BadgerDAO Security Review

Auditors

0xLeastwood, Lead Security Researcher

hyh, Lead Security Researcher

Emanuele Ricci, Security Researcher

Calvbore, Junior Security Researcher

Hagrid, Junior Security Researcher

Report prepared by: Pablo Misirov

August 20, 2023

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

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.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of ebtc according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 16 days in total, Badger-Finance engaged with Spearbit to review the ebtc protocol. In this period of time a total of **61** issues were found.

Summary

Project Name	Badger-Finance
Repository	ebtc
Commit	be8062...3084
Type of Project	Stablecoin, DeFi
Audit Timeline	June 21 to July 12
Two week fix period	July 12 - Jan 26

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	9	3	6
Medium Risk	8	3	5
Low Risk	16	4	12
Gas Optimizations	3	0	3
Informational	25	7	18
Total	**61**	**17**	**44**

5 Findings

5.1 High Risk

5.1.1 `_openCdp` could end up inserting the CDP in `SortedCdp` in the incorrect position

Severity: High Risk

Context: BorrowerOperations.sol#L384, BorrowerOperations.sol#L417

Description: If `_checkDeltaIndexAndClaimFee` has not triggered the splitting fee logic that updates the `stFeePerUnitg` state variable and `collateral.getPooledEthByShares(DECIMAL_PRECISION)` is indeed greater compared to the last one stored in `stFeePerUnitg`, the `sortedCdps.insert` could end up inserting the new CDP in the wrong position.

Many logics of the `eBTC` protocol rely on the invariant that each CDP in `SortedCdps` contract is always dynamically and correctly sorted.

If before querying or updating the `SortedCdps` state the `stFeePerUnitg` is not correctly updated, the function ends up with a not-correctly sorted linked list or an invalid query value.

Recommendation: BadgerDAO should always execute `claimStakingSplitFee` before any operation logic to be sure to have updated `stFeePerUnitg` to the most up-to-date value and have correctly executed the "splitting fee" logic.

BadgerDAO: The recommendations have been implemented in PR 521.

Spearbit: Fixed.

5.1.2 HintHelpers's getRedemptionHints can return incorrect collateral and eBTC amounts and avoids valid partial redemptions when collateral remainder is low but is above minimum

Severity: High Risk

Context: HintHelpers.sol#L150-L163

Description: HintHelpers's `getRedemptionHints()` cancels the partial redemption due when CDP debt is above the amount needed if it would leave the CDP with less than the minimum collateral.

However, the comparison used isn't correct as `partialRedemptionNewColl`, being in shares, is compared to `stETH` amount `MIN_NET_COLL`:

- HintHelpers.sol#L101-L119

```
    // If this CDP has more debt than the remaining to redeem, attempt a partial redemption
    if (currentCdpDebt > vars.remainingEbtcToRedeem) {
        uint _cachedEbtcToRedeem = vars.remainingEbtcToRedeem;
>>      (partialRedemptionNewColl, partialRedemptionHintNICR) = _calculatePartialRedeem(
            vars,
            currentCdpDebt,
            _price
        );

        // If the partial redemption would leave the CDP with less than the minimum allowed coll, bail
        ↪  out of partial redemption and return only the fully redeemable
        // TODO: This seems to return the original coll? why?
>>      if (partialRedemptionNewColl < MIN_NET_COLL) {
            partialRedemptionHintNICR = 0; //reset to 0 as there is no partial redemption in this case
            vars.remainingEbtcToRedeem = _cachedEbtcToRedeem;
            break;
        }
    } else {
        vars.remainingEbtcToRedeem = vars.remainingEbtcToRedeem - currentCdpDebt;
    }
```

- HintHelpers.sol#L150-L163

```
>>  function _calculatePartialRedeem(
        ...
    ) internal view returns (uint, uint) {
        ...
        if (_oldIndex < _newIndex) {
            newColl = _getCollateralWithSplitFeeApplied(vars.currentCdpId, _newIndex, _oldIndex);
        } else {
            (, newColl, ) = cdpManager.getEntireDebtAndColl(vars.currentCdpId);
        }

        vars.remainingEbtcToRedeem = vars.remainingEbtcToRedeem - maxRedeemableEBTC;
        uint collToReceive = collateral.getSharesByPooledEth(
            (maxRedeemableEBTC * DECIMAL_PRECISION) / _price
        );

>>      uint _newCollAfter = newColl - collToReceive;
        return (
>>          _newCollAfter,
```

Also, `partialRedemptionNewColl` should return the full collateral of the CDP if the partial redemption won't happen because of the `MIN_NET_COLL` check.

Right now it will return a partial value that didn't pass the check and will be misleading because the partial redemption doesn't happen according to the logic.

- HintHelpers.sol#L56-L69

```
    function getRedemptionHints(
        uint _EBTCamount,
        uint _price,
        uint _maxIterations
    )
        external
        view
        returns (
            bytes32 firstRedemptionHint,
            uint partialRedemptionHintNICR,
            uint truncatedEBTCamount,
>>          uint partialRedemptionNewColl
        )
    {
```

- HintHelpers.sol#L101-L117

```
                // If this CDP has more debt than the remaining to redeem, attempt a partial redemption
                if (currentCdpDebt > vars.remainingEbtcToRedeem) {
                    uint _cachedEbtcToRedeem = vars.remainingEbtcToRedeem;
>>                  (partialRedemptionNewColl, partialRedemptionHintNICR) = _calculatePartialRedeem(
                        vars,
                        currentCdpDebt,
                        _price
                    );

                    // If the partial redemption would leave the CDP with less than the minimum allowed
                    ↪  coll, bail out of partial redemption and return only the fully redeemable
                    // TODO: This seems to return the original coll? why?
>>                  if (partialRedemptionNewColl < MIN_NET_COLL) {
                        partialRedemptionHintNICR = 0; //reset to 0 as there is no partial redemption
                        ↪  in this case
                        vars.remainingEbtcToRedeem = _cachedEbtcToRedeem;
                        break;
                    }
                }
```

Also, if partial redemption is successful then `vars.remainingEbtcToRedeem` should be set to zero as all was redeemed, but it never happens:

- HintHelpers.sol#L92-L123

```
            while (
                vars.currentCdpUser != address(0) &&
                vars.remainingEbtcToRedeem > 0 &&
                _maxIterations-- > 0
            ) {
                // Apply pending debt
                uint currentCdpDebt = cdpManager.getCdpDebt(vars.currentCdpId) +
                    cdpManager.getPendingEBTCDebtReward(vars.currentCdpId);

                // If this CDP has more debt than the remaining to redeem, attempt a partial redemption
>>              if (currentCdpDebt > vars.remainingEbtcToRedeem) {  // @audit this CDP is all what's
↪  needed
                    uint _cachedEbtcToRedeem = vars.remainingEbtcToRedeem;
                    (partialRedemptionNewColl, partialRedemptionHintNICR) = _calculatePartialRedeem(
                        vars,
                        currentCdpDebt,
                        _price
                    );

                    // If the partial redemption would leave the CDP with less than the minimum allowed
                    ↪  coll, bail out of partial redemption and return only the fully redeemable
                    // TODO: This seems to return the original coll? why?
                    if (partialRedemptionNewColl < MIN_NET_COLL) {
                        partialRedemptionHintNICR = 0; //reset to 0 as there is no partial redemption
                        ↪  in this case
>>                      vars.remainingEbtcToRedeem = _cachedEbtcToRedeem;  // @audit rewind the amount
↪  when bailed out
                        break;
>>                  }  // @audit but when the partial redemption is ok, why remainingEbtcToRedeem is
↪  left as is?
                } else {
                    vars.remainingEbtcToRedeem = vars.remainingEbtcToRedeem - currentCdpDebt;
                }

                vars.currentCdpId = sortedCdps.getPrev(vars.currentCdpId);
                vars.currentCdpUser = sortedCdps.getOwnerAddress(vars.currentCdpId);
            }
```

Because of that the `truncatedEBTCamount = _EBTCamount - vars.remainingEbtcToRedeem` will be incorrect:

- HintHelpers.sol#L117-L127

```
                } else {
                    vars.remainingEbtcToRedeem = vars.remainingEbtcToRedeem - currentCdpDebt;
                }

                vars.currentCdpId = sortedCdps.getPrev(vars.currentCdpId);
                vars.currentCdpUser = sortedCdps.getOwnerAddress(vars.currentCdpId);
            }
        }

>>      truncatedEBTCamount = _EBTCamount - vars.remainingEbtcToRedeem;
    }
```

Impact: Partial redemptions from CDPs that have less than `2e18` shares remaining will be rejected, while getRedemptionHints() will return incorrect `partialRedemptionNewColl` and `truncatedEBTCamount` amounts.

For the rejection part as of now it is about 13% mistake (`getPooledEthByShares(1e18) = 1.131651731942226`), which will increase along with index growth. Incorrect amounts will be returned even when a partial redemption itself was treated correctly.

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

Recommendation: All points combined:

- HintHelpers.sol#L103-L116

```
    uint _cachedEbtcToRedeem = vars.remainingEbtcToRedeem;
    (partialRedemptionNewColl, partialRedemptionHintNICR) = _calculatePartialRedeem(
        vars,
        currentCdpDebt,
        _price
    );

    // If the partial redemption would leave the CDP with less than the minimum allowed coll, bail out
    ↪  of partial redemption and return only the fully redeemable
    // TODO: This seems to return the original coll? why?
-   if (partialRedemptionNewColl < MIN_NET_COLL) {
+   if (collateral.getPooledEthByShares(partialRedemptionNewColl) < MIN_NET_COLL) {
        partialRedemptionHintNICR = 0; //reset to 0 as there is no partial redemption in this case
+       partialRedemptionNewColl = 0;
        vars.remainingEbtcToRedeem = _cachedEbtcToRedeem;
-       break;
-   }
+   } else {
+       vars.remainingEbtcToRedeem = 0;
+   }
+   break;
```

BadgerDAO: Fixed in PR 513.

Spearbit: Fix looks good for all three parts of the issue.

5.1.3 `CdpManager.redeemCollateral` is not collecting "staking fee" before executing the redeem logic

Severity: High Risk

Context: CdpManager.sol#L325-L453

Description: The current implementation of `CdpManager.redeemCollateral` is not calling `claimStakingSplitFee()` that claim the staking fees and updated the global fee index `stFeePerUnitg`.

Because `claimStakingSplitFee` is not triggered, both the `TCR` and individual CDP ICR will return an outdated value. This means that

- The redeem is executed even if the protocol is in Recover Mode (`_requireTCRoverMCR` does not revert when the protocol is indeed in Recovery Mode).
- The block of code that selects the first CDP to be redeemed from (the one with the lower ICR above MCR) is incorrectly calculated.

Recommendation: BadgerDAO should update the `redeemCollateral` function to execute `claimStakingSplitFee()` as soon as possible.

Spearbit: Acknowledged.

5.1.4 `RolesAuthority.canCall` allows users with allowed role to execute the `function` at `target` location, even if the Capability has been burned

Severity: High Risk

Context: RolesAuthority.sol#L79-L94

Description: The `RolesAuthority.canCall` function

```
function canCall(
    address user,
    address target,
    bytes4 functionSig
) public view virtual override returns (bool) {
    CapabilityFlag flag = capabilityFlag[target][functionSig];

    return
        (flag != CapabilityFlag.Burned && flag == CapabilityFlag.Public) ||
        bytes32(0) != getUserRoles[user] & getRolesWithCapability[target][functionSig];
}
```

if implemented correctly, should follow this logic:

- if `flag == CapabilityFlag.None` only allows executing the call if the user has a role with the capability (`bytes32(0) != getUserRoles[user] & getRolesWithCapability[target][functionSig]`).
- if `flag == CapabilityFlag.Burned` return false directly.
- if `flag == CapabilityFlag.Public` return true directly.

The current implementation of the function does not indeed follow the logic explained above and is allowing users with a valid authenticated role to execute the `functionSig` function at `target` even when the capability has been **burned**.

Let's see an example to better understand the problem. Let's assume that the user has a role and that role is enabled for `capabilityFlag[target][functionSig]`. This means that `bytes32(0) != getUserRoles[user] & getRolesWithCapability[target][functionSig]` will return `TRUE`.

Now let's see what happens with the different cases of `capabilityFlag[target][functionSig]`

- if `flag == CapabilityFlag.None` → `(flag != CapabilityFlag.Burned && flag == CapabilityFlag.Public)` is equal to `false`. The function returns anyway `true` because the second part is equal to `true`. Correct.
- if `flag == CapabilityFlag.Burned` → `(flag != CapabilityFlag.Burned && flag == CapabilityFlag.Public)` is equal to `false`. The function returns anyway `true` because the second part is equal to `true`. **INCORRECT**: If the `capabilityFlag` has been burned, it should **always** return **false** no matter what.
- if `flag == CapabilityFlag.Public` → `(flag != CapabilityFlag.Burned && flag == CapabilityFlag.Public)` is equal to `true`. The function returns directly `true`. Correct.

Recommendation: BadgerDAO should reimplement the function to follow the straightforward logic explained below:

- if `flag == CapabilityFlag.Burned` return false directly
- if `flag == CapabilityFlag.Public` return true directly
- if `flag == CapabilityFlag.None` only allows executing the call if the user has a role with the capability (`bytes32(0) != getUserRoles[user] & getRolesWithCapability[target][functionSig]`)

BadgerDAO: The recommendations have been implemented in PR 513.

Spearbit: Fixed.

5.1.5 Anyone can call `TellorCaller.setFallbackTimeout` and update `timeOut` to an arbitrary value

Severity: High Risk

Context: TellorCaller.sol#L59-L63

Description: The `timeOut` value returned by the `fallbackTimeout()` function in `TellorCaller` is used by `PriceFeed` to check whether the fallback oracle is frozen or not

```
function _fallbackIsFrozen(FallbackResponse memory _fallbackResponse) internal view returns (bool) {
    return
        _fallbackResponse.timestamp > 0 &&
        _responseTimeout(_fallbackResponse.timestamp, fallbackCaller.fallbackTimeout());
}

function _responseTimeout(uint256 _timestamp, uint256 _timeout) internal view returns (bool) {
    return block.timestamp - _timestamp > _timeout;
}
```

The current implementation of `TellorCaller` allows anyone to call `setFallbackTimeout(uint256 _newFallbackTimeout)` and update `timeOut` to an arbitrary value.

- If `timeOut` is equal to `0`, `PriceFeed` will automatically discard the answer of the oracle because it will be automatically and forcefully considered frozen
- The more the `timeOut` value is lower, the higher is the % that `PriceFeed` will consider the FallbackOracle frozen. Considering that for how Tellor is queried, all the answers will be at least `tellorQueryBufferSeconds` seconds old, so any `timeOut` value lower than that will automatically set the Fallback Oracle as frozen by the `PriceFeed`
- If `timeOut` is set to a very high number, it's possible that `PriceFeed` will accept an answer even if it was stale

Recommendation: `BadgerDAO` should

1) Setup proper authorizations flag to prevent anyone from calling the `setFallbackTimeout`.
2) Considering adding proper lower and upper boundaries to the value that `timeOut` can be updated to.

BadgerDAO: Confirm tellor code will be removed due to other security reasons from the system itself.

Spearbit: Acknowledged.

5.1.6 `PriceFeed` will store and use Chainlink price even if not "fully validated" when the Fallback oracle cannot be trusted or is not configured

Severity: High Risk

Context: PriceFeed.sol#L226-L235, PriceFeed.sol#L292-L296

Description: In some part of the `PriceFeed` if the Fallback oracle is not configured (equal to `address(0)`) or cannot be trusted (broken or frozen) the Chainlink price is directly stored and used without validating if the Chainlink current price differs more than `MAX_PRICE_DEVIATION_FROM_PREVIOUS_ROUND` compared to the previous round.

Without this check, prices that are 100x more or less (as an example) will be blindly trusted, stored and used by the platform without further check.

Recommendation: BadgerDAO should consider to always having a Fallback oracle configured to further validate the Chainlink answer for this scenario. If the Fallback oracle is configured but cannot be trusted and the Chainlink price diff from the previous round is above `MAX_PRICE_DEVIATION_FROM_PREVIOUS_ROUND`, BadgerDAO cannot fully trust the Chainlink price even if it was correct in reality.

BadgerDAO: Keeping as nofix at this time, we're talking about changes to the design of the Price Feed and are interested in suggestions.

Spearbit: Acknowledged.

5.1.7 Fallback oracle price if blindly trusted without further check about previous price delta difference

Severity: High Risk

Context: PriceFeed.sol#L148, PriceFeed.sol#L178, PriceFeed.sol#L218C20-L218C39, PriceFeed.sol#L273, PriceFeed.sol#L289, PriceFeed.sol#L312

Description: Unlike the Chainlink price response that is validated against the previous round and a max price difference compared to the previous round, the price response from the fallback oracle is blindly trusted (when the Fallback oracle is not broken or frozen) and stored directly via `_storeFallbackPrice` without further check on a possible price difference compared to the latest Fallback oracle value.

This is particularly critical in the case where the Chainlink price diff of the previous round is above the `MAX_PRICE_DEVIATION_FROM_PREVIOUS_ROUND` and the current price diff compared to the Fallback price is above `MAX_PRICE_DIFFERENCE_BETWEEN_ORACLES`. In this case, the Fallback price is blindly trusted and stored. In the case where the Fallback price was the one incorrect, `BadgerDAO` is going to store an incorrect price that deviates compared to the **real one**.

Recommendation: BadgerDAO should consider validating further the price returned by the Fallback oracle.

BadgerDAO: Acknowledged, this can be addressed in the fallback oracle as well.

Spearbit: Acknowledged.

5.1.8 `PriceFeed` initial price from Chainlink is not fully validated

Severity: High Risk

Context: PriceFeed.sol#L86

Description: When both oracles (Chainlink and Fallback) are correctly working, the Chainlink price is validated against a max delta that the current price cannot surpass compared to the last round price.

This check is not performed when the `PriceFeed` oracle is deployed and if Chainlink is not broken or frozen, the last round price is directly stored in `lastGoodPrice` even if the price in the current round is 100x the value compared to the previous round.

If at this point the `PriceFeed` enters in one of the paths that returns `lastGoodPrice`, the `PriceFeed` will return a price that has not been fully validated whether it was correct or not and the `eBTC` platform will trust it blindly.

Recommendation: BadgerDAO should

1) If during `constructor` time the Fallback oracle has not been configured and `_chainlinkPriceChangeAboveMax(chainlinkResponse, prevChainlinkResponse) == true` the system should revert because the Chainlink price cannot be fully validated and trusted.
2) If during `constructor` time the Fallback oracle is configured, the Chainlink price should be further validated and should follow the full-validation that is executed inside `fetchPrice` and revert the deployment of `PriceFeed` if the Chainlink price cannot be trusted.

Spearbit: Acknowledged.

5.1.9 `PriceFeed` could return a stale price

Severity: High Risk

Context: PriceFeed.sol#L116, PriceFeed.sol#L124, PriceFeed.sol#L137, PriceFeed.sol#L144, PriceFeed.sol#L157, PriceFeed.sol#L164, PriceFeed.sol#L206, PriceFeed.sol#L214, PriceFeed.sol#L253, PriceFeed.sol#L262, PriceFeed.sol#L269, PriceFeed.sol#L280, PriceFeed.sol#L285, PriceFeed.sol#L300, PriceFeed.sol#L320, PriceFeed.sol#L325, PriceFeed.sol#L344

Description: The `PriceFeed` contract scope is to return the most up-to-date and correct price for the `stETH:BTC` pair. The contract relies on three different oracles

- Chainlink `stETH:ETH` oracle.
- Chainlink `ETH:BTC` oracle.
- Tellor `stETH:BTC` oracle.

Chainlink is the main source of price and when the Chainlink oracles cannot be trusted (oracle response is broken, oracle timeouts or the price is too high between rounds) the protocol switches to a fallback oracle (Tellor).

When both the oracles cannot be trusted, the system is always returning the value stored in `lastGoodPrice` that stores the latest valid price returned by one of the two oracle system. Note that the contract is not currently storing also when `lastGoodPrice` was updated (or to be more precise, when it has been updated on the oracle side) and which is the source of the price (Chainlink or Tellor).

Because we do not know when `lastGoodPrice` was stored, we do not know if that value is "fresh" enough to be trusted or if it's a **stale** value that should be discarded and the whole system paused. If the price of `stETH:BTC` provided by `PriceFeed` is stale, it could deviate by a lot (or anyway a significant amount) compared to the **real market price**. Because the protocol blindly trusts the response of `PriceFeed.fetchPrice` it could end up using a stale and wrong price.

Recommendation: BadgerDAO should consider store:

- When `lastGoodPrice` was updated.
- When the source oracle has updated the price stored in `lastGoodPrice`.
- Which is the source of `lastGoodPrice`.

If both the Oracles are untrusted and the `lastGoodPrice` is too stale, BadgerDAO should consider pausing all the actions that rely on the `stETH:BTC` price provided by `PriceFeed`.

BadgerDAO: Nofixing at this time as we believe it's best to keep the protocol alive and for people to be able to exit rather than have the protocol stuck.

Spearbit: Acknowledged.

5.2 Medium Risk

5.2.1 Chainlink `priceFeed.getRoundData(lastRoundId - 1)` could return a "false broken" answer, triggering the primary oracle broken path

Severity: Medium Risk

Context: PriceFeed.sol#L723-L737, PriceFeed.sol#L739-L753

Description: Useful links

- ETH/USD Proxy Price Feed.
- ETH/USD Price Aggregator used by the Proxy in this phase (`phaseID` 6).
- Official Chainlink Documentation about "Getting Historical Data".

When the `PriceFeed` contracts execute the `fetchPrice` function, it gathers the current Chainlink response and the previous one by executing

```
int256 ethBtcAnswer;
    int256 stEthEthAnswer;
    try ETH_BTC_CL_FEED.getRoundData(_currentRoundEthBtcId - 1) returns (
        uint80 roundId,
        int256 answer,
        uint256,
        /* startedAt */
        uint256 timestamp,
        uint80 /* answeredInRound */
    ) {
        ethBtcAnswer = answer;
        prevChainlinkResponse.roundEthBtcId = roundId;
        prevChainlinkResponse.timestampEthBtc = timestamp;
    } catch {
        // If call to Chainlink aggregator reverts, return a zero response with success = false
        return prevChainlinkResponse;
    }

    try STETH_ETH_CL_FEED.getRoundData(_currentRoundStEthEthId - 1) returns (
        uint80 roundId,
        int256 answer,
        uint256,
        /* startedAt */
        uint256 timestamp,
        uint80 /* answeredInRound */
    ) {
        stEthEthAnswer = answer;
        prevChainlinkResponse.roundStEthEthId = roundId;
        prevChainlinkResponse.timestampStEthEth = timestamp;
    } catch {
        // If call to Chainlink aggregator reverts, return a zero response with success = false
        return prevChainlinkResponse;
    }
```

The `roundID` returned by `latestRoundData` and `getRoundData` is not the `roundID` of the Price Aggregator, but is the `roundID` of the Price Feed Proxy that internally will call the aggregator.

The "`proxyRoundId`" is a composed value that internally stores two information:

- `phaseID`: which was the `phaseId` when the query was made. By knowing the phase, we can understand which is the aggregator that has been used.
- `aggregatorRoundId`: which is the internal `roundId` from which the price comes from. This makes sense only for the aggregator used in the phase.

There could be multiple cases where executing `priceFeed.getRoundData(currentProxyRoundId - 1)` will return an invalid answer

- The proxy has switched to a new `PriceAggregator` and the `phaseId` inside the Proxy has increased.
- The `currentProxyRoundId - 1` would revert for underflow or return a wrong answer because `aggregatorRoundId` is equal to the first valid ID.

If the `currentProxyRoundId - 1` produces an invalid `roundId`, the Price Feed Proxy will return an invalid answer, but this does not mean that the **real** previous data on the proxy does not exist or that a **real** error has occurred. In those cases, the `eBTC Price Feed` will interpret it as an exception and will not trust the Chainlink oracle, falling back to the `FallbackOracle` if available or returning the `lastGoodPrice` that, as we already mentioned in other issues, could be a **stale price**.

Recommendation: Currently, there is not a clear way to easily and correctly retrieve the *safe* `previousRoundId`. BadgerDAO should monitor those reverts event to understand if the revert was *real* or because of a miscalculation of the `previousRoundId`.

Spearbit: Acknowledged.

5.2.2 BorrowerOperations CDP manipulation functions can use TRC based on an outdated stETH index

Severity: Medium Risk

Context: CdpManager.sol#L795-L809

Description: Whenever `_checkDeltaIndexAndClaimFee()` doesn't update `stFPPSg`, the CDP manipulation functions of BorrowerOperations, `_adjustCdpInternal()`, `_openCdp()` and `closeCd()`, will use an outdated TCR for decision making.

`stFPPSg` can end up not updated by `_checkDeltaIndexAndClaimFee() -> checkIfDeltaIndexTriggerRM()`:

- CdpManager.sol#L795-L809

```
// @dev return current TCR for given price and true if delta index is big enough to trigger recovery
↪  mode, otherwise false.
function checkIfDeltaIndexTriggerRM(uint _price) external view override returns (uint, bool) {
    uint _oldIndex = stFPPSg;
    uint _newIndex = collateral.getPooledEthByShares(DECIMAL_PRECISION);
    if (_newIndex > _oldIndex) {
        (uint _requiredDelta, uint _tcr) = _computeDeltaIndexToTriggerRM(
            _newIndex,
            _price,
            stakingRewardSplit
        );
        return (_tcr, (_newIndex - _oldIndex) >= _requiredDelta);
    } else {
        return (_getTCR(_price), false);
    }
}
```

But it is the only call CDP manipulation functions make before running the logic:

- BorrowerOperations.sol#L364-L385

```
    function _openCdp(
        uint _EBTCAmount,
        bytes32 _upperHint,
        bytes32 _lowerHint,
        uint _collAmount,
        address _borrower
    ) internal returns (bytes32) {
        require(_collAmount > 0, "BorrowerOps: collateral for CDP is zero");
        _requireNonZeroDebt(_EBTCAmount);

        LocalVariables_openCdp memory vars;

        // ICR is based on the net coll, i.e. the requested coll amount - fixed liquidator incentive
        ↪  gas comp.
        vars.netColl = _getNetColl(_collAmount);

        _requireAtLeastMinNetColl(vars.netColl);

        vars.price = priceFeed.fetchPrice();

        // Reverse ETH/BTC price to BTC/ETH
>>      uint _tcr = _checkDeltaIndexAndClaimFee(vars.price);
>>      bool isRecoveryMode = _checkRecoveryModeForTCR(_tcr);
```

- BorrowerOperations.sol#L460-L466

```
    function closeCdp(bytes32 _cdpId) external override {
        _requireCdpOwner(_cdpId);

        _requireCdpisActive(cdpManager, _cdpId);
        uint price = priceFeed.fetchPrice();
>>      uint _tcr = _checkDeltaIndexAndClaimFee(price);
>>      _requireNotInRecoveryMode(_tcr);
```

- BorrowerOperations.sol#L250-L269

```
    function _adjustCdpInternal(
        bytes32 _cdpId,
        uint _collWithdrawal,
        uint _EBTCChange,
        bool _isDebtIncrease,
        bytes32 _upperHint,
        bytes32 _lowerHint,
        uint _collAddAmount
    ) internal {
        _requireCdpOwner(_cdpId);

        LocalVariables_adjustCdp memory vars;

        _requireCdpisActive(cdpManager, _cdpId);

        vars.price = priceFeed.fetchPrice();

        // Reversed BTC/ETH price
>>      uint _tcr = _checkDeltaIndexAndClaimFee(vars.price);
>>      bool isRecoveryMode = _checkRecoveryModeForTCR(_tcr);
```

Impact varies between the functions, and can change on any logic updates. I.e. it can be fine to run an outdated TCR at the moment, but this might change on an unrelated logic update later on. For this end categorizing the overall impact as medium.

Per medium likelihood (as `_checkDeltaIndexAndClaimFee()` do not update the `stFPPSg` in a narrow range of cases, but still it is not a low probability assumption being a part of standard workflow) and impact setting the severity to be medium.

Recommendation: Consider unconditionally updating `stFPPSg` with the current stETH index before proceeding with the protocol logic in `_adjustCdpInternal()`, `_openCdp()` and `closeCd()`.

BadgerDAO: Fixed in PR 521.

Spearbit: Fix looks ok, `claimStakingSplitFee()` is now called unconditionally in `_openCdp()`, while `applyPendingRewards()` (also runs `claimStakingSplitFee` and updates CDP's coll and debt readings) is also called unconditionally in `_adjustCdpInternal()` and `closeCd()` before reading the TRC.

5.2.3 Partial liquidation misses CdpUpdated event for bad debt redistribution state change

Severity: Medium Risk

Context: LiquidationLibrary.sol#L374-L384

Description: `CdpUpdated` event is missed for the bad debt redistribution update during partial liquidation, as there two events are due: a missed one for the debt figure update, and another for reduction as a result of partial liquidation itself which `_reInsertPartialLiquidation()` performs.

In other cases across the protocol the event for bad debt redistribution is emitted by `_applyPendingRewards()`:

- CdpManagerStorage.sol#L239-L269

```
    function _applyPendingRewards(bytes32 _cdpId) internal {
        ...

            // Compute pending rewards
            uint pendingEBTCDebtReward = _getRedistributedEBTCDebt(_cdpId);

            Cdp storage _cdp = Cdps[_cdpId];
>>          uint prevDebt = _cdp.debt;
            uint prevColl = _cdp.coll;

            // Apply pending rewards to cdp's state
>>          _cdp.debt = prevDebt + pendingEBTCDebtReward;

            _updateCdpRewardSnapshots(_cdpId);

            address _borrower = ISortedCdps(sortedCdps).getOwnerAddress(_cdpId);
            emit CdpUpdated(
                _cdpId,
                _borrower,
>>              prevDebt,
                prevColl,
>>              Cdps[_cdpId].debt,
                prevColl,
                Cdps[_cdpId].stake,
                CdpManagerOperation.applyPendingRewards
            );
        }
    }
```

But in `_liquidateCDPPartially(()` the bad debt redistribution update is performed directly (`_applyPendingRewards()` isn't called, the update is in-lined) and silently, `_reInsertPartialLiquidation()` uses already renewed debt value as an old one for `CdpUpdated`:

- LiquidationLibrary.sol#L388-L393

```
            _reInsertPartialLiquidation(
                _partialState,
                LiquityMath._computeNominalCR(newColl, newDebt),
>>              _debtAndColl.entireDebt,
>>              _debtAndColl.entireColl
            );
```

As `_debtAndColl` is after update state:

- LiquidationLibrary.sol#L348

```
LocalVar_CdpDebtColl memory _debtAndColl = _getEntireDebtAndColl(_cdpId);
```

- LiquidationLibrary.sol#L976-L991

```
    function getEntireDebtAndColl(
        bytes32 _cdpId
    ) public view returns (uint debt, uint coll, uint pendingEBTCDebtReward) {
        debt = Cdps[_cdpId].debt;
        (uint _feeSplitDistributed, uint _newColl) = getAccumulatedFeeSplitApplied(
            _cdpId,
            stFeePerUnitg,
            stFeePerUnitgError,
            totalStakes
        );
>>      coll = _newColl;

        pendingEBTCDebtReward = getPendingEBTCDebtReward(_cdpId);

>>      debt = debt + pendingEBTCDebtReward;
    }
```

- LiquidationLibrary.sol#L460-L494

```
    function _reInsertPartialLiquidation(
        LocalVar_InternalLiquidate memory _partialState,
        uint _newNICR,
>>      uint _oldDebt,
        uint _oldColl
    ) internal {
        ...
        emit CdpUpdated(
            _cdpId,
            sortedCdps.getOwnerAddress(_cdpId),
>>          _oldDebt,
            _oldColl,
            Cdps[_cdpId].debt,
            Cdps[_cdpId].coll,
            Cdps[_cdpId].stake,
            CdpManagerOperation.partiallyLiquidate
        );
    }
```

Impact:

CDP change events are one of the base protocol monitoring venues and omitting one can yield losses for the corresponding CDP owners, who can miss threshold changes of their positions this way.

Per high likelihood (no low probability prerequisites there) and low impact setting the severity to be medium.

Recommendation: As `_liquidateCDPPartially()` performs the bad debt redistribution update directly consider emitting the `CdpUpdated` event there, e.g. at this point:

- LiquidationLibrary.sol#L374-L384

```
        // apply pending debt if any
        // and update CDP internal accounting for debt
        // if there is liquidation redistribution
        {
            if (_debtAndColl.pendingDebtReward > 0) {
                Cdps[_cdpId].debt = Cdps[_cdpId].debt + _debtAndColl.pendingDebtReward;
            }
        }

>>      // @audit save Cdps[_cdpId].debt before the change and emit CdpUpdated here

        // updating the CDP accounting for partial liquidation
        _partiallyReduceCdpDebt(_cdpId, _partialDebt, _partialColl);
```

Alternatively, supply the saved old `Cdps[_cdpId].debt` figure to `_reInsertPartialLiquidation()` for emitting the cumulative change.

BadgerDAO: PR 513.

Spearbit: Fix looks ok, `_reInsertPartialLiquidation()` is now run with `_cachedDebt = Cdps[_cdpId].debt` before the pending update, so old event now covers the cumulative change.

5.2.4 All the operations that change the CDP collateral should verify that the new collateral is `>= MIN_NET_COLL`

Severity: Medium Risk

Context: BorrowerOperations.sol#L324-L328, LiquidationLibrary.sol#L370-L372

Description: Each CDPs has an explicit invariant that should be respected: `collateral.getPooledEthByShares(CDP.collateral) >= 2 stETH`

Every operation that changes the CDP collateral should always require that the invariant remain true:

- Open CDP
- Adjust CDP
- Partially redeem CDP
- Partially liquidate CDP

If the invariant is not held, the function should revert.

`LiquidationLibrary._liquidateCDPPartially` and `BorrowerOperations._adjustCdpInternal` are currently checking the invariant only when `collateral.getPooledEthByShares(1e18) >= 1e18`.

Recommendation: BadgerDAO should update both `LiquidationLibrary._liquidateCDPPartially` and `BorrowerOperations._adjustCdpInternal` to always check the invariant.

Spearbit: Acknowledged.

5.2.5 `CdpManager.setBeta` has no sanity check

Severity: Medium Risk

Context: CdpManager.sol#L898-L903

Description: The current implementation of `CdpManager.setBeta` has no sanity checks on the input parameter `_beta`. If `_beta` is equal to `0` the `_updateBaseRateFromRedemption` called during the redeem process will revert making the whole redeem process always revert.

Recommendation: BadgerDAO should at least add a sanity check that prevents the caller from setting `beta` equal to zero to avoid the revert in the redeem process.

BadgerDAO should consider adding some basic lower/upper checks to the values that `beta` can assume when `setBeta` is called.

BadgerDAO: Acknowledged.

Spearbit: Acknowledged.

5.2.6 CdpManager's redeem compares shares with stETH amount and cancels valid partial redemptions when collateral remainder is low but is above minimum

Severity: Medium Risk

Context: CdpManager.sol#L210-L213

Description: CdpManager's `_redeemCollateralFromCdp()` cancels the partial redemption either when the provided hint is out of date or when collateral is below the minumum, which is currently defined as `2 stETH`:

- LiquityBase.sol#L34-L35

```
// Minimum amount of stETH collateral a CDP must have
uint public constant MIN_NET_COLL = 2e18;
```

However, `newColl` coming from internal accounting and is set in shares:

- CdpManager.sol#L171-L180

```
        // Repurposing this struct here to avoid stack too deep.
        LocalVar_CdpDebtColl memory _oldDebtAndColl = LocalVar_CdpDebtColl(
            Cdps[_redeemColFromCdp._cdpId].debt,
            Cdps[_redeemColFromCdp._cdpId].coll,
            0
        );

        // Decrease the debt and collateral of the current Cdp according to the EBTC lot and
        ↪  corresponding ETH to send
        uint newDebt = _oldDebtAndColl.entireDebt - singleRedemption.eBtcToRedeem;
>>      uint newColl = _oldDebtAndColl.entireColl - singleRedemption.stEthToRecieve;
```

The shares vs `stETH` amount `newColl < MIN_NET_COLL` check is then performed:

- CdpManager.sol#L210-L213

```
>>          if (newNICR != _redeemColFromCdp._partialRedemptionHintNICR || newColl < MIN_NET_COLL) {
                singleRedemption.cancelledPartial = true;
                return singleRedemption;
            }
```

Impact: Partial redemptions from CDPs that have less than `2e18` shares remaining will be rejected.

As of now it is about `13%` mistake (`getPooledEthByShares(1e18) = 1.131651731942226`), which will increase along with index growth.

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

Recommendation: Consider converting shares to stETH amount before the check

- CdpManager.sol#L210-L213

```
-   if (newNICR != _redeemColFromCdp._partialRedemptionHintNICR || newColl < MIN_NET_COLL) {
+   if (newNICR != _redeemColFromCdp._partialRedemptionHintNICR ||
↪   collateral.getPooledEthByShares(newColl) < MIN_NET_COLL) {
        singleRedemption.cancelledPartial = true;
        return singleRedemption;
    }
```

BadgerDAO: Resolved in PR 513.

Spearbit: Fix looks ok.

5.2.7 `setRedemptionFeeFloor` allows setting `redemptionFeeFloor` to `DECIMAL_PRECISION`, making the `redeem` operation to always revert.

Severity: Medium Risk

Context: CdpManager.sol#L702-L706, CdpManager.sol#L866-L878

Description: `setRedemptionFeeFloor` allows an authorized user to update the `redemptionFeeFloor` state variable to a value up to `DECIMAL_PRECISION` (100% fees).

The `redemptionFeeFloor` variable is used to calculate the % of the collateral redeemed by the `redeem` operation that will be taken by the protocol.

Higher the `redemptionFeeFloor` is and higher is the probability that the check performed in `_calcRedemptionFee` will make the redeem operation revert.

```
function _calcRedemptionFee(uint _redemptionRate, uint _ETHDrawn) internal pure returns (uint) {
    uint redemptionFee = (_redemptionRate * _ETHDrawn) / DECIMAL_PRECISION;
    require(redemptionFee < _ETHDrawn, "CdpManager: Fee would eat up all returned collateral");
    return redemptionFee;
}
```

The `_calcRedemptionFee` function correctly checks that the protocol does not "eat up all the collateral" redeemed by the user. The input variable `_redemptionRate` is calculated by adding `_baseRate` to the `redemptionFeeFloor` in the `_calcRedemptionRate` function, and the result is maxed out to `DECIMAL_PRECISION`.

As we said, the higher `redemptionFeeFloor` is, the higher is the chance that the redeem function will revert. If `redemptionFeeFloor` is set to `DECIMAL_PRECISION`, the redeem operation will **always** revert, preventing users from executing the operation.

Recommendation: BadgerDAO should consider reverting `setRedemptionFeeFloor` if `_redemptionFeeFloor` is equal to `DECIMAL_PRECISION`. BadgerDAO should also consider decreasing the upper bound of `_redemptionFeeFloor` that right now is `DECIMAL_PRECISION` in order to reduce the possibility of making the redeem operation to revert (because the protocol would "eat up all the collateral in fees").

If setting `redemptionFeeFloor` equal to `DECIMAL_PRECISION` is a proper way to **disable** the redeem operation, BadgerDAO should properly and carefully document this behavior and think about using a more explicit logic to achieve the purpose, like using a `pauseRedeem` flag.

BadgerDAO: Added separate pause flag PR 549.

Spearbit: The `redemptionsPaused` pause flag has been added, but the `setRedemptionFeeFloor` has not been changed. You can still have set it to DECIMAL_PRECISION or anyway a value near DECIMAL_PRECISION that will make the redeem operation revert. Are you planning to add a proper upper bound to the function input parameter _redemptionFeeFloor?

Marked as acknowledged.

5.2.8 `AuthNoOwner._initializeAuthority` and `setAuthority` should revert if `newAuthority` is equal to `address(0)`

Severity: Medium Risk

Context: AuthNoOwner.sol#L47-L55, AuthNoOwner.sol#L30-L44

Description: The `AuthNoOwner` contract is inherited by many contracts in the `eBTC` ecosystem, and its main purpose it configures an `authority` system that allows to manage with a fine level of granularity which function can be called by whom.

`_initializeAuthority` is the `internal` function that must be called by the contract that inherit from `AuthNoOwner` and its main purpose is to configure the initial `authority` and set the contract as initialized.

If the input parameter `newAuthority` is equal to `address(0)` we have the following consequences

- the contract is initialized and `_initializeAuthority` cannot be called anymore.
- no one will be able to execute the functions that use the modifier `requiresAuth`. `isAuthorized` will always return `false`.
- no one will be able to call `setAuthority` to change the authority because `authority.canCall(...)` will revert.

`setAuthority` is a utility function that allows an authorized user to swap the current authority with a new one. If the input parameter `newAuthority` is equal to `address(0)` we will have the same problem that we have already listed for `_initializeAuthority`:

- no one will be able to execute the functions that use the modifier `requiresAuth`. `isAuthorized` will always return `false`.
- no one will be able to call `setAuthority` to change the authority because `authority.canCall(...)` will revert.

Recommendation: BadgerDAO should

- revert in `_initializeAuthority` if `newAuthority` is equal to `address(0)`.
- change the first parameter of `emit AuthorityUpdated` to `msg.sender` in the `_initializeAuthority`.
- revert in `setAuthority` if `newAuthority` is equal to `address(0)`.
- be sure that **every contract** that inherits from `AuthNoOwner` is **always calling** `_initializeAuthority` to correctly initialize the authority system.

BadgerDAO: Not fixing:

- same logic as address(0) in constructor.
- renouncing is a desirable behavior to have.

Spearbit: Acknowledged.

5.3 Low Risk

5.3.1 `_openCdp`, `closeCdp` and `_adjustCdpInternal` should always call `claimStakingSplitFee` to ensure to have the internal accounting always up-to-date

Severity: Low Risk

Context: BorrowerOperations.sol#L268, BorrowerOperations.sol#L384, BorrowerOperations.sol#L465

Description: BadgerDAO should always be sure that before executing any macro-operation all the internal states, indexes and accounting variables are up-to-date and have performed all the internal accounting logic needed.

By directly calling `claimStakingSplitFee()`, calculating the TCR and executing `applyPendingRewards(cdpId)` BadgerDAO can ensure that all the internal accounting and indexes are always up-to-date.

Recommendation: BadgerDAO should remove the `_checkDeltaIndexAndClaimFee` function with the appropriate combination of functions needed for the macro operation.

Here's a pseudocode example of a modification that can be applied in `_adjustCdpInternal`

```
-uint _tcr = _checkDeltaIndexAndClaimFee(vars.price);
-bool isRecoveryMode = _checkRecoveryModeForTCR(_tcr);

+cdpManager.applyPendingRewards(_cdpId);
+uint256 _tcr = _getTCR(_price);
+bool isRecoveryMode = _checkRecoveryModeForTCR(_tcr);
```

Here's a pseudocode example of a modification that can be applied in `_openCdp`

```
-uint _tcr = _checkDeltaIndexAndClaimFee(vars.price);
-bool isRecoveryMode = _checkRecoveryModeForTCR(_tcr);

+ICdpManagerData(address(cdpManager)).claimStakingSplitFee();
+uint256 _tcr = _getTCR(_price);
+bool isRecoveryMode = _checkRecoveryModeForTCR(_tcr);
```

BadgerDAO: Fixed in PR 521.

Spearbit: Fixed.

5.3.2 CDP operations break if `stakingRewardSplit` is ever set to `0`

Severity: Low Risk

Context: CdpManager.sol#L862, CdpManager.sol#L800-L803, LiquityBase.sol#L149

Description: If `stakingRewardSplit` is set to `0` this function will thow a division by zero error.

The function `_computeDeltaIndexToTriggerRM()` takes `stakingRewardSplit` as an argument:

```
function checkIfDeltaIndexTriggerRM(uint _price) external view override returns (uint, bool) {
    //...
    uint _newIndex = collateral.getPooledEthByShares(DECIMAL_PRECISION);
    if (_newIndex > _oldIndex) {
        (uint _requiredDelta, uint _tcr) = _computeDeltaIndexToTriggerRM(
            _newIndex,
            _price,
            stakingRewardSplit
        );
    //...
```

```
function _computeDeltaIndexToTriggerRM(
    uint _currentIndex,
    uint _price,
    uint _stakingSplit
) internal view returns (uint, uint) {
    uint _tcr = _getTCR(_price);
    if (_tcr <= CCR) {
        return (0, _tcr);
    } else if (_tcr == LiquityMath.MAX_TCR) {
        return (type(uint256).max, _tcr); // system cold start
    } else {
        uint _splitIndex = (_currentIndex * MAX_REWARD_SPLIT) / _stakingSplit;
        return ((_splitIndex * (_tcr - CCR)) / _tcr, _tcr);
    }
}
```

Opening, closing, and adjusting CDPs all rely on this function. Setting `stakingRewardSplit` to `0` would break those functions when the TCR is greater than the CCR and less than the `MAX_TCR`.

Recommendation: Handle the case where `stakingRewardSplit` is equal to zero and return early without taking fees (but updating all the timestamps and so on).

BadgerDAO: The `BorrowerOperations._checkDeltaIndexAndClaimFee` and `CdpManager.checkIfDeltaIndexTriggerRM` functions have been removed in PR 521.

Spearbit: Fixed.

5.3.3 Liquidations are subject to race conditions and gas wars within blocks

Severity: Low Risk

Context: LiquidationLibrary.sol#L52, LiquidationLibrary.sol#L683

Description: CDP liquidations seem to be subject to race conditions and gas wars within blocks. All liquidation calculations are deterministic, collateral received will be the same for any liquidator, pushing liquidators to attempt to front run one another in the block's transaction order.

Additionally, a liquidator may front run with a partial liquidation of a CDP, causing another liquidator of the same CDP to receive less collateral than they were expecting (as well as paying less eBTC in total).

Liquidators will likely minimize this by using a flashbots relayer to keep their transactions out of the public mempool, but it is worth noting that these conditions seem to exist.

Recommendation: Add the following user provided arguments to liquidation functions:

- An expiry **time** (say Bob the liquidator doesn't want to risk changing market conditions, so limit a time window when the call can be executed).
- A minimum **amount** threshold (Bob has limited budget and doesn't want to be spread thin due to the execution costs and minimum amount requirements of other opportunities).
- A minimum liquidation **reward** percentage realized (say market experienced a sharp drop and due to ongoing collateral volatility Bob doesn't want to execute with `< 5%`, but is ok with `>= 5%` premium).

BadgerDAO: Acknowledged.

Spearbit: Acknowledged.

5.3.4 `CdpManager constructor` is missing input sanity checks

Severity: Low Risk

Context: BorrowerOperations.sol#L97-L106

Description: The following inputs of `BorrowerOperations constructor` should be properly validated to avoid initializing the contract to an invalid state.

- `_cdpManagerAddress` should be not equal to `address(0)`.
- `_collSurplusPoolAddress` should be not equal to `address(0)`.
- `_sortedCdpsAddress` should be not equal to `address(0)`.
- `_ebtcTokenAddress` should be not equal to `address(0)`.
- `_feeRecipientAddress` should be not equal to `address(0)`.
- `address(AuthNoOwner(_cdpManagerAddress).authority())` should be not equal to `address(0)`.

Recommendation: BadgerDAO should implement the sanity checks suggested above and revert if one of those values is equal to `address(0)`.

BadgerDAO: Acknowledged. Not going to use address(0) checks for immutables and will instead confirm system functionality and correctness post-deploy.

Spearbit: Acknowledged.

5.3.5 `ERC3156FlashLender.setMaxFeeBps` could lead the contract in an inconsistent state

Severity: Low Risk

Context: ERC3156FlashLender.sol#L27-L34

Description: The `setFeeBps` function requires that the `_newFee` is `<= maxFeeBps`. The `setMaxFeeBps` function requires that the new upper bound limit must be `<= MAX_BPS` but does not check if the current `feeBps` is consistent with the new upper bound.

Because of this, after executing `setMaxFeeBps` the contract could end up with a `feeBps` that is `> maxFeeBps` that is an inconsistent state for the checks that are applied in `setFeeBps`.

Recommendation: BadgerDAO should ensure in `setMaxFeeBps` that the current `feeBps` is also less or equal to the new upper bound `_newMaxFlashFee`.

BadgerDAO: The recommendations have been implemented in PR 513. The function setMaxFeeBps has been removed and now MAX_FEE_BPS is a constant value.

Spearbit: Fixed.

5.3.6 Setting feeRecipientAddress to one of the system contract addresses is possible, while it will disable eBTC flash loans

Severity: Low Risk

Context: BorrowerOperations.sol#L829-L836

Description: In order to keep flash loans operable `feeRecipientAddress` in addition to zero address cannot be `address(ebtcToken)`, `address(cdpManager)` or `address(this)` as eBTC token do not allow to transfer to all these addresses:

- EBTCToken.sol#L117-L118

```
function transfer(address recipient, uint256 amount) external override returns (bool) {
    _requireValidRecipient(recipient);
```

- EBTCToken.sol#L279-L288.

```
    function _requireValidRecipient(address _recipient) internal view {
        require(
>>          _recipient != address(0) && _recipient != address(this),
            "EBTC: Cannot transfer tokens directly to the EBTC token contract or the zero address"
        );
        require(
>>          _recipient != cdpManagerAddress && _recipient != borrowerOperationsAddress,
            "EBTC: Cannot transfer tokens directly to the CdpManager or BorrowerOps"
        );
    }
```

While BorrowerOperations only forbids zero for `feeRecipientAddress`:

- BorrowerOperations.sol#L829-L836.

```
    function setFeeRecipientAddress(address _feeRecipientAddress) external requiresAuth {
        require(
>>          _feeRecipientAddress != address(0),
            "BorrowerOperations: cannot set fee recipient to zero address"
        );
        feeRecipientAddress = _feeRecipientAddress;
        emit FeeRecipientAddressChanged(_feeRecipientAddress);
    }
```

Flash loans will be disabled if `feeRecipientAddress` be unable to receive eBTC:

- BorrowerOperations.sol#L839-L865

```
    function flashLoan(
        ...
    ) external override returns (bool) {
        ...

        // Issue EBTC
        ebtcToken.mint(address(receiver), amount);

        // Callback
        require(
            receiver.onFlashLoan(msg.sender, token, amount, fee, data) == FLASH_SUCCESS_VALUE,
            "BorrowerOperations: IERC3156: Callback failed"
        );

        // Gas: Repay from user balance, so we don't trigger a new SSTORE
        // Safe to use transferFrom and unchecked as it's a standard token
        // Also saves gas
        // Send both fee and amount to FEE_RECIPIENT, to burn allowance per EIP-3156
>>      ebtcToken.transferFrom(address(receiver), feeRecipientAddress, fee + amount);
```

Impact: In the case if `feeRecipientAddress` be set to any of these addresses eBTC flash loans will be disabled.

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

Recommendation: Consider repeating all the checks from `EBTCToken._requireValidRecipient()` in `BorrowerOperations.setFeeRecipientAddress()`, i.e. in addition to zero address control for `address(ebtcToken)`, `address(cdpManager)` and `address(this)`.

BadgerDAO: Acknowledged, function is gated by governance and disabling flashloans is possible by proper means so is possible behavior from normal system operation with same permissions.

Spearbit: Acknowledged.

5.3.7 ActivePool and BorrowerOperations flash loans cannot be disabled by setting the fee to 100%

Severity: Low Risk

Context: ActivePool.sol#L323-L329

Description: Setting the flash loan fee to be `100%` is intended to be a functionality disabling lever per protocol docs:

> Flash Loan Fee
>
> There are three main reasons why the DAO might decide to adjust Flash Loan fees.
>
> Firstly, to maintain competitiveness in the market, these fees can be increased or decreased.
>
> Secondly, and more importantly, they may be adjusted to prevent them from becoming an obstacle or disincentive for liquidators and arbitrageurs.
>
> Lastly, **fees can be set to 100% in order to completely disable the Flash Loan functionality as necessary**.

However, currently setting this blocking fee level will not act as functionality pausing, as flash loans will continue to be operable, it will just be required for the outsized fee to be returned:

- ActivePool.sol#L259-L282

```
    function flashLoan(
        IERC3156FlashBorrower receiver,
        address token,
        uint256 amount,
        bytes calldata data
    ) external override returns (bool) {
        require(amount > 0, "ActivePool: 0 Amount");
        require(amount <= maxFlashLoan(token), "ActivePool: Too much");
        // NOTE: Check for `token` is implicit in the requires above

>>      uint256 fee = (amount * feeBps) / MAX_BPS;
>>      uint256 amountWithFee = amount + fee;
        uint256 oldRate = collateral.getPooledEthByShares(1e18);

        collateral.transfer(address(receiver), amount);

        // Callback
        require(
            receiver.onFlashLoan(msg.sender, token, amount, fee, data) == FLASH_SUCCESS_VALUE,
            "ActivePool: IERC3156: Callback failed"
        );

        // Transfer of (principal + Fee) from flashloan receiver
>>      collateral.transferFrom(address(receiver), address(this), amountWithFee);
```

Similarly for eBTC flash loans in BorrowerOperations:

- BorrowerOperations.sol#L839-L865

```
    function flashLoan(
        IERC3156FlashBorrower receiver,
        address token,
        uint256 amount,
        bytes calldata data
    ) external override returns (bool) {
        require(amount > 0, "BorrowerOperations: 0 Amount");
        require(token == address(ebtcToken), "BorrowerOperations: EBTC Only");
        require(amount <= maxFlashLoan(token), "BorrowerOperations: Too much");
        // NOTE: Check for `eBTCToken` is implicit in the two requires above

>>      uint256 fee = (amount * feeBps) / MAX_BPS;

        // Issue EBTC
        ebtcToken.mint(address(receiver), amount);

        // Callback
        require(
            receiver.onFlashLoan(msg.sender, token, amount, fee, data) == FLASH_SUCCESS_VALUE,
            "BorrowerOperations: IERC3156: Callback failed"
        );

        // Gas: Repay from user balance, so we don't trigger a new SSTORE
        // Safe to use transferFrom and unchecked as it's a standard token
        // Also saves gas
        // Send both fee and amount to FEE_RECIPIENT, to burn allowance per EIP-3156
>>      ebtcToken.transferFrom(address(receiver), feeRecipientAddress, fee + amount);
```

But if there be a bug discovered in the logic, the increased fee might not stop its execution. It might be profitable enough to be able to return `100%` fee, as an example it might be able to severely manipulate `eBTC` price, so that `2x` in terms of initial amount will be much less than initial valuation of the loaned funds.

Impact: It will now be impossible to disable flash loans by setting fee to be 100% as it is intended.

Per very low likelihood and high impact from missing the emergency lever, setting the severity to be low.

Recommendation: Max fee special check can be added

- ActivePool.sol#L323-L329

```
    function maxFlashLoan(address token) public view override returns (uint256) {
        if (token != address(collateral)) {
            return 0;
        }
+       if (feeBps == MAX_BPS) {
+           return 0;
+       }

        return collateral.balanceOf(address(this));
    }
```

- BorrowerOperations.sol#L882-L887

```
    function maxFlashLoan(address token) public view override returns (uint256) {
        if (token != address(ebtcToken)) {
            return 0;
        }
+       if (feeBps == MAX_BPS) {
+           return 0;
+       }

        return type(uint112).max;
    }
```

BadgerDAO: Fixed via the new FL fee mechanic described in Badger-Finance/ebtc#513.

Flashloans can be paused via governance via a new boolean parameter

Spearbit: Fix looks ok, new logic needs to be described in the protocol documentation.

5.3.8 `CdpManager constructor` is missing input sanity checks

Severity: Low Risk

Context: CdpManager.sol#L42-L43

Description: The following inputs of `CdpManager constructor` should be properly validated to avoid initializing the contract to an invalid state.

- `_liquidationLibraryAddress` should be not equal to `address(0)`.
- `_authorityAddress` should be not equal to `address(0)`.

The other values should also be checked, but this operation should be delegated directly to `CdpManagerStorage` and `LiquityBase` (see other similar issues related to those Contracts).

Recommendation: BadgerDAO should implement the sanity checks suggested above.

BadgerDAO: My general stance on this is that address(0) is not the only possible invalid input. the entire system setup should be reviewed and fork-simulated against in prod via a test suite to confirm properties after deploy. Not in favor of adding address(0) checks for one-time things as it adds a bunch of lines of code for little benefit. Open to changing mind.

Spearbit: Acknowledged.

5.3.9 `HintHelpers.getApproxHint` could end up using an outdated CDP NICR

Severity: Low Risk

Context: HintHelpers.sol#L209-L242

Description: The `getApproxHint` function in `HintHelpers` is not taking in consideration that the CDP collateral/debt (and stake) is not up-to-date because

1) Splitting fees needs to be collected.
2) Splitting fees must be applied to the CDP.
3) Distributed debt must be applied to the CDP.

Recommendation: BadgerDAO should consider re-writing the logic of `HintHelpers` to calculate the CDP NICR with the updated values of the CDP collateral and debt after synching and fetching the Splitting Fees and applying to the CDP state both the splitting fees and the distributed debt.

Spearbit: Acknowledged.

5.3.10 Consider updating the CDP Reward Snapshot (debt redistributed) only if the user has indeed accumulated debt after rounding error

Severity: Low Risk

Context: CdpManagerStorage.sol#L255

Description: The `_updateCdpRewardSnapshots` function updates the CDSs reward snapshot index to the global `L_EBTCDebt` one. When `_applyPendingRewards` is executed, `BadgerDAO` could consider to call `_updateCdpRewardSnapshots` only if `pendingEBTCDebtReward` is not equal to zero.

`pendingEBTCDebtReward` is equal to zero (for an active CDP) if `rewardSnapshots[_cdpId] == L_EBTCDebt` or if `(stake * rewardPerUnitStaked) < DECIMAL_PRECISION`.

By updating the CDPs index even when `pendingEBTCDebtReward` it would mean that the CDPs distributed reward won't be paid by the CDPs owner and will remain in the protocol.

Recommendation: BadgerDAO should consider calling `_updateCdpRewardSnapshots` in `_applyPendingRewards` only if `pendingEBTCDebtReward > 0`.

There are also some recommended refactoring of the functions involved, first consider optimizing `_getRedistributedEBTCDebt()` to save gas for non-active CDPs and zero change cases:

- CdpManagerStorage.sol#L200-L217

```
    function _getRedistributedEBTCDebt(
        bytes32 _cdpId
    ) internal view returns (uint pendingEBTCDebtReward) {
-       uint snapshotEBTCDebt = rewardSnapshots[_cdpId];
        Cdp storage cdp = Cdps[_cdpId];

        if (cdp.status != Status.active) {
            return 0;
        }

-       uint stake = cdp.stake;

-       uint rewardPerUnitStaked = L_EBTCDebt - snapshotEBTCDebt;
+       uint rewardPerUnitStaked = L_EBTCDebt - rewardSnapshots[_cdpId];

        if (rewardPerUnitStaked > 0) {
-           pendingEBTCDebtReward = (stake * rewardPerUnitStaked) / DECIMAL_PRECISION;
+           pendingEBTCDebtReward = (cdp.stake * rewardPerUnitStaked) / DECIMAL_PRECISION;
        }
    }
```

And, based on the fact that `_hasRedistributedDebt()` returns false when cdp isn't active, the `_requireCdpIsActive()` call in `_applyPendingRewards()` can be removed:

- CdpManagerStorage.sol#L219-L227

```
function _hasRedistributedDebt(bytes32 _cdpId) internal view returns (bool) {
    ...
    if (Cdps[_cdpId].status != Status.active) {
        return false;
    }
```

- CdpManagerStorage.sol#L492-L494

```
function _requireCdpIsActive(bytes32 _cdpId) internal view {
    require(Cdps[_cdpId].status == Status.active, "CdpManager: Cdp does not exist or is closed");
}
```

Combining these points, as `_getRedistributedEBTCDebt()` will have a comparable cost with `_hasRedistributedDebt()` and similarly to `_hasRedistributedDebt() == false` it will be `pendingEBTCDebtReward == 0` if the CDP isn't active, there is no point in repeating the same logic, and the cumulative recommendation can be the following:

- CdpManagerStorage.sol#L239-L246

```
    function _applyPendingRewards(bytes32 _cdpId) internal {
        _applyAccumulatedFeeSplit(_cdpId);

+       // Compute pending rewards
+       uint pendingEBTCDebtReward = _getRedistributedEBTCDebt(_cdpId);
-       if (_hasRedistributedDebt(_cdpId)) {
+       if (pendingEBTCDebtReward > 0) {
-           _requireCdpIsActive(_cdpId);

-           // Compute pending rewards
-           uint pendingEBTCDebtReward = _getRedistributedEBTCDebt(_cdpId);
```

Spearbit: Acknowledged.

5.3.11 `CdpManagerStorage constructor` is missing input sanity checks

Severity: Low Risk

Context: CdpManagerStorage.sol#L130-L133

Description: All the inputs of `CdpManagerStorage constructor` should be properly validated to avoid initializing the contract to an invalid state.

- `_borrowerOperationsAddress` should be not equal to `address(0)`.
- `_collSurplusPool` should be not equal to `address(0)`.
- `_ebtcToken` should be not equal to `address(0)`.
- `_sortedCdps` should be not equal to `address(0)`.

Recommendation: BadgerDAO should implement the sanity checks suggested above.

BadgerDAO: Acknowledged, not going to use address(0) checks for immutables and will instead confirm system functionality and correctness post-deploy.

Spearbit: Acknowledged.

5.3.12 `LiquityBase constructor` is missing input sanity checks

Severity: Low Risk

Context: LiquityBase.sol#L52-L56

Description: All the inputs of `LiquityBase constructor` should be properly validated to avoid initializing the contract to an invalid state.

- `_activePoolAddress` should be not equal to `address(0)`
- `_priceFeedAddress` should be not equal to `address(0)`
- `_collateralAddress` should be not equal to `address(0)`

Recommendation: BadgerDAO should implement the sanity checks suggested above.

BadgerDAO: Acknowledged, not going to use address(0) checks for immutables and will instead confirm system functionality and correctness post-deploy.

Spearbit: Acknowledged.

5.3.13 `FeeRecipient constructor` is missing input sanity checks

Severity: Low Risk

Context: FeeRecipient.sol#L22-L25

Description: All the inputs of `FeeRecipient constructor` should be properly validated to avoid initializing the contract to an invalid state.

- `_ownerAddress` should be not equal to `address(0)` otherwise `safeTransfer` will revert.
- `_authorityAddress` should not be equal to `address(0)`. See also *'AuthNoOwner._initializeAuthority and setAuthority should revert if newAuthority is equal to address(0)'*. Without a proper authority configured, no one will be able to call the `sweepToken` function.

Recommendation: BadgerDAO should implement the sanity checks suggested above.

BadgerDAO: Acknowledged, not going to use address(0) checks here and will instead confirm system functionality and correctness post-deploy. Will document the behavior.

Spearbit: Acknowledged.

5.3.14 `CollSurplusPool constructor` is missing input sanity checks

Severity: Low Risk

Context: CollSurplusPool.sol#L44-L53

Description: All the inputs of `CollSurplusPool constructor` should be properly validated to avoid initializing the contract to an invalid state.

- `_borrowerOperationsAddress` should be not equal to `address(0)`.
- `_cdpManagerAddress` should be not equal to `address(0)`.
- `_activePoolAddress` should be not equal to `address(0)`.
- `_collTokenAddress` should be not equal to `address(0)`.
- `IActivePool(activePoolAddress).feeRecipientAddress()` should be not equal to `address(0)`. Without a proper `feeRecipientAddress` the `sweepToken` will revert when called.
- `_authorityAddress` should not be equal to `address(0)`. See also *'AuthNoOwner._initializeAuthority and setAuthority should revert if newAuthority is equal to address(0)'*. Without a proper authority configured, no one will be able to call the `sweepToken` function.

Recommendation: BadgerDAO should implement the sanity checks suggested above.

BadgerDAO: Acknowledged, not going to use address(0) checks and will instead confirm system functionality and correctness post-deploy. will document the incorrect setup behavior described.

Spearbit: Acknowledged.

5.3.15 `ActivePool.maxFlashLoan` is allowing to flashloan more than the total CDPs collateral

Severity: Low Risk

Context: ActivePool.sol#L328

Description: Quoting directly from the eBTC specification document shared

> Similarly, stETH can also be flash borrowed. In this case, the requested amount is taken from the collateral pool contract, transferred to the user, and then returned to the pool once the loan is repaid. The amount of collateral that can be flash borrowed is limited by the amount held in the pool.

This means that the `maxFlashLoan` function in `ActivePool` should return only the total amount of collateral that is held by all the CDPs. The current implementation of `ActivePool.maxFlashLoan` instead returns `collateral.balanceOf(address(this));` allowing the user to flash-loan the whole amount of `stETH` owned by the `ActivePool`.

This contradicts the Specification Document for two reasons:

1) `ActivePool` owns more than the total amount of `stETH` deposited by CDPs owner. Inside `ActivePool` we also have
    - `stETH` sent to the contract by mistakes
    - Liquidator Fees (not tracked by `StEthColl`)
    - `StEthColl` (collateral of users)
    - `FeeRecipientColl` (fees from redeem operation + fees from fee split when `stETH` increases in value)
2) `maxFlashLoan` is not calculating the correct amount of shares owned by the CDPs because is not checking and applying the splitting-fee mechanism.

Recommendation: BadgerDAO should allow users to only flash-loan an amount of `stETH` that is equal or less of the total amount of `stETH` owned by the CDPs. Before returning that amount `BadgerDAO` should trigger `CdpManager.claimStakingSplitFee()` to calculate the real amount of shares that are owned by the CDPs.

BadgerDAO: In trying to apply the fix we ran into the issue that during the Flashloan, the `maxFlashLoan` function returns an invalid value (the ratio).

In lack of any specific risk we believe it's ok to allow flashloaning the entire amount since the repayment check ensures that the:

- Repayment is done + fee.
- stETH did not rebase mid FlashLoan.

Spearbit: Acknowledged.

5.3.16 PriceFeed is not considering that the "Trigger parameters" of the Chainlink oracle could change

Severity: Low Risk

Context: PriceFeed.sol#L32-L33

Description: Usually, a new answer in the Chainlink oracle is written when one of the "Trigger parameters" of the oracle's configuration is matched.

- The deviation threshold of the prices is greater than `x%`.
- `X time` has passed since the last answer.

When one of these condition is met, a new answer is written and can be fetched by querying the Oracle.

The `PriceFeed` has two `constant` values `TIMEOUT_ETH_BTC_FEED` and `TIMEOUT_STETH_ETH_FEED` that are used to understand whether a Chainlink answer is outdated or not (frozen).

If the Chainlink "Trigger parameters" configuration changes, those `constant` values should be updated to be relevant. Without updating them, we could end up in the following scenarios:

- A valid Chainlink answer is discarded because it has been considered "frozen" (incorrectly).
- A frozen Chainlink answer is accepted (incorrectly) because the timeout constant value was too relaxed.

Recommendation: BadgerDAO should consider not making those values as constant and create proper setters in the case they need to be updated when Chainlink updates the Oracle configuration.

Spearbit: Acknowledged.

5.4 Gas Optimization

5.4.1 Redundant check in `_openCdp()` can be removed

Severity: Gas Optimization

Context: BorrowerOperations.sol#L371-L379

Description: The require statement at the start of the function and `_requireAtLeastMinNetColl()` are redundant checks. Solidity will throw an error on `vars.netColl = _getNetColl(_collAmount)` if `_collAmount` is less than `LIQUIDATOR_REWARD` as the math is checked. Using only `_requireAtLeastMinNetColl` should ensure that the collateral for the CDP is greater than zero.

Recommendation: Remove the require statement at the start of the function, leave a note and document that `_requireAtLeastMinNetColl` and solidity's checked math ensures that the function will revert if `_collAmount` is less than `MIN_NET_COLL + LIQUIDATOR_REWARD`.

Spearbit: Acknowledged.

5.4.2 `tellorQueryBufferSeconds` could be defined as constant

Severity: Gas Optimization

Context: TellorCaller#26

Description: The comment over that variable explains that soft governance might help to change this default configuration. However, there is no way to change that variable with current functionalities.

Recommendation: The `tellorQueryBufferSeconds` variable could be defined as constant if it is designed not to be changed. Therefore, it will consume less gas during the deployment. As another suggestion, make it dynamic by adding a setter and allow the governance to change the value without the need to update the fallback oracle on `PriceFeed`.

Spearbit: Acknowledged.

5.4.3 `SLOAD` can be avoided by using `msg.sender` on `_transferOwnership`

Severity: Gas Optimization

Context: Ownable.sol#L79-L81

Description: The onlyOwner modifier ensures both public functions (renounceOwnership and transferOwnership) are called by msg.sender. address oldOwner could be assigned to msg.sender to save on an SLOAD op. Although, on construction this will result in the OwnershipTransferred event reading as from msg.sender to msg.sender.

Recommendation: Avoid loading from storage using `_owner` variable and pass `msg.sender` to the event.

BadgerDAO: Acknowledged. We'll maintain the canonical implementation as isn't isn't a user-facing or common operation.

Spearbit: Acknowledged.

5.5 Informational

5.5.1 `BorrowerOperations.maxFlashLoan` is not returning the correct max flash-loanable amount

Severity: Informational

Context: BorrowerOperations.sol#L882-L887

Description: By following the EIP-3156 specification, the `maxFlashLoan` function should return "The amount of currency available to be lent."

In the specific case of `BorrowerOperations.maxFlashLoan` it should return the max amount of `eBTC` that the `msg.sender` can flash-mint at time `T`. The current implementation of `maxFlashLoan` just returns the constant value of `type(uint112).max` without checking the `totalSupply()` of `eBTC`.

If, for example, `_totalSupply() > type(uint256).max - type(uint112).max` and the user tries to flashloan the max flashloanable amount of `eBTC` supported by `BorrowerOperations` the operation will revert to an overflow error because the `totalSupply` would be above `type(uint256).max`.

Recommendation: BadgerDAO should consider to re-rewrite the `maxFlashLoan` function, considering the `eBTC` `totalSupply` in the calculation of the maximum amount that a user can flash-loan at time `T`.

BadgerDAO: Acknowledged, at this time we don't see a specific reason for a specific cap. We also don't believe a overflow is realistic since all ETH in the world is less than 2^128.

Spearbit: Acknowledged.

5.5.2 Assert is used in BorrowerOperations, CdpManager, CdpManagerStorage and EBTCToken

Severity: Informational

Context: BorrowerOperations.sol#L279 CdpManager.sol#L345 CdpManagerStorage.sol#L162 EBTCToken.sol#L230-L232

Description: There is a number of instances where `assert` is used to check system invariants.

4 appearencies in BorrowerOperations, starting with:

- BorrowerOperations.sol#L279

```
assert(msg.sender == _borrower);
```

3 in CdpManager, starting with:

- CdpManager.sol#L345

```
assert(ebtcToken.balanceOf(msg.sender) <= totals.totalEBTCSupplyAtStart);
```

4 in CdpManagerStorage, starting with:

- CdpManagerStorage.sol#L162

```
assert(closedStatus != Status.nonExistent && closedStatus != Status.active);
```

6, all being zero address checks, in EBTCToken, starting with:

- EBTCToken.sol#L230-L232

```
function _transfer(address sender, address recipient, uint256 amount) internal {
    assert(sender != address(0));
    assert(recipient != address(0));
```

While it is suitable for the development phase, using the operation after that isn't recommended both from gas costs and system transparency points of view.

Impact: Assert will consume all the available gas and gives away no information for troubleshooting, failing to provide any additional benefits when being used instead of require in production.

Setting gas severity for gas costs and informational for system transparency component.

Recommendation: Consider replacing all `assert` instances with `require`, which both returns gas and allows for error message.

Also, it is recommended to place this checks as early as possible, for example this one can be moved upwards:

- BorrowerOperations.sol#L303

```
assert(_collWithdrawal <= _cdpCollAmt);
```

In some cases this can be paired with optimizations, as an example this removes assert and duplicate `balanceOf()` call, groups similar logic (there are no other usages of `_requireEBTCBalanceCoversRedemption()`):

- CdpManager.sol#L341-L345

```
+       totals.totalEBTCSupplyAtStart = _getEntireSystemDebt();
-       _requireEBTCBalanceCoversRedemption(ebtcToken, msg.sender, _EBTCamount);
+       _requireEBTCBalanceCoversRedemptionAndWithinSupply(ebtcToken, msg.sender, _EBTCamount,
↪   totals.totalEBTCSupplyAtStart);

-       totals.totalEBTCSupplyAtStart = _getEntireSystemDebt();
-       // Confirm redeemer's balance is less than total EBTC supply
-       assert(ebtcToken.balanceOf(msg.sender) <= totals.totalEBTCSupplyAtStart);
```

- CdpManager.sol#L820-L829

```
-   function _requireEBTCBalanceCoversRedemption(
+   function _requireEBTCBalanceCoversRedemptionAndWithinSupply(
        IEBTCToken _ebtcToken,
        address _redeemer,
        uint _amount,
+       uint _totalSupply
    ) internal view {
+       uint callerBalance = _ebtcToken.balanceOf(_redeemer);
        require(
-           _ebtcToken.balanceOf(_redeemer) >= _amount,
+           callerBalance >= _amount,
            "CdpManager: Requested redemption amount must be <= user's EBTC token balance"
        );
+       require(
+           callerBalance <= _totalSupply,
+           "CdpManager: user's EBTC balance exceeds total supply"
        );
    }
```

BadgerDAO: Fixed in PR 513.

Spearbit: Fix looks ok, `assert` instances are now replaced, and suggested optimizations are implemented.

5.5.3 The requirement made in `ActivePool` are not actually verifying that the flash-loaner has repaid the `amount+fee`

Severity: Informational

Context: ActivePool.sol#L293-L297

Description: Inside `ActivePool` does not only own the collateral sent by the users that have opened a CDP, in the current implementations it owns

- `stETH` sent to the contract by mistakes.
- Liquidator Fees (not tracked by `StEthColl`).
- `StEthColl` (collateral of users).
- `FeeRecipientColl` (fees from redeem operation + fees from fee split when `stETH` increases in value).

As a consequence, both `collateral.balanceOf(address(this))` and `collateral.sharesOf(address(this))` could already be greater than `collateral.getPooledEthByShares(StEthColl)` and `StEthColl` respectively. Because of this, those requirements are not enough to verity that the `msg.sender` has indeed repaid the flashloaned `amount + fees`.

Recommendation: BadgerDAO should first of all allow users to only borrow the actual collateral deposited in the protocol.See *'ActivePool.maxFlashLoan is allowing to flashloan more than the total CDPs collateral'*.

BadgerDAO could also think to only "store" in the `ActivePool` the collateral owned by the CDPs and move away to other contracts the `Liquidator Fees` and `FeeRecipientColl` amounts to make the calculations and checks much more clear and easy to maintain.

BadgerDAO: Per our discussions we believe that the check above: ActivePool.sol#L282-L285 ensures that full payment is made.

The checks below are not as effective as we thought, and more specifically the only useful check is: ActivePool.sol#L298-L301 which ensures that stETH didn't rebase mid Flashloan (causing losses)

We have considered removing the two extra checks but don't see any harm in keeping them at this time.

Spearbit: Acknowledged.

5.5.4 `burnCapability()` function can be redesigned

Severity: Informational

Context: RolesAuthority.sol#L150-L151

Description: If `enabledFunctionSigsByTarget[target]` variable only contains *burned* capabilities, then, `target` parameter should be removed from the targets set.

Recommendation: Consider removing the `target` parameter from `targets` variable at the end of the `burnCapability()` call.

BadgerDAO: Acknowledged as we do not consider the optimization worth changing the code.

Spearbit: Acknowledged.

5.5.5 All the operations should properly apply a set of pre-post system and function invariants

Severity: Informational

Context: Whole project

Description: Following the concept introduced by Nascent FREI-PI blog post, BadgerDAO should identify a set of invariants that should always be held true system wise and function wise.

On top of those invariants that must be applied before and after each operation, BadgerDAO should define a set of operations that should be performed before each function's logic like for example

- Calling `claimStakingSplitFee()` to fetch "staking fees" if possible.
- Calling `applyPendingRewards(cdpId)` to be sure that the CDP collateral, debt, stake and ICR is up-to-date.
- Calculating the TCR, ICR, NICR and RM only after the staking fee has been calculated and applied to the CDP.
- Other operation needed before.

The same logic should be applied to all the clean-up operations that should be applied after the main operation flow.

All these invariants, pre-post main-operation checks, pre-post main-operation sub operations should be well documented and applied for every function of the codebase.

Recommendation: BadgerDAO should follow the suggestions listed in the section above.

BadgerDAO: Acknowledged.

Spearbit: Acknowledged.

5.5.6 Local variables and functions are defined and never used

Severity: Informational

Context: LiquidationLibrary.sol#L323, LiquidationLibrary.sol#L980, LiquityBase.sol#L65-L67, CdpManager.sol#L718, CdpManager.sol#L730

Description: Local variables and functions are defined and never used.

Recommendation: Pay attention to solidity compiler warnings and remove unused variables.

BadgerDAO: Fixed in PR 519.

Spearbit: Fixed.

5.5.7 Move requirement from internal function to external `insert()` function for readability

Severity: Informational

Context: SortedCdps.sol#L175

Description: The require statement that ensures the caller is authorized to perform the insertion is located within the internal function. Readability would be improved if this requirement was moved to the external function where it is actually called from.

Recommendation: Move the requirement to the external function `insert()`.

Spearbit: Acknowledged.

5.5.8 Error messages do not conform to established patterns

Severity: Informational

Context: SortedCdps.sol#L128, LiquidationLibrary.sol#L68, LiquidationLibrary.sol#L473, CdpManagerStorage.sol#L391, CdpManager.sol#L472, CdpManager.sol#L476, CdpManager.sol#L850, BorrowerOperations.sol#L125-L126, BorrowerOperations.sol#L607, BorrowerOperations.sol#L832, BorrowerOperations.sol#L845-L847, BorrowerOperations.sol#L858, BorrowerOperations.sol#L876,

Description: Error messages to not conform to the patterns established by the other error messages in their respective contracts.

Recommendation: Add or edit error messages to conform to the pattern established in each contract.

Spearbit: Acknowledged.

5.5.9 Consider replacing `2 ** 256 - 1` with `type(uint256).max`

Severity: Informational

Context: LiquityMath.sol#L7

Description: `type(uint256).max` could be used for max uint256 value instead of `2 ** 256 - 1`.

Recommendation: Consider using type(uint256).max for readability.

BadgerDAO: Fixed in PR 513.

Spearbit: Fixed.

5.5.10 `LiquidationLibrary` and `CdpManager` have many duplicate functions that could be shared in a common contract

Severity: Informational

Context: LiquidationLibrary.sol#L934-L939, LiquidationLibrary.sol#L943-L949, LiquidationLibrary.sol#L954-L958, LiquidationLibrary.sol#L960-L962, LiquidationLibrary.sol#L965-L972, LiquidationLibrary.sol#L976-L991

Description: The following functions are implemented in both the `LiquidationLibrary` and `CdpManager`. The code of those function is identical between the contracts. To avoid possible confusion, code duplication and bugs, those functions should be extracted and placed in a common contract that will be inherited by both the `LiquidationLibrary` and `CdpManager` contracts.

Recommendation: BadgerDAO should consider extracting those functions, place them in a separate contract and make `LiquidationLibrary` and `CdpManager` inherit from that common contract.

BadgerDAO: The recommendations have been implemented in PR 513.

Spearbit: Fixed.

5.5.11 Contract files are unused

Severity: Informational

Context: Dependencies/IBalancerV2Vault.sol, CheckContract.sol

Description: These contracts are unused within the codebase.

Recommendation: Remove the unused contracts.

BadgerDAO: it is still used in test contracts, but is not necessary. removing in PR 513.

Spearbit: The recommendations have been implemented.

5.5.12 Assembly block can be replaced for simplicity

Severity: Informational

Context: EBTCToken.sol#L214-L216

Description: Assembly block can be reduced to a single line for readability and simplicity.

Recommendation: Replace with `block.chainid`.

Spearbit: Acknowledged.

5.5.13 Magic numbers can be replaced by defined constant variables

Severity: Informational

Context: BorrowerOperations.sol#L326, CdpManager.sol#L67, ActivePool.sol#L271, ActivePool.sol#L299, HintHelpers.sol#L148, LiquidationLibrary.sol#L370, Dependencies/LiquityMath.sol#L99,

Description: Magic numbers such as `1e18` are referenced throughout the code base and can be replaced by defined constant variables such as `DECIMAL_PRECISION`.

Recommendation: Replace magic numbers with defined constants.

BadgerDAO: Recommendations implemented in the PR 513.

Spearbit: Fixed.

5.5.14 Comment inaccuracies and typos

Severity: Informational

Context: CdpManager.sol#L135, CdpManager.sol#L166, CdpManager.sol#L264, CdpManager.sol#L621, CdpManager.sol#L660, CdpManager.sol#L786, CdpManagerStorage.sol#L280, BorrowerOperations.sol#L139, BorrowerOperations.sol#L162, BorrowerOperations.sol#L175, BorrowerOperations.sol#L219, BorrowerOperations.sol#L267, BorrowerOperations.sol#L383, BorrowerOperations.sol#L401-L406, BorrowerOperations.sol#L670, BorrowerOperations.sol#L674, BorrowerOperations.sol#L881, BorrowerOperations.sol#L889, SortedCdps.sol#L38, PriceFeed.sol#L785, PriceFeed.sol#L787, Dependencies/LiquityBase.sol#L39 Dependencies/LiquityBase.sol#L115-L122, Dependencies/LiquityBase.sol#L128-L130,

Description: There are several inaccuracies and typos in the comments within the codebase.

Recommendation: Correct, update, or remove the inaccuracies and typos.

Spearbit: Acknowledged.

5.5.15 `CdpManager.redeemCollateral` could revert when called during an eBTC Token flash-loan operation

Severity: Informational

Context: CdpManager.sol#L344-L345

Description: The `BorrowerOperations` contract allows any user to flash-mint `type(uint112).max eBTC`. This amount of flash-minted `eBTC` could be greater than the amount of the total `eBTC` borrowed by all the CDPs.

If that's the case, the `redeemCollateral` function will revert when the `assert(ebtcToken.balanceOf(msg.sender) <= totals.totalEBTCSupplyAtStart)` check is executed, preventing a user from using the flash-minted `eBTC` to be used during a redeem operation.

Recommendation: The `assert(ebtcToken.balanceOf(msg.sender) <= totals.totalEBTCSupplyAtStart);` check is still a valid check with a purpose. BadgerDAO should carefully evaluate what are the consequences of leaving or removing the check to allow or not the usage of a flash-minted quantity of `eBTC`, that is greater than the whole borrowed amount, to be used in the redeem process.

BadgerDAO: Agree with Ack, if you redeem 100% of the eBTC the fee will reach 100% so it's not a rational thing to do.

Spearbit: Acknowledged.

5.5.16 `CdpManagerStorage._closeCdpWithoutRemovingSortedCdps` should also reset `stFeePerUnitcdp` mapping when a CDP is closed

Severity: Informational

Context: CdpManagerStorage.sol#L161-L175

Description: The role of `_closeCdpWithoutRemovingSortedCdps` is to clean up all the information relative to the CDPs with ID `_cdpId`. The function is correctly updating the CDP status and resetting all the other properties of the struct. In addition to that is also cleaning the debt distribution index `rewardSnapshots[_cdpId]`.

To completely clean all the information relative to the CDP, the function should also reset the value relative to `stFeePerUnitcdp[_cdpId]`.

Recommendation: Consider to also cleaning the `stFeePerUnitcdp[_cdpId]` value during the closure of the CDP.

BadgerDAO: The recommendations have been implemented in the PR 513.

Spearbit: Fixed.

5.5.17 Misc improvements / suggestions

Severity: Informational

Context: CdpManagerStorage.sol#L462-L463, FeeRecipient.sol#L17

Description:

- `_stFeePerUnitgError` and `_totalStakes` function parameters from `CdpManagerStorage.getAccumulatedFeeSplitApplied` can be removed because they are never used.
- `FeeRecipient SweepTokenSuccess` event could declare both `_token` and `_recipient` as 'indexed.
- `INDEX_UPD_INTERVAL`, `CollateralIndexUpdateIntervalUpdated` and `syncUpdateIndexInterval` can be removed from the codebase.
- `BORROWING_FEE_FLOOR` state variable, `getBorrowingRate()`, `getBorrowingRateWithDecay()`, `_calcBorrowingRate()`, `getBorrowingFee()`, `getBorrowingFeeWithDecay()`, `_calcBorrowingFee()` and `decayBaseRateFromBorrowing()` can all be removed from `CdpManager` because `eBTC` does not support borrowing fees.
- `_liquidatorReward` input parameter in `LiquidationLibrary._calculateSurplusAndCap` is never used and can be removed.
- `LiquidationTotals.totalCollToRedistribute` can be removed from the `LiquidationTotals` struct and the whole codebase.
- `_liquidatorRewardShares` can be removed from `BorrowerOperations. _activePoolAddColl` because is never used
- `BorrowerOperations._getUSDValue` is never used and can be removed.
- `LiquityBase. _getPriceReciprocal` and `LiquityBase. _convertDebtDenominationToEth` are never used and can be removed.
- `LiquityBase` constants `_100pct`, `_105pct` and `_5pct` are never used and can be removed.

Recommendation: BadgerDAO should consider following the above suggestions.

BadgerDAO: Resolved in PR 513.

Spearbit: The recommendations have not been implemented yet. Are those recommendations implemented in another PR, should they be included in this PR, or do you plan not to implement them at all? Marking as Acknowledged.

5.5.18 `AuthNoOwner` state variables should be set as `private`

Severity: Informational

Context: AuthNoOwner.sol#L13-L14

Description: Both the `authority` and `authorityInitialized` are declared as `public` variables. Because of this, any contract that inherits from `AuthNoOwner` could freely update those variables without passing through the internal logic of `AuthNoOwner` skipping the checks that are performed by `setAuthority` and `_initializeAuthority`.

Recommendation: BadgerDAO should consider setting `authority` and `authorityInitialized` as `private` (see OpenZeppelin `Ownable` as a reference example). The variable value should be updatable only by the internal logic of the `AuthNoOwner` contract If BadgerDAO decides to follow this recommendation, they should also implement the respective `getter` function to expose the value to external contracts/dApps.

BadgerDAO: The recommendations have been implemented in the PR 526.

Spearbit: Fixed.

5.5.19 `flashLoan` should use `flashFee` to calculate the amount of fees to be repaid

Severity: Informational

Context: ActivePool.sol#L269, BorrowerOperations.sol#L850

Description: To completely adhere to the EIP-3156 standard, the `flashLoan` function should enforce that the fee paid to flashloan an `amount` is equal to the amount returned by `flashFee`.

Currently, both `ActivePool` and `BorrowerOperations` are re-calculating manually the fees in the `flashLoan` function.

Recommendation: BadgerDAO should use `flashFee` to calculate the amount of fees that the user must repay.

BadgerDAO: The recommendations have been implemented in the PR 545.

Spearbit: Fixed.

5.5.20 BadgerDAO should document in an extensive way the default values chosen for `TellorCaller.tellorQueryBufferSeconds` and `TellorCaller.timeOut`

Severity: Informational

Context: TellorCaller.sol#L31, TellorCaller.sol#L26

Description: Tellor is a decentralized Oracle protocol that allows anyone (that has staked some `TRB` tokens) to submit a price update. The price is immediately added to the history of the oracle and will be returned as the latest valid price. The price can be disputed by anyone, and if the dispute is accepted, it will be removed.

Because anyone is allowed to submit a new price and because that new price is immediately added as the latest price, to prevent the usage of a fake or incorrect price, Tellor allows the integrator to specify a timestamp from which you want to get the first price available before that timestamp.

`tellorQueryBufferSeconds` defines how many seconds must have at least passed before fetching the price. This means that the last price could even been older than `tellorQueryBufferSeconds` seconds.

The `timeOut` parameters instead define the number of seconds that will be used by the `PriceFeed` to decide if an answer is too old and the Fallback Oracle must be considered "frozen" or not.

Both `tellorQueryBufferSeconds` and `timeOut` are crucially important to define the minimum and maximum "staleness" that an answer can be and should be documented and carefully validated by BadgerDAO.

Recommendation: BadgerDAO should carefully and deeply document why it has chosen the current value used for both `tellorQueryBufferSeconds` and `timeOut`.

BadgerDAO: At the time of the engagement we had agreed not to use the Tellor Oracle, we made a mistake in not removing it from the Repo and have no plans on using it.

Spearbit: The BadgerDAO team still needs to detail which service will replace `Tellor` or if the fallback oracle solution will be removed and only Chainlink will be used as the primary and only price feed.

5.5.21 BadgerDAO should consider documenting the constant value used by `PriceFeed` to determine whether an answer can be trusted or not

Severity: Informational

Context: PriceFeed.sol#L32, PriceFeed.sol#L33, PriceFeed.sol#L36, PriceFeed.sol#L42

Description: The `PriceFeed` use the following `constant` values to understand if an answer from ChainLink or the Fallback Oracle can be trusted or not

- `uint256 public constant TIMEOUT_ETH_BTC_FEED = 4800.`
- `uint256 public constant TIMEOUT_STETH_ETH_FEED = 90000.`
- `uint256 public constant MAX_PRICE_DEVIATION_FROM_PREVIOUS_ROUND = 5e17.`
- `uint256 public constant MAX_PRICE_DIFFERENCE_BETWEEN_ORACLES = 5e16.`

BadgerDAO should consider disclosing and documenting why those very specific values have been picked, and why they are enough to validate the goodness of an oracle's answer (also in relation with the Chainlink ETH/BTC and Chainlink stETH/ETH Trigger parameters).

Recommendation: BadgerDAO should consider disclosing and documenting why those very specific values have been picked, and why they are enough to validate the goodness of an oracle's answer.

Spearbit: Acknowledged.

5.5.22 Contracts, interfaces and libraries are lacking natspec documentation

Severity: Informational

Context: Multiple files across the whole codebase

Description: Multiple contracts, interfaces and libraries in the codebase are totally lacking the support of the Natspec documentation.

By supporting the Natspec documentation, the project will inherit multiple benefits like:

- Clear understanding of a contract/function for both developers and auditors.
- Automatic generation of useful descriptions by external tools like Etherscan.

Recommendation: BadgerDAO should consider adding the Natspec documentation to all the contracts, interfaces and libraries to improve the cleanliness of the project and a better DX for both developers and auditors.

Spearbit: Acknowledged.

5.5.23 Consider replacing all the `uint` instances with `uint256`

Severity: Informational

Context: Multiple files across the whole codebase

Description: In Solidity, `uint` is an alias for `uint256` but it's a common best practice to always use the more explicit type`uint256` instead of `uint`.

The current codebase uses a mix of both creating confusion and making it less readable.

Recommendation: BadgerDAO should consider replacing all the `uint` instances with `uint256`.

BadgerDAO: Will mitigate as a final PR due to the visual messiness.

Spearbit: Acknowledged.

5.5.24 Remove unused solidity imports and unused files

Severity: Informational

Context: Multiple files in the codebase

Description: Multiple solidity contracts and interfaces are currently importing files (other contracts or interfaces) that are not used by the contract/interface itself.

Just to make an example, by looking at the `ICdpManager.sol` interface we can see that `IEBTCToken.sol`, `IFeeRecipient.sol` and `ICollSurplusPool.sol` are imported but never used by the interface itself.

BadgerDAO should also consider removing all those files that are not currently used by the codebase. For example, files like `SafeMath.sol` and `LiquitySafeMath128.sol` (the contract is compiled with `solc > 0.8.x`)

Recommendation: Consider reviewing each solidity contract and interface and removing all the unused imports to clean up the code.

BadgerDAO: Fixed. We will not fix SafeMath as it's part of some tests.

Spearbit: Acknowledged. There are a lot more files that contains this problem. Just to make another example, `BorrowerOperations.sol` imports `Ownable.sol` that is never used by the contract itself.

5.5.25 Consider renaming variables related to `stETH` collateral and `stETH` shares to make the code more clear

Severity: Informational

Context: Suggestion that is applied to multiple Contracts and multiple variables

Description: The collateral used by the `eBTC` project to allow the user to borrow `eBTC` is `stETH`. `stETH` is a rebasing token and as suggested by Lido documentation for integrators, integrators should book keep the amount of shares and not the amount of ETH.

In the `eBTC` codebase, there are multiple places where the `stETH` is converted to shares and shares are converted back to `stETH`. Not all the codebase correctly name the variables that store `stETH` or `stETH Shares` with a clear name, and both developers and auditors need to always go back to the source of the variable to understand if it's storing pure `stETH` or the share amount.

Not having a clear standard/nomenclature for those variables lead to more confusions and could lead to errors because the developer could end up by mixing shares with `stETH`.

Because of the confusion, the audit and development process will slow down and take more time because you always need to understand what is stored in the variable and be sure to not mix up share with `stETH`.

Recommendation: Consider renaming all the variables that refer to the collateral to explicitly explain if the variable is storing `stETH` or `stETH share` amounts.

Spearbit: Acknowledged.