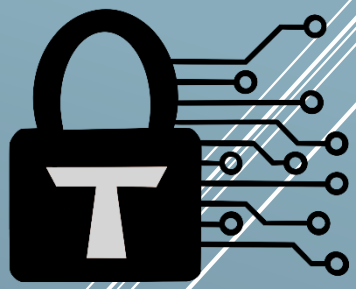
Trust Security

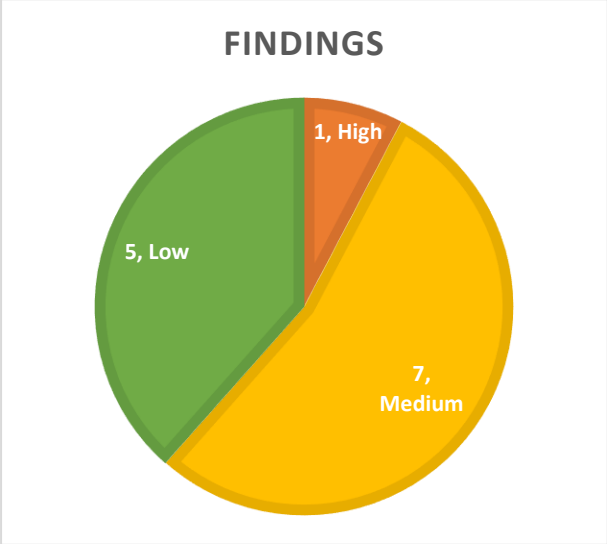


Smart Contract Audit

BadgerDAO eBTC

20/09/2023

Executive summary



Category	Stablecoin
Audited file count	35
Lines of Code	5211
Auditor	Bernd Artmüller, Lambda
Time period	15/05-09/06

Findings

Severity	Total	Fixed	Acknowledged
High	1	1	-
Medium	7	6	1
Low	5	1	4

Centralization score

Signature

Digitally signed by Trust
Date: 2023-09-21 11:19:31

Document properties

Versioning

Version	Date	Description
0.1	09/06/2023	Client report
0.2	19/09/2023	Public release
0.3	20/09/2023	Team response

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

- ./LiquidationLibrary.sol
- ./BorrowerOperations.sol
- ./CdpManager.sol
- ./PriceFeed.sol
- ./SortedCdps.sol
- ./LeverageMacroBase.sol
- ./CdpManagerStorage.sol
- ./EBTCToken.sol
- ./ActivePool.sol
- ./HintHelpers.sol
- ./Governor.sol
- ./SimplifiedDiamondLike.sol
- ./MultiCdpGetter.sol
- ./CollSurplusPool.sol
- ./EBTCDeployer.sol
- ./LeverageMacroFactory.sol
- ./LeverageMacroReference.sol
- ./LeverageMacroDelegateTarget.sol
- ./FeeRecipient.sol
- ./Migrations.sol
- ./Interfaces/ISortedCdps.sol
- ./Interfaces/ICdpManagerData.sol
- ./Interfaces/ICdpManager.sol
- ./Interfaces/IPriceFeed.sol
- ./Interfaces/IActivePool.sol
- ./Interfaces/IBorrowerOperations.sol
- ./Interfaces/ICollSurplusPool.sol
- ./Interfaces/IERC3156FlashLender.sol
- ./Interfaces/IFeeRecipient.sol
- ./Interfaces/IEBTCToken.sol
- ./Interfaces/IPool.sol
- ./Interfaces/IERC3156FlashBorrower.sol
- ./Interfaces/IFallbackCaller.sol
- ./Interfaces/IWETH.sol

- ./Interfaces/ILiquityBase.sol

Repository details

- **Repository URL:** https://github.com/Badger-Finance/ebtc
- **Commit hash:** 181cb500190571798e32da053939d5fb1e5565f1

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Trust is the leading auditor at competitive auditing service Code4rena, reported several critical issues to Immunefi bug bounty platform and is currently a Code4rena judge.

About the auditors

Lambda is a Security Researcher and Developer with multiple years of experience in IT security and traditional finance. This experience combined with his academic background in Data Science, Mathematical Finance, and High-Performance Computing enables him to thoroughly examine even the most complicated code bases, resulting in several top placements in various audit contests.

Bernd is a blockchain and smart contract security researcher transitioning from a successful full-stack web developer career. His ability to quickly grasp new concepts and technologies and his attention to detail have helped him become a top auditor in the blockchain space. Having conducted 40+ audits, Bernd has identified numerous vulnerabilities across a wide range of DeFi protocols, wallets, bridges, and VMs.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	**Good**	Project kept code as simple as possible, reducing attack risks
Documentation	**Excellent**	An extensive documentation was provided for the audit. Furthermore, many security-relevant assumptions are documented within the codebase.
Best practices	**Excellent**	The project follows industry standards whenever possible.
Centralization risks	**Good**	The system is non-upgradeable and the privileges of the owner are limited.

Findings

High severity findings

TRST-H-1 Oracle results are combined incorrectly, resulting in a wrong stETH/BTC price

- **Category:** Mathematical error
- **Source:** PriceFeed.sol
- **Status:** Fixed

Description

The function *_formatClAggregateAnswer()* uses the following formula to get the stETH/BTC price based on the ETH/BTC and stETH/ETH prices:

$$\text{stETH/BTC} = \frac{10^{\text{dec(stETH/ETH)}-\text{dec(ETH/BTC)}} * \text{ETH/BTC} * 10^{18}}{\text{stETH/ETH}}$$

However, this is wrong. Consider the case when the ETH/BTC rate is 0.06803827 and the stETH/ETH 0.99. We have ETH/BTC=6803827 and stETH/ETH=990000000000000000 (because of the decimals). Plugging this in the formula above results in stETH/BTC=68725525252525252, i.e. a stETH/BTC rate of 0.068725525. Therefore, stETH/BTC > ETH/BTC (i.e., you get more BTC for on stETH), which is wrong when stETH/ETH < 1 (i.e., you get less than 1 ETH for one stETH).

While the difference is relatively small when stETH/ETH is around 1, it can get very large for stETH/ETH << 1 (i.e., depeg events), resulting in completely wrong collateralization ratios.

Recommended mitigation

Correct the formula:

$$\text{stETH/BTC} = \frac{\text{ETH/BTC} * \text{stETH/ETH} * 10^{\text{dec(stETH/ETH)}-\text{dec(ETH/BTC)}}}{10^{18}}$$

Team Response

Fixed here.

Medium severity findings

TRST-M-1 Wrong oracle recovery logic when no fallback is present

- **Category:** Logical errors
- **Source:** PriceFeed.sol
- **Status:** Fixed

Description

The PriceFeed contract supports the deactivation of the fallback oracle by passing **address(0)** to *setFallbackCaller()*. While the contract works fine afterwards as long as the Chainlink oracle works properly, it can never recover from a Chainlink failure (i.e., **Status.bothOraclesUntrusted**). In such a scenario, a stale price will always be used until a new, valid fallback oracle is set.

Moreover, if the Chainlink oracle is nonfunctioning and the PriceFeed state remains in **Status.bothOraclesUntrusted**, adding a fallback oracle does not change the state and the returned price remains stale.

Recommended mitigation

When there is no fallback oracle, the recovery logic should be different to avoid scenarios where the prices are not updated again. The best that can be done in such a situation is to use the chainlink price again as soon as it is not broken anymore.

Additionally, adding a fallback oracle should transition the state accordingly to prevent a stale price.

Team Response

Fixed here.

TRST-M-2 The function maxFlashLoan() returns wrong values

- **Category:** Logical errors
- **Source:** BorrowerOperations.sol
- **Status:** Acknowledged

Description

The function *maxFlashLoan()* returns **type(uint112).max**, but there currently is no maximum enforced within *flashLoan()* and the real limit would therefore be **type(uint256).max**.

Recommended mitigation

Either enforce the limit or change the *maxFlashLoan()* function. In general, consider potentially introducing a reasonable limit. There are for instance protocols that do unchecked operations on **uint112** balances because they assume that an overflow is not feasible. Having no limit could lead to problems with such protocols. Of course, the underlying issue is not

within eBTC in such situations, but trying to avoid any integration issues might still be desirable.

Team Response

Acknowledged.

TRST-M-3 Temporary DoS due to stETH index syncing update frequency assumption

- **Category:** Logical errors
- **Source:** CdpManagerStorage.sol
- **Status:** Fixed

Description

The *_syncIndex()* function in the CdpManagerStorage contract, which is called from the *claimStakingSplitFee*() function, updates the **stFPPSg** index and ensures the index is updated with a specific frequency (currently twice a day) within a particular timeframe. This assertion is made by the require statement in the *_requireValidUpdateInterval()* function, reverting if the index is attempted to be updated too frequently.

```
function _requireValidUpdateInterval() internal view {
    require(
        block.timestamp - lastIndexTimestamp > INDEX_UPD_INTERVAL,
        "CdpManager: update index too frequent"
    );
}
```

The protocol assumes that calling Lido's stETH *getPooledEthByShares()* function with the **DECIMAL_PRECISION = 1e18** constant leads to currently no more than two modifications per day, coinciding with stETH's current rebasing frequency of approximately 24 hours.

However, the index may change in between these expected updates. This can be caused by invoking *Lido.unsafeChangeDepositedValidators()* by the Lido owner, or since Lido (i.e., stETH) is an upgradeable contract, other unforeseen ways to modify the amount of pooled ETH may be introduced, leading to a more frequent change in the index.

Should the index change more frequently than expected, and **_newIndex != _oldIndex** in line 501 of CdpManagerStorage evaluates to true, the subsequently called *requireValidUpdateInterval() function* will revert. This situation could temporarily cause a Denial of Service (DoS) to eBTC's core functions, given that the *_syncIndex()* function, specifically, the *claimStakingSplitFee()* function, is called in many places elsewhere. This DoS would persist until updating the index is once again permitted.

Recommended mitigation

We recommend reassessing the need to enforce a limit on the update frequency of the index and consider eliminating this constraint.

Team response

Fixed by always syncing to the latest index.

TRST-M-4 Macros deployed with LeverageMacroFactory.deployNewMacro are inoperative due to mixed-up arguments for LeverageMacroReference

- **Category:** Logical errors
- **Source:** LeverageMacroFactory.sol
- **Status:** Fixed

Description

Deploying a new macro with the *LeverageMacroFactory.deployNewMacro()* function creates a new contract instance of LeverageMacroReference. However, the arguments supplied to the constructor, **sortedCdps** and **stETH**, are incorrect and mixed up.

Consequently, those values are passed to the LeverageMacroBase constructor in the wrong order as well, leading to broken functionality.

Recommended mitigation

We recommend swapping the **sortedCdps** and **stETH** arguments supplied to the LeverageMacroReference constructor in the *LeverageMacroFactory.deployNewMacro()* function.

Team response

Fixed here.

TRST-M-5 lastFeeOperationTime is not modified correctly in function _updateLastFeeOpTime() leading to a slower base rate decay

- **Category:** Logical errors
- **Source:** CdpManager.sol
- **Status:** Fixed

Description

Whenever eBTC is redeemed, a redemption fee is charged. This fee consists of a variable base rate (**baseRate**) and a fixed fee floor (**redemptionFeeFloor**). The variable base rate decays with time and increases based on the redeemed eBTC amount.

Whenever eBTC is redeemed, the decayed base rate is calculated with the *_calcDecayedBaseRate()* function, based on the number of minutes that have passed since the last recorded **lastFeeOperationTime**. If at least one minute has passed since the previous **lastFeeOperationTime**, it is subsequently updated to the current time via the *_updateLastFeeOpTime*() function.

```
function _updateLastFeeOpTime() internal {
    uint timePassed = block.timestamp > lastFeeOperationTime
        ? block.timestamp - lastFeeOperationTime
```

```
        : 0;

    if (timePassed >= SECONDS_IN_ONE_MINUTE) {
        lastFeeOperationTime = block.timestamp;
        emit LastFeeOpTimeUpdated(block.timestamp);
    }
}
```

However, if 1.999 minutes have passed, the base rate only decays for 1 minute, while 1.999 minutes are added to **lastFeeOperationTime**. In the worst-case scenario, the base rate will only decay for 1 minute for every 1.999 minutes that have passed. As a result, the base rate is likely to experience a slower decay than anticipated.

Recommended mitigation

Add the rounded down number of elapsed minutes, retrieved by the *_minutesPassedSinceLastFeeOp*() function, to the **lastFeeOperationTime** in the *_updateLastFeeOpTime*() function.

Team response

Fixed by adding the exact amount used.

TRST-M-6 getAccumulatedFeeSplitApplied() can revert in rare edge cases, causing redemptions to fail

- **Category:** Logical errors
- **Source:** CdpManagerStorage.sol
- **Status:** Fixed

Description

The *getAccumulatedFeeSplitApplied()* function in the CdpManagerStorage contract calculates the applied fee split for a given CDP based on the accumulated stETH staking rewards and returns both the fee split and the remaining CDP collateral. The remaining collateral is calculated as the difference between the CDP collateral and the fee split.

The fee split, **_feeSplitDistributed**, is calculated as the product of the CDP's stake and the difference between the current and previous stETH staking reward index.

```
function getAccumulatedFeeSplitApplied(
    bytes32 _cdpId,
    uint _stFeePerUnitg,
    uint _stFeePerUnitgError,
    uint _totalStakes
) public view returns (uint, uint) {
    if (
        stFeePerUnitcdp[_cdpId] == 0 ||
```

```
        Cdps[_cdpId].coll == 0 ||
        stFeePerUnitcdp[_cdpId] == _stFeePerUnitg
    ) {
        return (0, Cdps[_cdpId].coll);
    }

    uint _oldStake = Cdps[_cdpId].stake;

    uint _diffPerUnit = _stFeePerUnitg - stFeePerUnitcdp[_cdpId];
    uint _feeSplitDistributed = _diffPerUnit > 0 ? _oldStake * _diffPerUnit
: 0;

    uint _scaledCdpColl = Cdps[_cdpId].coll * DECIMAL_PRECISION;

    require(
        _scaledCdpColl > _feeSplitDistributed,
        "LiquidationLibrary: fee split is too big for CDP"
    );

    return (_feeSplitDistributed, (_scaledCdpColl - _feeSplitDistributed) /
DECIMAL_PRECISION);
}
```

The *getAccumulatedFeeSplitApplied*() function ensures that the CDP collateral exceeds the fee split, in order to prevent a subtraction underflow error from occurring. However, since this function is called in numerous places throughout the protocol, a failed assertion can lead to the inability to redeem, liquidate or adjust the affected CDP. Most importantly, this could result in a Denial of Service for eBTC redemptions, as these redemptions are processed CDP by CDP.

The conditions necessary for this scenario to happen involve the **Cdps[_cdpId].stake** value, which grows over time due to the "corrected stake" mechanism, surpassing **Cdps[_cdpId].coll** by a substantial factor, and **_diffPerUnit** reaching or exceeding **DECIMAL_PRECISION = 1e18**. This implies that **_stFeePerUnitg** is greater than or equal to **2e18**, occurring if 1 stETH equals 2 ETH (i.e., the accumulated stETH staking rewards ROI is 100%). Given the current stETH APR of 4.7%, this would take approximately 15 years.

Recommended mitigation

Consider handling the case where the fee split surpasses the CDP collateral more gracefully, e.g., by setting the fee split to the CDP collateral and the remaining collateral to zero.

Team response

Fixed by adding this edge case check.

TRST-M-7 EBTCToken is not compliant with the EIP-2612 standard

- **Category:** Specification issues
- **Source:** EBTCToken.sol

- **Status:** Fixed

Description

The EBTCToken contract intends to adhere to the EIP-2612: Permit Extension for EIP-20 Signed Approvals standard. However, the *DOMAIN_SEPARATOR()* function, which is required by the standard, is missing. Instead, the incorrectly named *domainSeparator()* function is implemented.

Consequently, the EBTCToken contract is not compliant with the EIP-2612 standard.

Recommended mitigation

Rename the *domainSeparator()* function to comply with the EIP-2612 standard.

Team response

FIxed by adding the function.

Low severity findings

TRST-L-1 MAX_PRICE_DIFFERENCE_BETWEEN_ORACLES code and comment discrepancy

- **Category:** Specification issues
- **Source:** PriceFeed.sol
- **Status:** Acknowledged

Description

The comment within *_bothOraclesSimilarPrice()* states that the maximum allowed deviation should be 3%, but the value of **MAX_PRICE_DIFFERENCE_BETWEEN_ORACLES** corresponds to a maximum allowed deviation of 5%.

Recommended mitigation

Change the comment or the implementation, depending on what the intended value is.

Team response

Acknowledged.

TRST-L-2 HintHelpers._calculatePartialRedeem() returns stale collateral amount

- **Category:** Logical errors
- **Source:** HintHelpers.sol
- **Status:** Acknowledged

Description

The *HintHelpers._calculatePartialRedeem()* function calculates the remaining CDP collateral amount after a partial redemption. However, the returned **newColl** value is stale and does not reflect the actual collateral amount after the partial redemption. This is because the received collateral amount (**collToReceive**) is not deducted from the collateral amount (**newColl**).

Consequently, this leads to a misleading value of **partialRedemptionNewColl** in *HintHelpers.getRedemptionHints()*.

Recommended mitigation

We recommend deducting the received collateral amount (**collToReceive**) from the returned collateral amount (**newColl**) in the *HintHelpers._calculatePartialRedeem()* function.

Team response

Acknowledged.

TRST-L-3 HintHelpers.getRedemptionHints() returns misleading partialRedemptionHintNICR hint if partial redemption causes remaining CDP collateral to fall below the minimum allowed collateral

- **Category:** Logical errors
- **Source:** HintHelpers.sol
- **Status:** Acknowledged

Description

If a partial redemption would lead to a CDP's remaining collateral falling below the minimum allowed collateral (**MIN_NET_COLL**), the *HintHelpers.getRedemptionHints()* function returns a misleading **partialRedemptionHintNICR** hint. This occurs as the last evaluated **partialRedemptionHintNICR** hint, determined by the *_calculatePartialRedeem()* function in line 103, is not reset to 0 before being returned to the caller.

This is misleading as there is no partial redemption in this case. Hence, the **partialRedemptionHintNICR** hint should be 0.

This finding does not pose any direct security impact despite the misleading hint.

Recommended mitigation

We recommend resetting the **partialRedemptionHintNICR** hint to 0 in the *HintHelpers.getRedemptionHints()* function if the CDP's remaining collateral falls below the minimum allowed collateral in line 111.

Team response

Acknowledged.

TRST-L-4 Renouncing the ownership of the FeeRecipient contract will send swept tokens to zero address

- **Category:** Logical errors
- **Source:** FeeRecipient.sol
- **Status:** Acknowledged

Description

The FeeRecipient contract is assumed to receive fees collected by the protocol. An authorized user can call the *sweepToken()* function to transfer the specified token and amount to the current owner of the contract, depicted by the *owner()* function. However, if the ownership of the contract got renounced via the *Ownable.renounceOwnership* function, the *owner()* function will return the zero address.

```
function sweepToken(address token, uint amount) public requiresAuth {
    uint256 balance = IERC20(token).balanceOf(address(this));
    require(amount <= balance, "FeeRecipient: Attempt to sweep more than
balance");

    IERC20(token).safeTransfer(owner(), amount);
}
```

Consequently, sweeping tokens results in an unrecoverable loss of tokens due to the transfer to the zero address.

Recommended mitigation

Consider preventing renouncing the ownership of the FeeRecipient contract by overriding the *Ownable.renounceOwnership()* function, or, alternatively, add a parameter to the *sweepToken()* function to specify the recipient of the swept tokens.

Team response

Acknowledged.

TRST-L-5 Potential DoS due to Chainlink price feed decimals mismatch

- **Category:** Overflow flaws
- **Source:** PriceFeed.sol
- **Status:** Fixed

Description

While calculating the price of stETH/BTC in the *_formatClAggregateAnswer()* function, the value of **_ethBtcAnswer** is scaled by the difference in decimals between stETH/ETH (18 decimals) and ETH/BTC (8 decimals).

```
function _formatClAggregateAnswer(
    int256 _ethBtcAnswer,
    int256 _stEthEthAnswer,
```

```
    uint8 _ethBtcDecimals,
    uint8 _stEthEthDecimals
) internal view returns (uint256) {
    return (((10 ** (_stEthEthDecimals - _ethBtcDecimals)) *
        (uint256(_ethBtcAnswer) * LiquityMath.DECIMAL_PRECISION)) /
uint256(_stEthEthAnswer));
}
```

However, in the unlikely event that the decimals of one of the two used Chainlink price feeds changes, causing **_stEthEthDecimals** to be less than **_ethBtcDecimals**, the result of the calculation will revert with an underflow error. This leads to a denial of service of the eBTC protocol.

Recommended mitigation

Consider the possibility of **_stEthEthDecimals** being less than **_ethBtcDecimals** and scale **_ethBtcAnswer** accordingly.

Team response

Fixed and fuzzed by adding logic that adapts to decimals.

.

Additional recommendations

Trust Security was engaged for a pre-audit check in which some additional recommendations (regarding the test suite, deployment setup, CI/CD pipeline, and operational security) were already outlined.

Use of deprecated *Lido.getOracle()* function

Syncing the global **stFPPSg** index via the *CdpManagerStorage._syncIndex()* function too frequently is prevented by checking the time elapsed since the last sync in the *CdpManagerStorage._requireValidUpdateInterval()* function. The function reverts if the time elapsed is less than or equal to the minimum update interval **INDEX_UPD_INTERVAL**.

The **INDEX_UPD_INTERVAL** is initialized and updated by calling the *CdpManager.syncUpdateIndexInterval()* function. This function calls the *Lido.getOracle()* function to retrieve the beacon chain config (**epochsPerFrame**, **slotsPerEpoch**, **secondsPerSlot**) used to calculate the minimum update interval.

However, the *Lido.getOracle()* function is deprecated.

Consider retrieving the beacon chain config from Lido's HashConsensus contract in the *CdpManager.syncUpdateIndexInterval()* function:

```
ILidoLocator _locator = collateral.getLidoLocator();
BaseOracle _accountingOracle = BaseOracle(_locator.accountingOracle());
IConsensusContract _consensusContract =
_accountingOracle.getConsensusContract();

(, uint256 epochsPerFrame, ) = _consensusContract.getFrameConfig();
(uint256 slotsPerEpoch, uint256 secondsPerSlot, ) =
_consensusContract.getChainConfig();
```

Team response

Fixed by reading the index from stETH: _readStEthIndex.

Open TODO comments left in the code

The code contains several TODO comments, referring to possible unaddressed improvements and optimizations. It is considered good practice to assess and address those comments before deploying the code to production.

Moreover, the *Governor* contract contains a few functions (*getActiveRoles()*, *getCapabilitiesForTarget()*, *getCapabilitiesByRole()*) that revert with an error string stating that they are not implemented yet. We therefore ignored these functions during this audit. It

is recommended to ensure that these functions will also be audited at least once before deploying the protocol.

Team response

Acknowledged.

Outdated comments and references to the Liquity protocol

As eBTC is a modified fork of the Liquity protocol, the code contains several outdated comments mentioning Liquity and Liquity-specific mechanisms. For instance, the ActivePool contract includes a comment referring to Liquity's Stability pool and default pool:

```
/**
 * The Active Pool holds the collateral and EBTC debt (but not EBTC tokens)
for all active cdps.
 *
 * When a cdp is liquidated, it's collateral and EBTC debt are transferred
from the Active Pool, to either the
 * Stability Pool, the Default Pool, or both, depending on the liquidation
conditions.
 */
```

To prevent confusion and to ensure the comments explain the underlying eBTC mechanisms, it is recommended to update the comments and remove references to Liquity.

Team response

Acknowledged.

Use of SafeMath can be avoided to save gas

Due to using Solidity version 0.8, overflow protection is implemented by default at the language level. Consequently, using OpenZeppelin's SafeMath library is redundant and can be removed to save gas.

Team response

Fixed by removing safeMath in the code.

Missing NatSpec comments

It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI) as stated in the Solidity NatSpec documentation. We

observed that appropriate @notice, @param and @return fields are missing throughout the codebase in many publicly available functions.

Team response

Acknowledged and improved.

Unused code

The code contains several unused functions and variables. It is recommended to remove all unused code to prevent any confusion and to reduce the attack surface. For instance, the following function and struct properties are unused:

- CdpManager.sol: *decayBaseRateFromBorrowing()*
- ICdpManagerData.sol: **LiquidationTotals.totalCollToRedistribute** and **LiquidationValues.collToRedistribute**

Team response

Fixed.

Centralization risks

No maximum fee enforced for ERC3156FlashLender

The variable **maxFeeBps** should be 1,000 BPS according to the comment. But it is set to **MAX_BPS**, i.e., 10,000 BPS. Therefore, there is currently no maximum fee and a value of up to 100% could be set.

Team response

Fixed here.

Governance can grant permissions to mint and burn eBTC tokens arbitrarily

eBTC can be minted via the *EBTCToken.mint()* and burned with the *EBTCToken.burn()* function. Both functions are permissioned and only callable by the BorrowerOperations or CDPManager contract or an authorized address (i.e., governance).

In the latter case, the supply of eBTC changes, which may lead to undercollateralized eBTC if additional tokens are minted.

The rationale behind allowing governance (or any other authorized address) to mint/burn eBTC seems to be extending the system and composing with other protocols.

Nonetheless, it imposes a certain risk for users holding eBTC and undermines the trust assumptions.

Team response

Acknowledged.

Redemptions can be stopped

The CdpManager contract has a function *setRedemptionFeeFloor()* which allows governance to update the **redemptionFeeFloor** variable. The function allows a broad range of values, it just enforces that the updated value is above **MIN_REDEMPTION_FLOOR_FEE** and below **DECIMAL_PRECISION** (100%). A redemption fee of 100% would lead to a situation where redemptions no longer work, as the function *_calcRedemptionFee()* would always revert. The redemption fee floor can therefore be abused by governance to set unreasonably high fees or even stop redemptions completely.

Additionally, by setting the value of **beta**, the denominator in the base rate calculation for redemptions, to either zero, causing a division by zero error, or a very large value, redemptions can be stopped as well.

Team response

Acknowledged.

eBTC flash loan fee recipient can be purposely set to disable flash loan functionality

eBTC flash loans, implemented via the *flashLoan()* function in the BorrowerOperations contract, are subject to an optional fee. After the callback succeeded, the borrowed amount plus the optional fee is transferred to the **feeRecipientAddress** and the **amount** of eBTC is burned.

The eBTC implementation, EBTCToken, validates the recipient address of a transfer in the *_requireValidRecipient()* function and reverts if the tokens are transferred to the zero address, the eBTC token contract itself, the CDP manager contract (**cdpManagerAddress**), and the borrower operations contract (**borrowerOperationsAddress**).

The **feeRecipientAddress** in the BorrowerOperations contract can be changed by governance to any address except the zero address. However, setting the **feeRecipientAddress** to any of the prohibited eBTC recipient addresses will cause the transfer to revert and effectively disables the flash loan functionality.

It is advised to add additional checks to the *setFeeRecipientAddress()* function in the BorrowerOperations contract to prevent setting the **feeRecipientAddress** to any of the prohibited eBTC recipient addresses.

Team response

Acknowledged, flashLoaning is also pausable by governance.

Systemic risks

stETH Dependency

eBTC inherently depends on stETH and its correct functioning. Any bug or vulnerability in stETH may impact eBTC significantly and lead to a temporary or permanent loss of funds.

Moreover, the immutable eBTC contracts build on top of the mutable stETH contracts: While it is very unlikely that Lido significantly changes the stETH token, the token contract is mutable and thus upgradeable. The interface and internal workings could change over time. In contrast, the eBTC protocol remains immutable. Moreover, stETH's capability to pause certain functions, most notably the ability to transfer tokens (see here), poses a risk to eBTC. Should stETH be paused, eBTC would be impacted significantly. The following stETH functionalities can be paused:

- token transfer
- handling oracle report (i.e., rebasing)
- depositing ETH into Lido (i.e., minting stETH)

A pause of these functionalities may lead to a significant reduction of the stETH/ETH rate (because of market panic), in which case a lot of positions could become liquidatable. However, this would fail when token transfers are paused. Such a black swan scenario could therefore lead to a (temporary) depeg of eBTC.

These risks are present for any protocol that builds on top of stETH, and they cannot be avoided completely. However, users should be aware of them.

Team response

Acknowledged.