdpFundsLeftInProtocol() public {
    // Assume we are at a point where some users have opened the CDP, redempted, liquidated, closed and so
    ↪  on
    // We are at a point where there is only a user with an open CDP
    // The system is designed to revert always when a CDP is being closed but it's the last one active
    // This mean that the last CDP cannot be closed and the user won't be able to remove the MIN_NET_COLL
    // and LIQUIDATOR_REWARD for a total of ~2.2stETH (it depends on the share value at closing time)

    // setup the test
    address payable[] memory users;
    users = _utils.createUsers(2);
    user1 = users[0];
    vm.label(user1, "user_1");
    _dealCollateralAndPrepForUse(user1);

    // user1 creates a CDP
    uint256 borrowedAmount = _utils.calculateBorrowAmount(
        100 ether,
        priceFeedMock.fetchPrice(),
        CCR
    );
    vm.prank(user1);
    cdp1 = borrowerOperations.openCdp(borrowedAmount, "hint", "hint", 100 ether + GAS_STIPEND);

    // calculate the final amount of collateral and debt of the CDP
    (uint256 debt, uint256 coll, ) = cdpManager.getEntireDebtAndColl(cdp1);

    // the user cannot repay ALL the debt because otherwise it would revert
    vm.prank(user1);
    borrowerOperations.repayEBTC(cdp1, debt - 1, "", "");

    // the user cannot withdraw all the collateral. at least MIN_NET_COLL must remain in the CDP
    vm.prank(user1);
    vm.expectRevert("BorrowerOperations: Cdp's net coll must be greater than minimum");
    borrowerOperations.withdrawColl(cdp1, coll - 2 ether + 1, "", "");

    vm.prank(user1);
    borrowerOperations.withdrawColl(cdp1, coll - 2 ether, "", "");

    // user only have 1 wei of eBTC in the balance
    assertEq(eBTCToken.balanceOf(user1), 1);

    // active pool still have the remaining stETH of the user (MIN_NET_COLL + LIQUIDATOR_REWARD)
    assertEq(collateral.balanceOf(address(activePool)), 2.2 ether);
}

function testDAOCDP() public {
    // one possible solution would be to have a DAO CDP that allows the last user to close the CDP
    // the problem with this DAO CDP is that it's just a "normal" CDP so at any point in time
    // it could be liquidated or redeemed (and closed) and we would be again at the start of the problem
```

```
        // setup the test
        address payable[] memory users;
        users = _utils.createUsers(2);
        user1 = users[0];
        address DAO = users[1];
        vm.label(user1, "user_1");
        vm.label(DAO, "DAO");
        _dealCollateralAndPrepForUse(user1);
        _dealCollateralAndPrepForUse(DAO);

        // DAO creates a CDP
        uint256 borrowedAmount = _utils.calculateBorrowAmount(
            100 ether,
            priceFeedMock.fetchPrice(),
            CCR
        );

        console.log("DAO open CDP");
        vm.prank(DAO);
        bytes32 cdpDAO = borrowerOperations.openCdp(
            borrowedAmount,
            "hint",
            "hint",
            100 ether + GAS_STIPEND
        );

        // user1 creates a CDP
        borrowedAmount = _utils.calculateBorrowAmount(100 ether, priceFeedMock.fetchPrice(), CCR);
        console.log("user1 open CDP");
        vm.prank(user1);
        cdp1 = borrowerOperations.openCdp(borrowedAmount, "hint", "hint", 100 ether + GAS_STIPEND);

        // user now can close it, the DAO CDP cannot be closed
        vm.prank(user1);
        borrowerOperations.closeCdp(cdp1);
    }
}
```

Recommendation: BadgerDAO could consider opening a healthy CDP to allow the last CDP (of the user) to be correctly closed. BadgerDAO needs to consider that this "DAO CDP" is anyway considered as a "normal CDP" and can be at any time liquidated or redeemed if the platform allows it. If this happens, we would return to the start of the scenario, where the last user won't be able to correctly close the CDP.

3.4 Informational

3.4.1 `LeverageMacroBase` does not allow the owner of the contract to perform CDP operations without performing a Flashloan

Severity: Informational

Context: LeverageMacroBase.sol#L161

Description: The current implementation of `LeverageMacroBase` reverts if the `doOperation` does not execute a `stETH` or `eBTC` flashloan operation.

There could be cases for which the owner of the contract could want to be able to adjust or close the CDP without performing a flashloan.

Recommendation: BadgerDAO should consider allowing the caller of `LeverageMacroBase.doOperation` to perform the `LeverageMacroOperation operation` without executing a flashloan.

3.4.2 `LeverageMacroBase.sweepToCaller` could transfer `stETH shares` instead of "pure" `stETH` to avoid leaving dust into the contract

Severity: Informational

Context: LeverageMacroBase.sol#L226

Description: From the Lido official documentation about `stETH/wstETH` integration guide, there is a specific paragraph about "1-2 wei corner case".

> stETH balance calculation includes integer division, and there is a common case when the whole stETH balance can't be transferred from the account while leaving the last 1-2 wei on the sender's account. The same thing can actually happen at any transfer or deposit transaction. In the future, when the stETH/share rate will be greater, the error can become a bit bigger. To avoid it, one can use transferShares to be precise.

Lido itself suggests using `transferShares` instead of `transferFrom` to avoid this `1-2 wei corner case`.

Recommendation: BadgerDAO could consider using `stETH.transferShares` instead of `stETH.transfer` in the `LeverageMacroBase.sweepToCaller` execution. If BadgerDAO decides to chose so, it should also update `collateralBal` value properly get the correct share balance of the contract by executing `stETH.sharesOf(address(this));`

3.4.3 Leverage contracts should fetch the protocol contract addresses from `cdpManager` instead of being passed as constructor parameter

Severity: Informational

Context: LeverageMacroBase.sol#L51-L68, LeverageMacroDelegateTarget.sol#L41-L60, LeverageMacroFactory.sol#L21-L35, LeverageMacroReference.sol#L17-L42

Description: All the `Leverage*` contracts (`LeverageMacroBase`, `LeverageMacroDelegateTarget`, `LeverageMacroFactory` and `LeverageMacroReference`) are initializing the `immutable` variables of `borrowerOperations`, `activePool`, `cdpManager`, `ebtcToken`, `sortedCdps` and `stETH` with the corresponding value passed down from the contract's `constructor`.

To reduce the human error and the possibility that the wrong value is passed to such `constructor`, BadgerDAO could consider to only passing the `cdpManager` contract into the `constructor` and then gathering all the other addresses by querying the `cdpManager` itself.

- `borrowerOperations` can be fetched by calling `cdpManager.borrowerOperationsAddress()`
- `activePool` can be fetched by calling `cdpManager.activePool()`
- `ebtcToken` can be fetched by calling `cdpManager.ebtcToken()`
- `sortedCdps` can be fetched by calling `cdpManager.sortedCdps()`
- `stETH` can be fetched by calling `cdpManager.collateral()`

Recommendation: BadgerDAO should consider gathering the addresses needed to initialize `borrowerOperations`, `activePool`, `ebtcToken`, `sortedCdps` and `stETH` from the `cdpManager` instead of passing them down as `constructor` inputs.

This would reduce the possibility of human error or of a misconfiguration of the `Leverage*` contracts (`LeverageMacroBase`, `LeverageMacroDelegateTarget`, `LeverageMacroFactory` and `LeverageMacroReference`)

3.4.4 Use the correct function "State Mutability" when needed

Severity: Informational

Context: LeverageMacroBase.sol#L334, LeverageMacroBase.sol#L423, LeverageMacroBase.sol#L438, LeverageMacroDelegateTarget.sol#L63, LeverageMacroReference.sol#L44

Description: Here's the list of the function that could be declared with a more specific and restrictive "State Mutability":

- `LeverageMacroBase.decodeFLData` can be declared as `pure`
- `LeverageMacroBase._doSwapChecks` can be declared as `view`
- `LeverageMacroBase._ensureNotSystem` can be declared as `view`
- `LeverageMacroDelegateTarget.owner` can be declared as `view`
- `LeverageMacroReference.owner` can be declared as `view`

Recommendation: BadgerDAO should consider replacing the function "state mutability" property with the suggested one.

3.4.5 Replace `.transfer`, `.transferFrom` and `.approve` with the corresponding ERC20 "safe" version of them

Severity: Informational

Context: LeverageMacroBase.sol#L226, LeverageMacroBase.sol#L395-L398, LeverageMacroBase.sol#L417, LeverageMacroReference.sol#L40-L41, BorrowerOperations.sol#L533, ActivePool.sol#L276, ActivePool.sol#L285, ActivePool.sol#L288

Description: While it's true that `stETH` is following the `ERC20` standard and the `approve`, `transfer` and `transferFrom` on their token always returns `true` or reverts it's recommended to anyway, use the corresponding "safe" version of those function to be bullet-proof for the future. Lido could decide at some point to upgrade their `stETH` token to an implementation that does not follow the `ERC20` standard anymore.

The suggestion is even more highly recommended when such operations are performed on an external and arbitrary `ERC20` token.

Note that the current implementation of `stETH` `.transferShares` and `.transferSharesFrom` do not return `bool` value, but instead the amount of `stETH` tokens that have been moved from the `sender` to the `recipient` account.

Recommendation: BadgerDAO should consider replacing the `.transfer`, `.transferFrom` and `.approve` calls with the corresponding ERC20 "safe" version of them.