PUBLIC

Code Assessment

of the Hedgehog Protocol Smart Contracts

Jun 02, 2025

Produced for

HEDGEHOG PROTOCOL

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Hedgehog Team,

Thank you for trusting us to help Hedgehog with this security audit. Our executive summary provides an overview of subjects covered in the latest reviewed contracts of Hedgehog Protocol according to Scope to support you in forming an opinion on their security risks.

Hedgehog implements a hedging instrument for the change of the base fee on Ethereum mainnet. Hedgehog has forked Liquity v1 and adapted the smart contracts to implement the gas derivative used for hedging. This review was limited to the smart contract modifications applied by Hedgehog, under the assumption of Liquity's codebase being safe. However, it is important to acknowledge that any potential bug in Liquity could impact Hedgehog too.

The most critical subjects covered in our review are functional correctness and access control. Initially, security regarding functional correctness was improvable, while security regarding access control was satisfactory. A set of severe issues were introduced in the initial versions of the codebase, mainly from two changes:

1. The debt token BaseFeeLMA was using 6 decimals
2. The Base fee oracle returned a price with 1 decimal and the token pair BaseFeeLMA:ETH

These changes were not reflected consistently in the codebase, hence breaking multiple pre-existing functions. These issues have been resolved in the final version.

In Version 4 of the codebase a new functionality to enforce a system-wide withdrawal limit was added. The implementation of this functionality introduced a set of new bugs, the most severe being Liquidations are blocked from Withdrawal Limit. These findings have been resolved in the final version.

The general subjects covered are trustworthiness, documentation, and testing. Security regarding trustworthiness have been improved throughout the review, but privileged roles in non-core contracts can still block user operations, see Trust Model and Roles. Documentation and specification are improvable and can be extended to describe the changes more thoroughly and systematically. The testing suite has been enhanced in the later iterations, but testing remains improvable. The tested contracts do not always match the deployed contracts (i.e. *HogToken on Base*) and not all code paths are covered by tests. Hence, we recommend further testing.

The final code version has some lower severity findings were (partial) risks have been accepted (see Open Findings).

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	5
• Code Corrected	5
High-Severity Findings	11
• Code Corrected	10
• Specification Changed	1
Medium-Severity Findings	27
• Code Corrected	20
• Specification Changed	3
• Code Partially Corrected	1
• Risk Accepted	2
• Acknowledged	1
Low-Severity Findings	26
• Code Corrected	17
• Specification Changed	2
• Code Partially Corrected	2
• Risk Accepted	3
• Acknowledged	2

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Hedgehog Protocol repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	11 Mar 2024	5d507ca2b2c0dede6ac6f8efe63169cbe78a3a4c	Initial Version
2	18 May 2024	1773f6fb8e5490a67b6eca4342df0452ac7e4a2b	Version 2
3	03 Jul 2024	242c53a23241679a39d434a5a8a5eef9d7381ad8	Version 3
4	26 Aug 2024	96670a9d1ec8cdbf71b9e9439a52dc8bd39b9af5	Version 4
5	24 Feb 2025	f07d83daea7349db62493307b69e2f274c13fb63	Version 5
6	29 Apr 2025	47daa8e63885edbcc08f14e17bd3a716e782d518	Version 6
7	21 May 2025	9e9b8156cc8ada17d1bdeed0bce148ea43326295	Final Version

For the solidity smart contracts, the compiler version `0.8.19` was chosen.

The following files in the folder `contracts` are in scope:

```
dependencies
    BaseMath.sol
    CheckContract.sol
    HedgehogBase.sol
    IERC2612.sol
    LiquityMath.sol
    LiquitySafeMath128.sol
HOG
    CommunityIssuance.sol
    HOGToken.sol
ActivePool.sol
BaseFeeLMAToken.sol
BaseFeeOracle.sol
BorrowerOperations.sol
CollSurplusPool.sol
DefaultPool.sol
FeesRouter.sol
GasPool.sol
HintHelpers.sol
PriceFeed.sol
SortedTroves.sol
StabilityPool.sol
TroveManager.sol
```

In Version 3 and Version 4 the following contracts were included in scope:

```
BaseFeeOracleArb.sol (replaced BaseFeeOracle.sol in version 3)
PriceFeedArb.sol (replaced PriceFeed.sol in version 3)
BorrowerOperationsArb.sol (replaced BorrowerOperations.sol in version 3)
TroveManagerArb.sol (replaced TroveManager.sol in version 3)
```

The following contracts were removed from the scope:

```
BaseFeeOracle.sol
PriceFeed.sol
TroveManager.sol
BorrowerOperations.sol
```

In Version 5, the previous change was reverted, and the following contracts were included in scope:

```
BaseFeeOracle.sol (replaced BaseFeeOracleArb.sol in version 5)
PriceFeed.sol (replaced PriceFeedArb.sol in version 5)
BorrowerOperations.sol (replaced BorrowerOperationsArb.sol in version 5)
TroveManager.sol (replaced TroveManagerArb.sol in version 5)
```

The following contracts were removed from the scope:

```
deprecated
    BaseFeeOracleArb.sol
    PriceFeedArb.sol
    BorrowerOperationsArb.sol
    TroveManagerArb.sol
```

In Version 6, the following files were removed from the scope:

```
LiquitySafeMath128.sol
```

2.1.1 Excluded from scope

Any contracts that are not explicitly listed above are out of the scope of this review. The contracts in folders `helpers` and `LPRewards` are explicitly excluded from scope. Third-party libraries are also out of the scope of this review.

The implementations of the collateral token `WStETH` in Ethereum mainnet and Base (Ethereum L2), and the respective bridge are not in scope of this review. Furthermore, the HOGToken contract is planned to be deployed on Ethereum mainnet, while the bridged token on Layer 2 chain is not in scope of this review. In this report, we assume the protocol is deployed on the Base chain, hence the correctness of the codebase if deployed on other chains is not in scope of this review.

We assume that the collateral token `WStETH` is an ERC20 token with 18 decimals that has a conversion rate to native ETH of less than 100:1.

Earlier versions of the codebase (Version 1 - Version 4) were developed for deployment on Arbitrum. However, Arbitrum-specific contracts have since been deprecated. The following contracts from earlier versions are no longer in scope and should not be deployed: `BaseFeeOracleArb.sol` , `PriceFeedArb.sol`, `BorrowerOperationsArb.sol` and `TroveManagerArb.sol`.

In this report, we assume the Liquity v1 is safe, and the review is focused on the changes applied by Hedgehog to the smart contracts from Liquity. Therefore, any bug present in Liquity might still be present in Hedgehog.

Finally, the soundness of the financial model was not evaluated.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Hedgehog offers a hedging instrument for the price change of the base fee in Ethereum mainnet. Hedgehog has forked Liquity v1 and adapted the smart contracts to implement the hedging instrument. In this report, we focus on the smart contract changes applied by Hedgehog. A more complete documentation of Liquity's smart contracts and functionalities can be found here.

EIP-1559 introduced a new pricing model for the gas costs of transactions in Ethereum mainnet. EIP-1559 splits the gas costs into two parts: 1) base fee, which represents the minimum amount of Ether a user must pay to have their transactions included in a block; 2) priority fee, or the tip paid to validators for including a transaction in a block. While the priority fee is chosen freely by users, the base fee is dynamically adjusted based on changes in network demand to maintain a target level of block space utilization. When the network is congested, the base fee increases, hence making transactions more expensive. Conversely, when the network is less congested, the base fee decreases, providing users with lower transaction fees. The base fee changes with at most 12.5% (higher/lower) depending on the utilization of the previous block.

Hedgehog implements an ERC20-compliant token named `BaseFeeLMAToken` that is pegged to the base fee in the Ethereum mainnet, hence enabling users to hedge gas costs of future transactions. Obviously, the main difference from Liquity is that the debt token `BaseFeeLMAToken` does not maintain a fixed value in terms of USD, instead its value follows the base fee in mainnet. We detail in the next section the differences from Liquity.

2.2.1 Hedgehog customizations

Hedgehog plans to deploy the smart contracts on a Layer-2 chain, Arbitrum, hence the codebase is adapted accordingly.

Collateral asset

Hedgehog uses the WStETH (Wrapped Staked ETH) token as a collateral asset instead of Ether used by Liquity. Therefore, functionalities that transfer the collateral asset have been revised to integrate with a WStETH, which is an ERC20 token.

Oracles

The Chainlink and Tellor oracles have been replaced with two instances of `BaseFeeOracle` which are deployed and maintained by Hedgehog. These oracles return the base fee price in Ethereum mainnet. A trusted off-chain service monitors Ethereum blocks and computes the price of `BaseFeeLMAToken` as a logarithmic moving average of the last 50 blocks in mainnet. The smallest is assigned to the newest value and the largest weight to the oldest value. The oracle returns the price of `BaseFeeLMAToken` quoted in wstETH (`BaseFeeLMA/wstETH`).

New prices should be published roughly every 14 minutes before an oracle is considered frozen by the contract `PriceFeed`. If the main oracle is frozen, `PriceFeed` relies on the prices returned by the backup oracle, which is another instance of `BaseFeeOracle` in the case of Hedgehog.

Collateralization parameters

Hedgehog has changed the minimum collateralization ratio from 110% to 150%. Similarly, the critical collateralization ratio was updated from 150% to 200%. These higher parameters affect the capital efficiency, liquidations, and the pegging of the `BaseFeeLMAToken` to the actual base fee in mainnet. In this report, we do not analyze in depth the financial impact of these changes.

The increase of minimum collateralization ratio (MCR) to 150% and critical collateralization ratio (CCR) reduce the capital efficiency as users can borrow less debt tokens for the same amount of collateral.

During normal mode, troves (borrowers' positions) are liquidated if their collateralization ratio falls below MCR (150%), while in recovery mode troves with collateralization ratio below CCR (200%) are also liquidated. Therefore, troves in Hedgehog are expected to have a higher collateralization ratio than in Liquity.

The minimum collateralization ratio plays a key role in maintaining the pegging of the debt token (`BaseFeeLMAToken`) to its actual value (base fee in mainnet). A higher MCR weakens the peg of `BaseFeeLMAToken` as the token can trade in secondary markets up to 150% of its real value. If the `BaseFeeLMAToken` price in a secondary market exceeds this limit (150%), then there is an arbitrage opportunity as one can make a profit by minting new `BaseFeeLMAToken` (borrowing) and selling them immediately.

Borrowing and redeeming fees

Hedgehog charges fees on borrow and redeem operations. The dynamic fees serve as a throttling mechanism by charging higher fees for operations that change significantly the total supply of `BaseFeeLMAToken` or the collateral held by the protocol. However, the way the dynamic fees are computed is different from Liquity. Importantly, the borrow rate does not depend on the redemptions anymore.

The borrowing rate is calculated with the following formula:

$$BorrowRate = BorrowFloor + BorrowBaseRate * BorrowDecayFactor^{Minutes} + \frac{IssuedBFee}{TotalBFeeSupply}$$

The borrowing decay factor is chosen such that after roughly 78 minutes the base rate for borrowing decays by 50%. Note that the borrowing rate does not depend on redemptions, hence the borrowing is not throttled after large redemptions. There is no cap on the borrowing rates, and they can go up to 100%. Hedgehog charges a borrow fee also during recovery mode although more restrictions apply in that setting: The trove's CR should be above CCR to improve the overall health of the system, hence making borrowing less attractive in recovery mode.

The redemption rate is calculated as follows:

$$RedeemRate = RedeemFloor + RedeemBaseRate * RedeemDecayFactor^{Minutes} + \frac{RedeemCollateral}{TotalCollateral}$$

The redeem decay factor is chosen such that the redeem base rate decays by 50% after 12 hours. The fee rate is doubled from Liquity, and they are not capped (Liquity caps the borrowing fee rate at 5%).

Fees router

The contract `FeesRouter` is new in Hedgehog and it manages the fee distribution from borrow and redeem operations. The Staking contract that received fees in Liquity has been removed. `FeesRouter` has a privileged role `SETTER` that can set configurations for the distribution of fees charged in debt or collateral tokens.

> Each configuration includes up to 3 arbitrary addresses that should receive fees in a predefined ratio. Configurations for fees in debt tokens and collateral tokens are different.

HOG token

`HOGToken` is an ERC20-compliant token that implements the permit extension as specified in EIP-2612. The total supply of `HOG` token is hard coded at 100 million and the whole supply is minted in the constructor to an arbitrary `multisigAddress` provided by the deployer. Differently from Liquity, no `HOG` tokens are allocated to special accounts such as bounty entitlements or LP Rewards entitlements. The account `multisig` is considered trusted and it should distribute `HOG` tokens to other contracts/accounts in the system as expected, e.g., to community issuance.

`HOGToken` is the only contract deployed on Ethereum mainnet, therefore the trusted `multisigAddress` should bridge tokens to the layer-2 chain where the protocol is deployed.

The initial `HOG` holder `multisigAddress` should send the expected number of tokens to the contract `CommunityIssuance`. The latter releases `HOG` tokens to users of the system based on a predefined

curve. Accounts with the privileged role `DISTRIBUTION_SETTER` can update the parameters `HOGSupplyCap` and `ISSUANCE_FACTOR` that alter the issuance curve.

Removed functionalities

The functionality to reward frontend providers has been removed from Hedgehog. The protocol token HOG is minted exclusively to the `multisig` account which is the sole holder after the contract is deployed. The staking functionality of `HOG` tokens has been removed. The protocol fees are handled by the contract `FeesRouter` as described above.

2.2.2 Trust Model and Roles

Several contracts in Hedgehog Protocol have privileged accounts that need to be trusted to behave correctly for the protocol to function as expected. We detail these accounts below.

In general, we assume the deployers of contracts are trusted to initially configure contracts with the correct parameters. Otherwise, users should not interact with contracts that have been misconfigured.

CommunityIssuance: Any account with the role `DISTRIBUTION_SETTER` or `DISTRIBUTION_SETTER_ADMIN` are considered fully trusted, and they should be carefully protected. If an account holding one of these roles is compromised, they can change freely the issuance curve of HOG tokens to block core functionalities of the system by causing underflows.

HOGToken: The `_multisigAddress` is fully trusted to bridge HOG tokens from Ethereum mainnet to Arbitrum and distribute them to other contracts of the system as expected, e.g., to CommunityIssuance contract.

BaseFeeOracle: Any account with the role `ULTIMATE_ADMIN` or `SETTER` is considered fully trusted, and they should be carefully protected. If an account holding one of these roles is compromised, they can publish false prices and liquidate healthy troves or mint arbitrary number of debt tokens by opening undercollaterized troves.

FeesRouter: Any account with the role `ULTIMATE_ADMIN` or `SETTER` is considered fully trusted, and they should be carefully protected. If an account holding one of these roles is compromised, they can redirect fees to arbitrary addresses or remove configurations to block core functionalities of the system.

Finally, the collateral token `WStETH` in Arbitrum is considered fully trusted, including its proxy admin and the bridging system.

2.2.3 Changes in Version 2:

- The BaseFeeOracle has been revised to return the price for the pair `BaseFeeLMA:wStETH` and uses 18 decimals.
- The ERC20 token `BaseFeeLMA` uses 18 decimals.
- The `PriceFeed` target digits are 18 decimals.
- Troves can be modified only once in a block.
- The parameter `_BaseFeeLMAAmount` in `openTrove()` excludes the gas compensation.

2.2.4 Changes in Version 3:

- Fixes of reported issues Locking of Troves Is Longer Than Specified and Misleading Variable Name in BaseFeeOracle are implemented in new contracts, namely `BaseFeeOracleArb.sol`, `PriceFeedArb.sol`, `BorrowerOperationsArb.sol` and `TroveManagerArb.sol`.
- The contracts `BaseFeeOracle.sol`, `PriceFeed.sol`, `BorrowerOperations.sol` and `TroveManager.sol` do not include the latest fixes, therefore they should not be deployed. These contracts were removed from the scope of the audit starting at Version 3.

- The issuance can no longer underflow when it is being reduced, so the `DISTRIBUTION_SETTER` can no longer block the system by reducing the issuance.

2.2.5 Changes in Version 4:

The Hedgehog introduced withdrawal limits to address issues Attacker With Sufficient Funds Can Lower Redemption Fees and Reducing Fees by Splitting Transactions:

On each deposit:

- The new withdrawal limit is set to the old limit plus 50% of the deposit amount.
- If the new limit exceeds the previous collateral value, it is set to 50% of the total collateral and resetting any withdrawals.

On each withdrawal:

- The withdrawn collateral is deducted from the withdrawal limit.
- Withdrawals from adjusting a trove, closing a trove, and redemptions can only use up to 80% of the current withdrawal limit.
- Withdrawals resulting from liquidations face no restriction but still update the withdrawal limit.
- Each withdrawal updates the *lastWithdrawalTimestamp*.

The withdrawal limit recovers linearly over time: After a wait time of *EXPAND_DURATION* (720 minutes or 12 hours in (Version 4)), the limit is fully recovered.

2.2.6 Changes in Version 5:

The contracts have been updated to enable deployment on the Layer-2 chain Base instead of Arbitrum.

The withdrawal limits introduced in (Version 4) have been modified.

On each deposit:

- The new withdrawal limit is set to the old limit plus 50% of the deposit amount.

On each withdrawal:

- If the collateral in the active pool falls below 10 WStETH *after* the withdrawal, the withdrawal limit is set to the remaining collateral amount.
- The withdrawn collateral is deducted from the withdrawal limit.
- Withdrawals from adjusting a trove, closing a trove, and redemptions can only use up to 80% of the current withdrawal limit.
- Withdrawals resulting from liquidations face no restriction but still lower the withdrawal limit.
- Each withdrawal updates the *lastWithdrawalTimestamp* and deducts the withdrawn amount

Withdrawal limits:

The withdrawal limit at any time t is determined by adding the previous limit to the recovered limit:

$$WithdrawalLimit_t = WithdrawalLimit_{t-1} + recoveredLimit_t$$

The withdrawal limit recovers linearly over time after the last withdrawal, gradually increasing until it reaches the total collateral-based limit over a period of `EXPAND_DURATION` (set to 720 minutes / 12 hours in Version 5).

After 4 hours (or 1/3 of the total recovery duration), the recovered limit is:

$$recoveredLimit_t = 1/3 * (totalCollBasedLimit - WithdrawalLimit_{t-1})$$

The limit continues recovering until it reaches the total collateral-based limit, which is calculated as the sum of 50% of the active collateral and the `WITHDRAWAL_LIMIT_THRESHOLD` (set to 10 WStETH in Version 5):

$$totalCollBasedLimit_t = 50\% * ActiveCollateral + 5$$

Note: The Active Collateral is retrieved after collateral has been sent out to the user. Therefore, the total collateral-based limit will be calculated using the post-withdrawal active collateral and does not include the withdrawn amount.

2.2.7 Changes in Version 6:

In Version 6 withdrawal limits were removed. Furthermore, the `SafeMath` libraries that were in the original Liquity code, but no longer needed with the current compiler version, were removed. A recent change by the Liquity team for one of the known issues of the system was integrated into the Hedgehog code: https://github.com/liquity/dev/pull/1044.

Last but not least, some gas optimizations were made.

2.2.8 Changes in Version 7:

In Version 7 the precision of the computation of the collateralization ratio has been increased.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	4

- Dependency on Current Block Time Risk Accepted
- Reducing Fees by Splitting Transactions Risk Accepted
- Missing Configurations in FeesRouter Compromise Accounting Code Partially Corrected
- Slow Expansion of the BaseFeeLMAToken Supply Due to High Costs Acknowledged

Low-Severity Findings	7

- Incorrect Rate Adjustment Acknowledged
- Gas Inefficiency in BaseFeeOracle Code Partially Corrected
- Attacker With Sufficient Funds Can Lower Redemption Fees Risk Accepted
- Redemptions Without Base Rate Increase Risk Accepted
- Incorrect Code Comments Code Partially Corrected
- Known Issues From Liquity Are Present in Hedgehog Risk Accepted
- Lack of Documentation Acknowledged

5.1 Dependency on Current Block Time

Design Medium Version 4 Risk Accepted

CS-HOG-066

The Price feed verifies the staleness of a price update by comparing the current block number with the block number of the last price update. In Version 4, Hedgehog uses a *TIMEOUT* of 1600 to determine if a price update is stale. That means that a price would have to occur within the last 1600 blocks to be considered fresh.

The current block time on Arbitrum is 0.25 seconds, so a timeout of 1600 blocks translates into 400 seconds or ~33 Etheruem Mainnet blocks.

Note that this condition would change if the block time on Arbitrum is changed. The current iteration of ArbOS allows a block time as low as 0.1 seconds. More information can be found here.

In that case the timeout would be 1600 * 0.1 = 160 seconds or ~13 Ethereum Mainnet blocks. In case the block time is lowered even further, the project could risk being permanently stuck in a state where the price feed is considered stale, if Hedgehog fails to update the price fast enough.

Changes in Version 5:

The protocol will now be deployed onto Base Chain instead of Arbitrum. Base Chain has a block time of 2 seconds, but other OP Chains (e.g., Unichain) have a 1-second block time, with plans to reduce it to 250ms in the future.

Risk accepted:

The Hedgehog has accepted this risk but has chosen to keep the code unchanged. Their response:

> Acknowledged. We are tracking the block times and would make adjustments to the deployment if anything changes. Currently in tests we've updated the timeout to 200 which corresponds to the same 400 seconds on Base blockchain

If the block time on Base Chain decreases by too much then `PriceFeed` can start to consider all prices as *frozen* and fallback to the last good price before the change. Hedgehog plans to deploy a new version of the protocol in that case and expects users to migrate their positions.

5.2 Reducing Fees by Splitting Transactions

Design Medium Version 2 Risk Accepted

CS-HOG-057

The borrowing fee depends on the amount of tokens borrowed compared to the total supply.

$$BorrowRate = BorrowFloor + BorrowBaseRate * BorrowDecayFactor^{Minutes} + \frac{IssuedBFee}{TotalBFeeSupply}$$

By splitting the borrowing operation into multiple smaller operations, users can effectively reduce the overall borrowing fees. This is because each individual call increases the total supply by a smaller percentage, and the total supply grows with each operation.

Consider a scenario where the current token supply is 10 million tokens, and the base borrowing rate (BorrowBaseRate) is 0. A user intends to borrow 1 million tokens.

Borrowing 1 million tokens in a single operation would result in a borrowing fee calculated as:

$$BorrowRate = 0.5\% + \frac{1}{10} = 10.5\%$$

However, if the user splits the borrowing into two transactions of 500,000 tokens each, the borrowing fee for each transaction is calculated as follows:

$$BorrowRate = 0.5\% + \frac{0.5}{10} = 5.5\%$$

$$BorrowRate = 0.5\% + \frac{0.5}{10.5} = 5.3\%$$

$$TotalBorrowRate = \frac{5.5\% \times 0.5 + 5.3\% \times 0.5}{10.5} = 5.4\%$$

The user can roughly half the fee they are paying by splitting the borrowing into two transactions. A sophisticated user can split the borrow further to lower the fees even more, and trade this off against the gas costs of the additional transactions.

By splitting the borrowing into two operations, the user can significantly reduce the total fee paid. A more sophisticated approach of further splitting the borrow can reduce the fees even more, balanced against the gas costs of additional transactions.

The underlying issue is that the amount of fees paid is *path dependent*, meaning that the fees paid depend on the path taken to reach the final state. Instead, each marginal token borrowed should have the same fee independent of if they were borrowed in a large or small transaction.

Similarly, the redemption fees are also path dependent. The redemption fee is calculated as follows:

$$RedeemRate = RedeemFloor + RedeemBaseRate * RedeemDecayFactor^{Minutes} + \frac{RedeemCollateral}{TotalCollateral}$$

Note that splitting a redemption into multiple transactions does lower the redemption share (redeemCollateral/TotalCollateral) and hereby the cost. Each redemption further decreases the collateral still locked in the system and hereby having an effect in the other direction. However, for all redemptions that are not the last one, the effect of the decreased collateral is smaller than the effect of the decreased share of the total collateral.

Note that Hedgehog expects price discovery to mostly happen on the secondary market, due to the high fees for borrowing and redemption. By exploiting this design flaw, users can effectively reduce the fees paid for borrowing and redemption. This is a design issue, as the fees paid should be independent of the path taken to reach the final state.

Risk accepted:

Hedgehog has accepted the risk but has decided to keep the redemption fee mechanism unchanged. In Version 4 they have introduced withdrawal limits to mitigate the impact of this issue. They answered:

```
Withdrawal limits should increase the economic barrier for redemption-type attacks.
Additionally, they would give the DAO and the HDG team time to compensate for any
loss of collateral at their own expense. This means that not only would it prevent
attacks that might be profitable for users, but it would also provide a way to mitigate
the damage from griefing attacks.
```

Note on audit process: We independently discovered this issue, which is also present in Liquity's codebase. However, it has a higher severity for Hedgehog, as redemption fees play a more critical role in limiting the number of redemptions. The Liquity team's write-up on this issue can be found here.

5.3 Missing Configurations in FeesRouter Compromise Accounting

Correctness | Medium | Version 1 | Code Partially Corrected

CS-HOG-023

The contract `FeesRouter` is responsible for distributing the protocol fees according to predefined configurations. Accounts with `SETTER` role in `FeesRouter` can set and update such configurations. However, if there is no configuration for a given percentage, then no fee is distributed.

In case the fee is charged in debt token, function `distributeDebtFee()` does not mint the respective `BaseFeeLMAToken`, thus the total supply of the debt token gets smaller than the total debt owed by all borrowers. This might render the last trove impossible to close as the circulating supply is less than the repayment amount.

Similarly, function `distributeCollFee()` does not distribute fees from the active pool when the respective configuration is not set in `collFeeConfigs`.

Code partially corrected:

New checks are added in the function `distributeDebtFee()` and `distributeCollFee()` that revert a transaction if the respective configuration is not set. However, the side effect of this approach is that key functionalities such as opening a trove, or liquidations might be blocked if a configuration is not properly set in the contract FeesRouter.

5.4 Slow Expansion of the BaseFeeLMAToken Supply Due to High Costs

Design | Medium | Version 1 | Acknowledged

CS-HOG-027

In Hedgehog's code, the borrowing rate is dependent on the total supply of BaseFeeLMATokens. The formula used to calculate the borrowing rate is:

$$BorrowRate = BorrowFloor + BorrowBaseRate + IssuedBFee/TotalBFeeSupply$$

This formula suggests that a rapid expansion of the token supply is not feasible, as doubling the supply (IssuedBFee = TotalBFeeSupply) would require the borrower to pay 100% of the borrowed amount as fees. Note that paying 100% in fees is not feasible, as the user also must cover the gas compensation.

The maximum amount of BaseFeeLMATokens that can be minted is enforced in `BorrowOperations.openTrove()` as:

$$IssuedBFee = BFeeMintedToUser + GasCompensation + BorrowRate * IssuedBFee$$

In Hedgehog's code, each borrowing event adds the borrowing rate to the `BorrowBaseRate`, which then decays over time with a half-life of approximately 1.3 hours (or 78 minutes). If the initial supply is low, the supply can only increase slowly due to excessive fees on large borrowings and then having to wait until the fee decays back. A profit-oriented minter would likely only accept a much lower percentage fee than the maximum possible, so the actual expansion rate would be much lower. It should be noted that an attacker could grieve the system by creating a very small initial supply, which would require manual intervention to increase the supply or slow down the system for several weeks.

Acknowledged:

Hedgehog has acknowledged the issue and provided the following reply:

```
The BaseFeeLMA economy project differs much from the stablecoin-based Liquity.
Particularly, secondary market is the preferred way of obtaining the token for
speculation or hedging. This is due to expected basefee jumps being only partially
offset by the LMA technique, which would pose excessive liquidation risks and lead
to collateral exhaustion in case of unlimited access to borrowing. Incentives for
secondary market participation are therefore given priority over steady token
supply growth.

A profit-oriented agent is, on the other hand, inclined to mint higher initial
supply within the OCR limit, due to zero baserate set for the initial borrowing
transaction. However, the possibility of supply grieving attack suggests that the
initial supply should be generated by the team immediately upon deployment, as a
security measure.
```

5.5 Incorrect Rate Adjustment

Correctness | Low | Version 6 | Acknowledged

CS-HOG-083

The Borrow Base Rate and the Redemption Rate are computed incorrectly. The reason is the same for both (just in different functions), hence, we explain it for the Borrow Base Rate:

The function `_calcDecayedBorrowBaseRate()` computes the new base rate using `_minutesPassedSinceLastBorrow()`, which contains the following code:

```
(block.timestamp - lastBorrowTime) / SECONDS_IN_ONE_MINUTE;
```

Then later the `lastBorrowTime` is updated as follows:

```
uint256 timePassed = block.timestamp - lastBorrowTime;

if (timePassed >= SECONDS_IN_ONE_MINUTE) {
    lastBorrowTime = block.timestamp;
    emit LastBorrowTimeUpdated(block.timestamp);
}
```

If the time that has passed is 1 minute and 59 seconds, then `_minutesPassedSinceLastBorrow()` will return 1 minute and hence the Borrow Base Rate will be updated based on 1 minute. However, the `lastBorrowTime` will be moved forwards by 1 minute and 59 seconds. Hence, almost two minutes have passed but the Borrow Base Rate has only decayed for one minute. Note that for less regular updates the relative error will not be as big.

In the case of the Redemption Rate, the variable incorrectly updated is `lastRedemptionTime`.

Acknowledged:

The Hedgehog has acknowledged the issue, but has decided to keep the code unchanged. They have provided the following reasoning:

```
it doesn't seem critical but the fix has significant impact on the figures and
we don't want to diverge from Liquity too much in this regard.
```

5.6 Gas Inefficiency in BaseFeeOracle

Design | Low | Version 5 | Code Partially Corrected

CS-HOG-075

The `feedBaseFeeValue` function of the `BaseFeeOracle` contract needs to be called regularly to provide the system with up-to-date base fee prices. Hence, its gas costs are relevant, even on a fairly cheap chain like Base chain.

During its execution a `Response` struct is stored.

1. The `Response` is defined as:

```
struct Response {
    int256 answer; // LogMA50(BaseFeePerGas) * WstETH / ETH ratio in wei
    uint256 blockNumber; // L1 block number from which the last BaseFeePerGas value was retrieved
```

```
    uint256 currentChainBN; // Current network's block number during which structure was updated
    uint256 roundId; // Round during which the structure was updated
}
```

However, values like block numbers and round IDs, do not require 256 bit. With smaller value types, less storage would be consumed by each stored `Response`. This would lower the execution cost of `feedBaseFeeValue()`.

2. The `Response` struct is always stored in a new place. It is unclear whether `Response` entries from e.g. ten rounds ago are still required. Otherwise, old `Response` entries could be overwritten, which would be significantly cheaper gas-wise.

Code partially corrected:

In Version 6, the blockNumber, currentChainBN, and roundId fields are cast to uint64, so that the `Response` struct can be stored in two storage slots instead of four slots. Hedgehog has decided not to overwrite old `Response` entries, providing the following reasoning:

```
We've decided that long-term retrieval of the oracle data is necessary for the protocol transparency
so we've left the behaviour unchanged.
```

5.7 Attacker With Sufficient Funds Can Lower Redemption Fees

Design | Low | Version 2 | Risk Accepted

CS-HOG-056

In Version 2 of Hedgehog's code the redemption fee is calculated based on the proportion of collateral redeemed relative to the total collateral locked in the system. A user can interact with the system only once per block, which prevents adding collateral and removing collateral to the same trove via flash loans.

However, an attacker with sufficient collateral can still lower the redemption fee by inflating the system's collateral. They can borrow additional funds from a 3rd party protocol, e.g., via flashloan, and proceed to pay back the loan with their own funds. The attack requires at least two open Troves (A and B) and can be performed within 3 blocks.

Block 100:

1. The attacker deposits their collateral (e.g. 1000 wstETH) into *Trove A*`.

Block 101:

1. The attackers borrow 1000 wstETH from a 3rd party protocol and add it as collateral to *Trove B*.
2. They redeem the target trove, paying a lower redemption fee due to the inflated collateral balance.
3. They remove the collateral from *Trove A* and repay the loan in the same transaction.

Block 102:

1. They remove the collateral from *Trove B*.

Note that Arbitrum has a block time of 0.25 seconds, which means that the attack can be performed within a second and pay negligible interest for the borrowed amount.

Risk accepted:

Due to the block-level limits, the impact is limited. Hedgehog is aware of this issue and is accepting the risk.

5.8 Redemptions Without Base Rate Increase

Security | Low | Version 2 | Risk Accepted

CS-HOG-052

The Version 2 of the protocol removed an assertion check from function `TroveManager._updateRedemptionBaseRateFromRedemption`. This check previously ensured that the new base rate was always non-zero after a redemption:

```
function _updateRedemptionBaseRateFromRedemption(
    uint _WStETHDrawn
) internal returns (uint) {
    ...
    // HEDGEHOG UPDATES: Calculation the fraction now as a ratio of Collateral
    // that is about to get redeemed and a sum of collateral in active & default pools.

    uint redeemedBaseFeeLMAFraction = _WStETHDrawn
        .mul(DECIMAL_PRECISION)
        .div(activePool.getWStETH() + defaultPool.getWStETH());

    // Hedgehog Updates: Remove division by BETA
    uint newBaseRate = decayedRedemptionBaseRate.add(
        redeemedBaseFeeLMAFraction
    );

    // cap baseRate at a maximum of 100%
    newBaseRate = LiquityMath._min(newBaseRate, DECIMAL_PRECISION);
    // Hedgehog Updates: Remove assertion check to make sure first redemption
    // does not revert after the bootstrapping period if more then 10^18 WstETH
    // was transfer into the contract
    // assert(newBaseRate > 0); // Base rate is always non-zero after redemption
```

The change allows a redemption to occur without increasing the base rate of the protocol. This is possible when the fraction `redeemedBaseFeeLMAFraction` rounds to zero, which can occur when the amount of WStETH drawn is small compared to the total collateral in the system.

$$\_WStETHDrawn < \frac{totalCollateral}{10^{18}}$$

An attacker can exploit this by splitting a large redemption into multiple smaller redemptions to avoid increasing the base rate of the protocol.

Risk accepted:

Hedgehog is aware of this issue but has decided to keep the code unchanged.

5.9 Incorrect Code Comments

Correctness | Low | Version 1 | Code Partially Corrected

CS-HOG-035

1. The code comment for the variable `CommunityIssuance.HOGSupplyCap` states that it should be set to 32 million, however the distribution setters can freely assign any value to it.

2. The code comment in the contract `ActivePool` refers to the collateral token as `stWStETH` instead of `WStETH`.
3. In function `BorrowerOperations._adjustTrove()`, the comment `Use the unmodified _BaseFeeLMAChange here, ...` does not match the code which passes `_BaseFeeLMAChange - vars.BaseFeeLMAFee` to the internal function `_moveTokensAndWStETHfromAdjustment()`.
4. The comment for function `PriceFeed.fetchPrice()` refers to the Liquity oracles.
5. The max deviation allowed for two consecutive prices in `PriceFeed` is set to `17.6%`, however several code comments refer to other percentages such as `12.5%` or `50%`.
6. Similarly, the max deviation allowed between main and backup oracle is set to `5%`, however comments are not in line: `Return true if the relative price difference is <= 3%`.
7. The struct `Response` in the contract `PriceFeed` does not include a `success` flag, however comments refer to it: `... return a zero response with success = false`.
8. The code comments in the function `TroveManager._calcRedemptionFee()` imply that checks are now performed in the `BorrowerOperations` contract, but they are performed in `TroveManager.redeemCollateral()`.
9. In the function `TroveManager._redeemCollateralFromTrove`, the comment `Change WStETHLOT calculations formula from ...` does not match the code which calculates the `WStETHLot` as debt x price.
10. The code comments for function `TroveManager._updateRedemptionBaseRateFromRedemption()` refer to a face value rate of `(1 BaseFeeLMA:1 USD)` instead of `(1 BaseFeeLMA:1 Base fee moving average)`. Furthermore, in contrast to the code comments, the `WStETH` is not converted to `BaseFeeLMA`, but the share of the collateral is used to calculate the redemption fee.
11. The code comments of function `DefaultPool.increaseBalance()` state that the function can only be called by the active pool, but it can only be called by the `TroveManager`.
12. The comment above function `TroveManager._checkPotentialRecoveryMode()` refers to the price of the pair `WStETH:USD`, however the price is used is for the pair `BaseFeeLMA:ETH`.
13. The comment above function `TroveManager.getNormalLiquidationPrice()` describes another functionality.
14. The code comment above function `StabilityPool._getCompoundedStakeFromSnapshots()` refers to front ends which have been removed in Hedgehog Protocol.
15. The code comments in the function `TroveManager._getCappedOffsetVals()` describing the cappedCollPortion formula do not match the code.

Version 4:

16. The constant `BaseFeeLMA_GAS_COMPENSATION` is set to 300.000 * 10e18, however the code comments refer to 100.000 * 10e6.
17. The code comments above `BorrowerOperationsArb.setAddresses()` refer to the wrong variable name `lastWithdrawTimestamp`.
18. The code comments above function `_handleWithdrawlLimit` imply that the limit is `limit = old limit + 50% + ...`, however the limit is calculated as `limit = old limit + 50% * ...`.
19. The code comments above function `_handleWithdrawlLimit` say that "new limit is greater than or equal to 50% of the new total collateral". This is not possible for collateral values above the threshold.

Version 5:

16. Code comments in `PriceFeed._scalePriceByDigits()` state that the main oracle has 8 digits, but the code uses 18 digits.
17. The URL to EIP 2612 in `IERC2612` has been changed to `**eips.wStETHeum.org/**`.
18. The natspec comments above `FeesRouter._getPctRange` state that "In case the fee is less than 3% it's going to round to 5% anyway". However, if the fee is less than 1%, it will be rounded to 0%.

Code partially corrected:

The incorrect code comments in points 1-4 and 6-15 have been fixed in Version 2 and incorrect code comments in points 16-18 have been fixed in Version 5.

5.10 Known Issues From Liquity Are Present in Hedgehog

Design | Low | Version 1 | Risk Accepted

CS-HOG-038

Known issues in Liquity published in the Github advisories and repository's documentation are present in Hedgehog codebase.

Risk accepted:

Hedgehog is aware of the known problems and leaves them for further examination.

5.11 Lack of Documentation

Correctness | Low | Version 1 | Acknowledged

CS-HOG-021

The functionalities modified by Hedgehog are not properly documented. The documentation for new functionalities in FeesRouter, BaseFeeOracle, and TroveManager (`get**LiquidationPrice()`) are not complete.

Acknowledged:

Hedgehog has improved inline specifications in the last iterations of the codebase, but complete documentations will be provided in the future.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	5

- Collateral Surplus Is Stuck in the Contract Code Corrected
- Incorrect Price Used for Collateralization Ratio Code Corrected
- Redeemed Amount Does Not Account for Decimals Code Corrected
- Wrong Conversion Formula Used in _getCappedOffsetVals Code Corrected
- Wrong Decimals Returned by _computeCR Code Corrected

High-Severity Findings	11

- Liquidations Are Blocked From Withdrawal Limit Code Corrected
- Withdrawal Limit Does Not Track Collateral Code Corrected
- Incompatible Interface With BaseFeeOracleArb Code Corrected
- Gas Compensation Is Ignored in Trove Redemption Code Corrected
- Gas Compensation Is Not Accounted for Correctly When Closing a Trove Code Corrected
- Price Feed Returns Wrong Price Code Corrected
- Price Feed Stores and Returns Wrong Decimals Code Corrected
- Redemption Fees Can Be Lowered to Floor Value Specification Changed
- Redemption Rate Double Counts the Redemption Share Code Corrected
- Withdrawing wstETH Gains to Trove Reverts Code Corrected
- Wrong Conversion Rate Used in HintHelpers Code Corrected

Medium-Severity Findings	23

- Adversary Can Keep Withdrawal Limit Tiny Code Corrected
- Incorrect CommunityIssuance Configuration Can Break the StabilityPool Code Corrected
- Closing a Trove Does Not Update the Withdrawal Limit Code Corrected
- Double Counting of Full Redemptions in Withdrawal Limit Calculation Code Corrected
- Limit Can Exceed Active Collateral Code Corrected
- Liquidations Update Withdrawal Limit in an Inconsistent Way Code Corrected
- Withdrawal Limit Reset on Collateral Deposits Code Corrected
- Incorrect Timeout Value in PriceFeedArb Code Corrected
- Liquidation Price Has Wrong Decimals Code Corrected
- Locking of Troves Is Longer Than Specified Specification Changed
- Change of Issuance Curve Has Unexpected Side Effects Code Corrected
- Chosen Values for Gas Compensation and Minimum Debt Are Low Code Corrected
- Closing Troves Requires Borrowers Having Larger Balance Than Needed Code Corrected

- Distribution Functions in FeesRouter Use Wrong Configs Code Corrected
- Function _getUSDValue Computes Wrong Value Specification Changed
- Gas Compensation Not Accounted on Redemption Hints Code Corrected
- Inconsistent Definition of Redemption Share Code Corrected
- Mismatch of NICR Specifications With Implementation Code Corrected
- Price Feed Compares Timestamp to Blocknumber Code Corrected
- PriceFeed Does Not Check if Main Oracle Recovers Code Corrected
- Redemption Share Is Rounded to Zero Specification Changed
- Unusual Decimals Used for Values in StabilityPool Code Corrected
- Wrong Value Used to Calculate the Borrowing Rate With Decay Code Corrected

Low-Severity Findings	19

- Collateralization Ratio Is Rounded Down Code Corrected
- Adversary Can Slow the Recovery of Withdrawal Limit Code Corrected
- Outdated Specification for _handleWithdrawalLimit Code Corrected
- Precision Issue in Withdrawal Limit Calculation Code Corrected
- Unclear Specification Regarding Oracle Decimals Specification Changed
- Withdrawal Limit Function Has Discontinuities Code Corrected
- Magic Value for Expand Duration Code Corrected
- Missing Event When Changing Withdrawal Limit Code Corrected
- Withdrawal Threshold Can Be Circumvented by Splitting Transactions Code Corrected
- Unnecessary Limitation When Opening a Trove Code Corrected
- Event BorrowBaseRateUpdated Is Emitted Twice Code Corrected
- Excess Fee Distribution in FeesRouter Code Corrected
- Function _findPriceBelowMCR Can Be Improved Specification Changed
- Immutable Parameters Should Be Constants Code Corrected
- Incomplete Error Message Code Corrected
- Incorrect Validation of Repayments Code Corrected
- Initial Stake Rounds Down to Zero Code Corrected
- Missing Event When Increasing Balance Code Corrected
- Missing Sanity Checks Code Corrected

Informational Findings	10

- Misleading Variable Name in BorrowerOperationsArb Code Corrected
- Misleading Variable Name in LiquityMath Code Corrected
- Withdrawal Limit Does Not Take Collateral From Redistributions Into Account Code Corrected
- Remaining ToDos Code Corrected
- Gas Optimizations Code Corrected
- Incorrect Interfaces Code Corrected
- Misleading Variable Name in BaseFeeOracle Code Corrected

- Misleading Variable Name in FeesRouter Code Corrected
- Misleading Variable Name in TroveManager Code Corrected
- Vulnerable Dependency Code Corrected

6.1 Collateral Surplus Is Stuck in the Contract

Correctness Critical Version 1 Code Corrected

CS-HOG-001

In Hedgehog's code, the function `TroveManager.liquidateTroves()` function fails to call *increaseBalance* to update the wstETH balance of the collateral surplus pool.

Similarly, `TroveManager._redeemCloseTrove()` fails to update the wstETH balance when sending the surplus wstETH. Now, whenever a user calls `CollSurplusPool.claimColl()` , the call reverts on underflows of WStETH and the funds are stuck in the contract.

Code corrected:

The function `TroveManager.liquidateTroves()` has been revised to call `increaseBalance()` when sending the collateral to the surplus pool:

```
if (totals.totalCollSurplus > 0) {
    collSurplusPool.increaseBalance(totals.totalCollSurplus);
    contractsCache.activePool.sendWStETH(
        address(collSurplusPool),
        totals.totalCollSurplus
    );
}
```

Similarly, the function `_redeemCloseTrove()` has been updated to `increaseBalance()` when sending collateral from the active pool to the surplus pool.

6.2 Incorrect Price Used for Collateralization Ratio

Correctness Critical Version 1 Code Corrected

CS-HOG-002

The function `LiquityMath._computeCR` is crucial to compute the collateralization ratio (CR) of a trove or the system:

```
uint newCollRatio = _coll.mul(DECIMAL_PRECISION).div(_debt).div(_price);
```

The value of `_price` is retrieved from `PriceFeed` which returns the price for the pair `BaseFeeLMA:ETH`. However, the function `_computeCR()` is called with `_coll` representing a WStETH amount and `_debt` representing a `BaseFeeLMA` amount. Therefore, the conversion of collateral amount into debt token is incorrect.

Code Corrected:

The `PriceFeed` now returns the price for the asset pair `BaseFeeLMA:WstETH`. Note that decimals are not correct yet (see: Wrong decimals returned by _computeCR).

6.3 Redeemed Amount Does Not Account for Decimals

Correctness Critical Version 1 Code Corrected

CS-HOG-003

The function `TroveManager._redeemCollateralFromTrove()` calculates the amount of WstETH that can be redeemed using the following formula:

```
singleRedemption.WStETHLot = singleRedemption.BaseFeeLMALot.mul(_price);
```

BaseFeOracle returns the price for the token pair `ETH/BaseFeeLMA`, hence an incorrect price is used to convert `BaseFeeLMALot` into `WStETHLot`. Furthermore, the oracle returns prices with 1 decimal which is not accounted in the formula above, causing an error on the redeemed amount with an order of magnitude.

Version 2:

The codebase had been updated to use 18 decimals for the BaseFeeLMA token, and 18 decimals for the price. Thus, the formula:

```
singleRedemption.BaseFeeLMALot.mul(_price);
```

returns a value in 36 decimals instead of 18, hence computing a wrong result.

Code corrected:

The codebase has been updated to divide the result of the multiplication by 1e18 to account for the 18 decimals in the price.

6.4 Wrong Conversion Formula Used in _getCappedOffsetVals

Correctness Critical Version 1 Code Corrected

CS-HOG-004

The function `TroveManager._getCappedOffsetVals()` incorrectly converts a debt amount into collateral:

```
uint cappedCollPortion = _entireTroveDebt.mul(MCR).mul(_price).div(DECIMAL_PRECISION);
```

The `_price` retrieved from the `PriceFeed` returns the exchange ratio for the token pair `BaseFeeLMA:ETH` and is incorrectly used to convert `BaseFeeLMA` tokens into `WStETH`. Furthermore, the debt amount `_entireTroveDebt` is in the base fee token (6 decimals), while the price uses 1 decimal. Thus, the division by `1e18` is incorrect.

Version 2:

The `PriceFeed` has been updated to return the price for the token pair `BaseFeeLMA:WStETH`. Further the BaseFeeLMA token and the price now have 18 decimals.

Thus, the formula:

```
uint cappedCollPortion = _entireTroveDebt.mul(MCR).mul(_price).div(DECIMAL_PRECISION);
```

returns a value of 36 decimals instead of 18, hence computing a wrong collateral amount. Furthermore, the intermediate result `_entireTroveDebt.mul(MCR).mul(_price)` has 48 decimals and can overflow.

Code corrected:

The function `_getCappedOffsetVals` has been updated to divide the intermediary result by `DECIMAL_PRECISION` to normalize the result to 18 decimals:

```
// Changed the cappedCollPortion formula from [entireTroveDebt] * [MCR] / [price]  to => [entireTroveDebt] * [MCR] / [DECIMAL_PRECISION] * [price] / [DECIMAL_PRECISION]
uint cappedCollPortion = _entireTroveDebt
    .mul(MCR)
    .div(DECIMAL_PRECISION)
    .mul(_price)
    .div(DECIMAL_PRECISION);
```

6.5 Wrong Decimals Returned by _computeCR

Correctness | Critical | Version 1 | Code Corrected

CS-HOG-005

The function `LiquityMath._computeCR()` does not handle decimals correctly, therefore returning wrong collateralization ratios (CR):

```
uint newCollRatio = _coll.mul(DECIMAL_PRECISION).div(_debt).div(_price);
```

Note that `_coll` is in 18 decimals (WStETH), `_debt` is in 6 decimals (BaseFeeLMA), while `_price` has 1 decimal (BaseFeeOracle). Therefore, the computed value is in 29 decimals. This is a severe issue since the system parameters `MCR` and `CCR` are in 18 decimals. Therefore, even undercollateralized troves would pass checks for `MCR` and `CCR`.

Version 2:

The codebase had been updated to use 18 decimals for the BaseFeeLMA token, and 18 decimals for the price. Thus, the formula:

```
uint newCollRatio = _coll.mul(DECIMAL_PRECISION).div(_debt).div(_price);
```

returns a value with no decimals instead of 18, hence computing a wrong CR value.

Code corrected:

The codebase now multiplies by `DECIMAL_PRECISION` (10**18) before dividing by the price, hence the CR is now in 18 decimals.

6.6 Liquidations Are Blocked From Withdrawal Limit

Design High Version 4 Code Corrected

CS-HOG-059

The Hedgehog specifies in function `TroveManagerArb._sendGasCompensation()` that liquidations should not revert when the withdrawn collateral exceeds 80% of the current withdrawal limit.

However, the function then updates the withdrawal limit in `BorrowerOperationsArb._handleWithdrawalLimit()`:

```
(uint256 fullLimit, uint256 singleTxWithdrawable) = LiquityMath
    ._checkWithdrawlLimit(
        lastWithdrawlTimestamp,
        EXPAND_DURATION,
        unusedWithdrawlLimit,
        activePool.getWStETH()
    );

if (_withSingleTxLimit && singleTxWithdrawable < _collWithdrawal) {
    revert(
        "BO: Cannot withdraw more then 80% of withdrawble in one tx"
    );
}

// Update current unusedWithdrawlLimit
unusedWithdrawlLimit = fullLimit - _collWithdrawal;
```

If the collateral withdrawn exceeds the limit, the transaction will not revert with "BO: Cannot withdraw more than 80% of withdrawable in one tx" since `_withSingleTxLimit` is false during liquidations. However, since the collateral withdrawn exceeds the calculated value of *full_limit*, updating the unused withdrawal limit will cause an underflow and the transaction will revert. This effectively blocks all liquidations once the withdrawal limit is reached.

Code Corrected:

A ternary operator was added to prevent the arithmetic underflow:

```
unusedWithdrawalLimit = fullLimit > _collWithdrawal
    ? fullLimit - _collWithdrawal
    : 0;
```

Hence, the code has been fixed.

6.7 Withdrawal Limit Does Not Track Collateral

Design High Version 4 Code Corrected

CS-HOG-064

The withdrawal limit is expected to be smooth to make its behavior predictable for outside users. However, there are certain interactions that are not accounted for in the calculation of the withdrawal limit and leading to erratic behavior.

1. The withdrawal limit is equal to the collateral value up to the *WITHDRAWAL_LIMIT_THRESHOLD* and half of the collateral value or less afterward. The limit jumps on this threshold value.
    - If the active pool has 99 wstETH and the limit is 99, adding 2 wstETH causes the limit to become 50.5.
2. When the unused withdrawal limit allows pushing the collateral below the threshold, all collateral can be withdrawn.
    - For example, if the active pool has 100 wstETH and the limit is 50, withdrawing some collateral causes the limit to reset to the threshold (100).

$activePool.getWStETH() - WITHDRAWAL\_LIMIT\_THRESHOLD \leq unusedWithdrawalLimit$

Code corrected:

The code has been corrected to make withdrawal limits track the collateral more smoothly. However, the withdrawal limit function is not smooth close to the withdrawal limit: Withdrawal limit function has discontinuities.

6.8 Incompatible Interface With BaseFeeOracleArb

Correctness High Version 3 Code Corrected

CS-HOG-054

The contract `PriceFeedArb` imports the interface `IBaseFeeOracle` which declares the following function:

```
function getRoundData(uint256 _roundId) external view
    returns (uint256, int256, uint256, uint256, uint256);
```

However, the respective functions implemented in `BaseFeeOracleArb` uses a different input type (`uint80` instead of `uint256`), which results in a different function selector:

```
function getRoundData(uint80 _roundId) public view
    returns (uint80, int256, uint256, uint256, uint80);
```

Furthermore, the types of return values for both functions `getRoundData()` and `latestRoundData()` in `BaseFeeOracleArb` are different from the interface declaration.

Code corrected:

In Version 4 of the contract, the function signature of `getRoundData()` and `latestRoundData()` in `BaseFeeOracleArb` is updated to match the interface declaration.

6.9 Gas Compensation Is Ignored in Trove Redemption

Correctness High Version 1 Code Corrected

When calculating the amount of BaseFeeTokens required to redeem a Trove in `TroveManager._redeemCollateralFromTrove()`, the gas compensation reimbursed to the user is not deducted from the debt of the Trove.

If the redeemer provides a value for `_maxBaseFeeLMAamount` that is large enough to cover the full debt of a Trove, the Trove should be closed. However, since the gas compensation is not deducted from the debt, the new debt becomes zero and the redemption will be subsequently canceled.

```
function _redeemCollateralFromTrove(..., _maxBaseFeeLMAamount, ...) {
    singleRedemption.BaseFeeLMALot = LiquityMath._min(_maxBaseFeeLMAamount, Troves[_borrower].debt);
    ...
    uint newDebt = (Troves[_borrower].debt).sub(singleRedemption.BaseFeeLMALot);
    ...
    if (newDebt == BaseFeeLMA_GAS_COMPENSATION) {
    ... close trove ...
    else {
        if (
            newNICR != _partialRedemptionHintNICR || _getNetDebt(newDebt) < MIN_NET_DEBT
        ) {
            singleRedemption.cancelledPartial = true;
            return singleRedemption;
        }
        ... update trove ...
    }
}
```

The function `TroveManager.redeemCollateral()` then halts any additional redemptions, assuming that the last Trove was partially redeemed, and the redeemer's collateral is exhausted.

This issue can be exploited to cause a DoS attack against the redemption of multiple Troves in a single transaction, as redeeming the first one halts execution. It is still possible to partially redeem a Trove. A single Trove could be fully redeemed by setting `_maxBaseFeeLMAamount = Troves[_borrower].debt - BaseFeeLMA_GAS_COMPENSATION`.

Moreover, an attacker could exploit this by reducing their Trove to a lower debt value than the gas compensation. This would create a trove that is not profitable to liquidate. Note that this is not possible in the current system since `MIN_NET_DEBT = BaseFeeLMA_GAS_COMPENSATION = 0.1 BaseFeeLMAToken`.

Code corrected:

The function `TroveManager._redeemCollateralFromTrove()` has been updated to deduct the gas compensation from the debt of the Trove.

```
function _redeemCollateralFromTrove(..., _maxBaseFeeLMAamount, ...) {
    singleRedemption.BaseFeeLMALot = LiquityMath._min(_maxBaseFeeLMAamount, Troves[_borrower].debt.sub(BaseFeeLMA_GAS_COMPENSATION));
    ...
}
```

6.10 Gas Compensation Is Not Accounted for Correctly When Closing a Trove

Correctness **High** **Version 1** **Code Corrected**

CS-HOG-007

When opening a trove the system mints `_BaseFeeLMAAmount - BaseFeeLMA_GAS_COMPENSATION - BaseFeeLMAFee` to the user and records `_BaseFeeLMAAmount` as trove's debt. When closing the trove, the user must burn

`debt - BaseFeeLMA_GAS_COMPENSATION` from their balance and `BaseFeeLMA_GAS_COMPENSATION` is burned from the GasPool.

After opening and closing a trove the User, GasPool and FeesRouter (their beneficiary) should have the following net balances of BaseFeeLMA tokens:

1. User: Initial_Balance - BaseFeeLMAFee
2. GasPool: BaseFeeLMA_GAS_COMPENSATION - BaseFeeLMA_GAS_COMPENSATION = 0
3. FeesRouter: BaseFeeLMAFee

However, the function `closeTrove()` burns the gas compensation from both the user and the gas pool address:

```
_repayBaseFeeLMA(
    activePoolCached,
    baseFeeLMATokenCached,
    msg.sender,
    debt
);
_repayBaseFeeLMA(
    activePoolCached,
    baseFeeLMATokenCached,
    gasPoolAddress,
    BaseFeeLMA_GAS_COMPENSATION
);
```

This violates the specifications that gas compensation is refunded to borrowers when closing a trove. Furthermore, the accounting of active pool is compromised as its debt is decreased with `debt + BaseFeeLMA_GAS_COMPENSATION` instead of `debt`.

Code corrected:

The accounting of gas compensation has been revised in Version 2. When opening a trove, the system records the total amount (minted amount + fee + gas compensation) as a debt of a user. When closing the trove, the gas compensation is withdrawn from the gas pool, while the rest of the debt (minted amount + fee) is withdrawn from the user.

6.11 Price Feed Returns Wrong Price

Correctness | High | Version 1 | Code Corrected

CS-HOG-008

The Function `PriceFeed.fetchPrice()` stores/returns the Main Oracle response when it should return the Backup Oracle response in four cases:

- Main Oracle is untrusted (Case 2):

```
// Otherwise, use Backup price
return _storeGoodPrice(mainOracleResponse, decimals);
```

- Main Oracle is frozen (Case 4):

```
// If Backup is working, return Backup current price
return _storeGoodPrice(mainOracleResponse, decimals);
```

- Main Oracle response is frozen and Backup Oracle is working (Case 4):

```
// if Main Oracle is frozen and Backup is working, keep using Backup (no status change)
return _storeGoodPrice(mainOracleResponse, decimals);
```

- Main Oracle response is live after being previously frozen, but the response is not within 5% of Backup Oracle response (Case 4):

```
// Otherwise if Main Oracle is live but price not within 5% of Backup, distrust Main Oracle, and return Backup price
_changeStatus(Status.usingBackupMainUntrusted);
return _storeGoodPrice(mainOracleResponse, decimals);
```

In these cases, the value is either incorrect, zero or outdated. An incorrect value can result in liquidations not getting performed when they should or health positions getting liquidated. If the value is zero the CR calculation that is performed on any user operations will fail, since it divides by zero (CR = coll / debt / price).

Code corrected:

The function `PriceFeed.fetchPrice()` now returns the Backup Oracle response in the four cases mentioned above.

6.12 Price Feed Stores and Returns Wrong Decimals

Correctness | High | Version 1 | Code Corrected

CS-HOG-009

The Function `PriceFeed.fetchPrice()` stores/returns the price of the Backup Oracle with the decimals of the Main Oracle when backup is live, and the oracles responses differ too much in price:

```
_changeStatus(Status.usingBackupMainUntrusted);
return _storeGoodPrice(backupOracleResponse, decimals);
```

Similarly, the function calculates the price deviation of the current and previous Backup Oracle response with the decimals of the Main Oracle in two cases:

```
// --- CASE 2: The system fetched last price from Backup ---
if (status == Status.usingBackupMainUntrusted) {
    if (
        _priceChangeAboveMax(
            backupOracleResponse,
            prevBackupOracleResponse,
            decimals
        )
    ) {
```

```
// If Backup is broken, both oracles are untrusted, and return last good price
if (
    _backupOracleIsBroken(backupOracleResponse) ||
    _priceChangeAboveMax(
        backupOracleResponse,
        prevBackupOracleResponse,
        decimals
```

```
        )
    )
```

Having two oracles with different decimals has severe consequences for the system as the stored prices have a large error.

Code corrected:

The function `fetchPrice()` has been revised to use the correct decimals in the code parts listed above.

6.13 Redemption Fees Can Be Lowered to Floor Value

Design | High | Version 1 | Specification Changed

CS-HOG-010

In Hedgehog's code, the redemption fee is calculated based on the proportion of collateral redeemed relative to the total collateral locked in the system.

$$RedRate = RedFloor + RedBaseRate * MinuteDecayFactor^{Minutes} + RedemptionEth/Collateral$$

An attacker can exploit this by inflating the system's collateral before redeeming a trove, thereby reducing their redemption fee. The collateral can be inflated by either adjusting a trove's collateral or by adding a new trove with a large amount of collateral. Here's a step-by-step scenario with an attacker that has an open Trove:

1. The attackers borrow a large amount of wstETH in a flash loan and add it as collateral to their Trove.
2. They redeem the target trove, paying a lower redemption fee due to the inflated collateral balance.
3. They remove the collateral from the trove and repay the flash loan.

Alternatively, an attacker can create a trove with the minimum debt and a large amount of collateral, execute Steps 1-3, and then repay their debt. With a large enough flash loan, an attacker will only pay redemption rate close to the floor rate (set to 0.5% in Hedgehog's code). The low redemption fee makes it likely that all collateral is exhausted when the base fee is underpriced in the secondary market.

Specification changed:

A new mechanism was implemented in Version 2 to forbid adding collateral to a trove via flash loans. The contract `BorrowerOperations` now limits the number of operations that modify a trove (such as open, adjust, or close) to at most once per block. Hence, it is not feasible anymore to add collateral to a trove and withdraw in the same transaction (required in case of flash loans).

Note that a variant of this attack (Attacker with sufficient funds can lower redemption fees) is still possible in Version 2.

6.14 Redemption Rate Double Counts the Redemption Share

Correctness High Version 1 Code Corrected

CS-HOG-011

The function `TroverManger.redeemCollateral()` double counts the redemption share (redemptionEth / Collateral) when calculating the redemption fees. First, it is added to the redemptionBaseRate in `_updateRedemptionBaseRateFromRedemption` and then it is added to the redemption rate in `_getRedemptionFee`:

```
function redeemCollateral(..) {
    ...
    _updateRedemptionBaseRateFromRedemption(totals.totalWStETHDrawn);
    // Calculate the WStETH fee
    totals.WStETHFee = _getRedemptionFee(totals.totalWStETHDrawn);
    ...
}
```

As fees may not exceed 100%, double counting of fees will block redemptions of more than 50% of collateral and make redemptions considerably more expensive and thus weaken the lower peg of the base fee token.

Code corrected:

The call path triggered by function `_getRedemptionFee()` has been refactored to avoid double counting of redemption share when computing the redemption fee:

```
function _getRedemptionFee(uint _WStETHDrawn) internal view returns (uint) {
    return _calcRedemptionFee(getRedemptionRate(), _WStETHDrawn);
}
```

The function `getRedemptionRate` computes the current fee rate and now does not depend on the redemption share.

6.15 Withdrawing wstETH Gains to Trove Reverts

Correctness High Version 1 Code Corrected

CS-HOG-012

The Function `StabilityPool.withdrawWStETHGainToTrove()` calls into `borrowerOperations.moveWStETHGainToTrove()` to pull collateral from the stability pool. However, since the `StabilityPool` does not provide any allowance to `BorrowerOperations`, the call always reverts. That will block all users from withdrawing their wstETH gains to their Trove.

Code corrected:

The function `StabilityPool.withdrawWStETHGainToTrove()` now approves the `BorrowerOperations` contract.

```
function withdrawWStETHGainToTrove(
    ...
    WStETHToken.approve(address(borrowerOperations), depositorWStETHGain);
    borrowerOperations.moveWStETHGainToTrove(
        msg.sender,
        _upperHint,
        _lowerHint,
        depositorWStETHGain
    );
}
```

6.16 Wrong Conversion Rate Used in HintHelpers

Correctness | High | Version 1 | Code Corrected

CS-HOG-013

The function `HintHelpers.getRedemptionHints()` incorrectly converts a debt amount into collateral:

```
uint newColl = WStETH.sub(maxRedeemableBaseFeeLMA.mul(_price));
```

The debt amount `maxRedeemableBaseFeeLMA` is in the base fee token (6 decimals), the collateral is in `WStETH` token (18 decimals), while `_price` stores the conversion rate for the pair `BaseFeeLMA:ETH` in 1 decimal. Thus, the conversion is incorrect.

Version 2:

The codebase had been updated to use 18 decimals for the BaseFeeLMA token, and 18 decimals for the price. Thus, the formula:

```
maxRedeemableBaseFeeLMA.mul(_price)
```

returns a value of 36 decimals instead of 18, hence computing a wrong collateral amount.

Code corrected:

The codebase has been updated to divide the intermediate value `maxRedeemableBaseFeeLMA.mul(_price)` by `DECIMAL_PRECISION`, hence the result has 18 decimals.

```
  uint newColl = WStETH.sub(
    maxRedeemableBaseFeeLMA.mul(_price).div(
        DECIMAL_PRECISION
    )
);
```

6.17 Adversary Can Keep Withdrawal Limit Tiny

Security Medium Version 5 Code Corrected

CS-HOG-072

One system assumption is that once the `ActivePool` holds a significant amount of funds, then the decay of the withdrawal limit will be too fast that an attacker could keep it small over a longer period of time without investing significant funds.

Below we explain why this might not be true. We assume that the `ActivePool` holds 20,000 WstETH and that the attacker holds 1 WstETH. The attacker opens two troves and initially deposits the 1 WstETH into the first one.

Each block, the attacker does the following:

- Withdraws 1 WstETH from the trove it is currently in
- Deposits 1 WstETH into the other trove

Hence, every two seconds (Base Chain Block Interval), the withdrawal limit will:

- Compute the decay accordingly to the formula
- Decrease `unusedWithdrawalLimit` by 1 WstETH due to the withdrawal
- Increase `unusedWithdrawalLimit` by 0.5 WstETH due to the deposit

Therefore, the question is, whether the decay will be larger than 0.5 WstETH. The maximum decay in this situation is:

`(totalCollBasedLimit - _unusedWithdrawalLimit) * percentageToGet`

= (10,005 WstETH - 0 WstETH) * (2 seconds / 720 minutes)

< 0.5 WstETH

(Note that technically the used attack amounts would be slightly different to not cause any reverts, but this has been omitted for readability.)

As the decay is smaller than the decrease introduced by the attacker, the attacker needs just 1/20,000 of the `ActivePool` funds to keep the withdrawal limit slightly above zero. The attacker only pays for transaction costs, which are relatively small on Base chain.

Code corrected:

In Version 6, the withdrawal limits have been removed resolving the griefing attack vector described.

6.18 Incorrect CommunityIssuance Configuration Can Break the StabilityPool

Security Medium Version 5 Code Corrected

CS-HOG-073

In the `CommunityIssuance` contract three different parameters can be set by a privileged role that control the issuance of `HOG` tokens. The `StabilityPool` calls the `CommunityIssuance` contract to calculate issuance and retrieve `HOG` tokens. The following could theoretically happen:

1. Inside the `CommunityIssuance` an incorrect value is set, e.g. `setHOGSupplyCap` is set to a value ten times higher than the actual supply of `HOG`.

2. An action inside the `StabilityPool`, like provide or withdraw, causes the `StabilityPool` to call `issueHOG` where the incorrect issuance will be calculated. The `StabilityPool` will calculate the `marginalHOGGain` and thereby remember how much `HOG` each user is entitled to. Even if the error is recognized now, the values are already stored inside the `StabilityPool`.
3. Whenever users use the provide or withdraw actions of the `StabilityPool` these precomputed `HOG` tokens are actually requested using `CommunityIssuance.sendHOG`.
4. Soon, no more `HOG` are inside the `CommunityIssuance` and the `sendHOG` command reverts.
5. From now on, all provide and withdraw operations on the `StabilityPool` will revert, and the funds are stuck inside the `StabilityPool`.

Code corrected:

The three parameters `HOGSupplyCap`, ``ISSUANCE_FACTOR` and `totalHOGIssued` in `CommunityIssuance` contract must now be modified in a two-step approach. For example the `HOGSupplyCap` must be be modified by first calling `proposeHOGSupplyCap`, and the change becomes effective after `acceptNewHOGSupplyCap` is called. The privileged role modifying issuance (*DISTRIBUTION_SETTER*) is expected to verify that the parameters are correct before calling `accept*`. Note, that this role can call the `propose` as well as `accept` functions. Therefore, the role could still negatively impact the system if it becomes malicious.

6.19 Closing a Trove Does Not Update the Withdrawal Limit

Design | Medium | Version 4 | Code Corrected

CS-HOG-060

The function `TroveManager.closeTrove()` withdraws collateral from the system. However, it does not update the withdrawal limit accordingly. As a result, the withdrawal limit can be trivially circumvented. Furthermore, one can inflate the limit by opening new troves, and closing them in the next block.

Code Corrected:

In Version 5 of the protocol, the function `TroveManager.closeTrove()` has been updated to update the withdrawal limit accordingly.

6.20 Double Counting of Full Redemptions in Withdrawal Limit Calculation

Design | Medium | Version 4 | Code Corrected

CS-HOG-061

The function `TroveManagerArb.redeemCollateral()` incorrectly double counts the collateral withdrawn by full redemptions when calculating the withdrawal limit.

First, the amount of collateral of the (fully) redeemed trove is accounted towards the withdrawal limit in `_redeemCloseTrove`. Second the collateral of all troves redeemed is collected in the summary variable `totalWStETHDrawn` that is then passed to function `handleWithdrawlLimit` a second time.

```
function redeemCollateral(uint _WStETHDrawn) public {

    IBorrowerOperations(borrowerOperationsAddress).handleWithdrawlLimit(
        totals.totalWStETHDrawn,
        true
    );
```

As a result, the collateral withdrawn for full redemptions is overestimated by a factor of 2, leading to lower withdrawal limits than intended.

Code corrected:

In Version 5 the withdrawal limit is only updated once, at the end of the function `redeemCollateral`.

6.21 Limit Can Exceed Active Collateral

Design | Medium | Version 4 | Code Corrected

CS-HOG-062

In the function `BorrowerOperationsArb._handleWithdrawalLimit()`, the unused withdrawal limit is set to the collateral in the active pool if the active pool holds less collateral than the withdrawal limit threshold.

```
if (activePool.getWStETH() > WITHDRAWL_LIMIT_THRESHOLD) {
...
} else {
    unusedWithdrawlLimit = activePool.getWStETH();
}
```

However, when removing collateral via `_adjustTrove`, the wstETH balance is only updated (in `_moveTokensAndWStETHfromAdjustment()`) after the withdrawal limit is already set. This means that the `unusedWithdrawalLimit` will be set to the collateral in the active pool prior to the withdrawal and hereby exceeding the amount of collateral after the withdrawal has been made. The same issue can be found in the calls to function `BorrowerOperationsArb.handleWithdrawlLimit()` in the functions `redeemCollateral()` and `_redeemCloseTrove()` in `TroveManagerArb`.

These discrepancies can accumulate over time, severely restricting the effectiveness of withdrawal limits.

Code corrected:

In Version 5 the logic of updating the unused withdrawal limit has been completely revamped.

1. If the active pool balance is below the withdrawal limit threshold, the function `LiquityMath._checkWithdrawalLimit()` returns the current collateral in the active pool.

```
function _checkWithdrawalLimit(
    ...
) internal view returns (uint256 fullLimit, uint256 singleTxWithdrawable) {
// If coll in the system is greater than the threshold - we check if user may withdraw the desired amount
// Otherwise they are free to withdraw whole amount
if (_currentTotalColl <= WITHDRAWAL_LIMIT_THRESHOLD) {
    return (_currentTotalColl, _currentTotalColl);
}
```

2. The unused withdrawal limit is then set to the `fullLimit` minus the collateral to be withdrawn.

```
function _handleWithdrawalLimit(
    uint256 _collWithdrawal,
    bool _isLiquidation
) internal {
    (uint256 fullLimit, uint256 singleTxWithdrawable) = LiquityMath
        ._checkWithdrawalLimit(
            lastWithdrawalTimestamp,
            EXPAND_DURATION,
            unusedWithdrawalLimit,
            activePool.getWStETH()
        );

    ...
    // Update current unusedWithdrawalLimit
    unusedWithdrawalLimit = fullLimit > _collWithdrawal
        ? fullLimit - _collWithdrawal
        : 0;

    ...
}
```

3. Functions `redeemCollateral()`, `_adjustTrove()`, and other related functions always call `_handleWithdrawalLimit` at the end, ensuring that the total collateral used in the calculation does not include the collateral withdrawn from the active pool.

In combination, the changes ensure that the unused withdrawal limit is below the amount of collateral in the active pool.

6.22 Liquidations Update Withdrawal Limit in an Inconsistent Way

Design | Medium | Version 4 | Code Corrected

CS-HOG-063

On a liquidation collateral is moved from the active pool:

1. to the Stability Pool to offset debt and collateral
2. to the Default Pool to redistribute debt and collateral
3. to the CollSurplusPool to store surplus collateral (optional)
4. to the caller as gas compensation

The withdrawal limit is updated on point 4 but not in the other cases.

Code corrected:

In Version 5, the withdrawal limit is updated once at the end of the functions `batchLiquidateTroves` and `liquidateTroves` with the total amount of liquidated collateral (including the gas compensation given to the caller).

6.23 Withdrawal Limit Reset on Collateral Deposits

Design | Medium | Version 4 | Code Corrected

CS-HOG-065

Version 4 of the protocol introduced withdrawal limits. According to the design, each new collateral deposit should increase the withdrawal limit by 50% of the deposit amount. However, the function `BorrowerOperationsArb._updateWithdrawlLimitFromCollIncrease()` resets the withdrawal limit to 50% of the (new) active collateral:

```
function _updateWithdrawlLimitFromCollIncrease(...) internal {
    uint256 newColl = _previousColl + _collIncrease;

    uint256 newLimit = (_previousColl / 2) + (_collIncrease / 2); // equivalent to newColl / 2
    if (newLimit >= _previousColl) {
        ...
    }

    unusedWithdrawlLimit = newLimit;
}
```

Therefore, any minimal deposit nullifies the effect of previous withdrawals and resets the limit.

Code corrected:

In Version 5, the function `_updateWithdrawlLimitFromCollIncrease` has been removed, and instead `BorrowerOperations._activePoolAddColl()` increases the withdrawal limit by half of the deposit amount sent to the active pool.

```
function _activePoolAddColl(
    IActivePool _activePool,
    uint _amount
) internal {
    WStETHToken.safeTransferFrom(msg.sender, address(_activePool), _amount);
    activePool.increaseBalance(_amount);

    // Update withdrawal Limit from collateral addition.
    unusedWithdrawalLimit = unusedWithdrawalLimit + _amount / 2;
}
```

6.24 Incorrect Timeout Value in PriceFeedArb

Correctness | Medium | Version 3 | Code Corrected

CS-HOG-055

The contract `PriceFeedArb` operates on Arbitrum block numbers and the timeout for fresh prices is set to 69 blocks, roughly 17 seconds. The timeout is lower than intended, therefore the contract `PriceFeedArb` would consider prices returned by `BaseFeeOracleArb` as stale.

Code corrected:

In Version 4 the timeout was updated to 1600 blocks or around 400 seconds at the current block time on Arbitrum. Security considerations of a change in Arbitrum block time are discussed in another issue, see Dependency on current block time.

6.25 Liquidation Price Has Wrong Decimals

Design Medium Version 2 Code Corrected

CS-HOG-050

The function `LiquityMath._findPriceBelowMCR()` calculates the liquidation price of a Trove from the Minimum Collateral Ratio (MCR), the collateral and debt of the Trove as:

```
function _findPriceBelowMCR(
    uint256 _coll,
    uint256 _debt,
    uint _mcr
) internal pure returns (uint256 price) {
    // Finds an exact price at which CR becomes MCR. Liquidation does not happen in
    // the event of them being equal, hence we add 1 to it to find closest liquidation price
    price = ((_coll * DECIMAL_PRECISION) / _debt / _mcr) + 1;
}
```

Note that *_coll* and *_debt* have 18 decimals precision, so the result of the intermediary division *_coll * DECIMAL_PRECISION / _debt* is a number with 18 decimals. The *mcr* has 18 decimals, so the result of the division *_coll * DECIMAL_PRECISION / _debt / _mcr* has 0 decimals instead of the expected 18.

Code corrected:

The codebase now multiplies the intermediary result by 1e18 before dividing by *_mcr*, so the result has 18 decimals.

6.26 Locking of Troves Is Longer Than Specified

Correctness Medium Version 2 Specification Changed

CS-HOG-051

The contract `BorrowerOperations` implements a new check `_checkAndSetUpdateBlock()` in Version 2 to restrict operations that modify a trove more than once in a single block. The specification of the function is:

```
// HedgehogUpdates: new private function, that checks if there was a transaction
    with a trove in the current block
```

However, the function uses `block.number` to check if a trove is being modified more than once in a block:

```
function _checkAndSetUpdateBlock(address _borrower) private {
    if (troveManager.getTroveUpdateBlock(_borrower) == block.number) {
        revert TroveAdjustedThisBlock();
    }
    ...
}
```

We would like to highlight that `block.number` in Arbitrum returns the estimated block in Layer-1 (Ethereum mainnet). Thus, the check above prevents transactions that modify a trove for more than one block.

Specification changed:

Contract `BorrowerOperations` is no longer in scope in Version 4. A new contract `BorrowerOperationsArb` has been added that will be deployed in Arbitrum and resolves the issue above by using `arbsys.arbBlockNumber()` to get block numbers.

6.27 Change of Issuance Curve Has Unexpected Side Effects

Design | Medium | Version 1 | Code Corrected

CS-HOG-014

The issuance computed by `CommunityIssuance.issueHOG()` is the difference between the current point on the issuance curve and the point at the last update (totalHOGIssued).

```
uint latestTotalHOGIssued = HOGSupplyCap
    .mul(_getCumulativeIssuanceFraction())
    .div(DECIMAL_PRECISION);
uint issuance = latestTotalHOGIssued.sub(totalHOGIssued);

totalHOGIssued = latestTotalHOGIssued;
```

An address with the `DISTRIBUTION_SETTER` role can modify the issuance curve by changing their convergence limit (HOGSupplyCap) or their rate of convergence (ISSUANCE_FACTOR) with the function `CommunityIssuance.setHOGSupplyCap()` or `CommunityIssuance.setISSUANCE_FACTOR()`, respectively. There is no function to modify the point of the last update (totalHogIssued).

If the value of the new curve is lower than the previous point, then the subtraction (latestTotalHOGIssued - totalHOGIssued) will underflow. Although only whitelisted accounts can set these parameters, misconfigurations are possible. If such misconfigurations happen, they cause all liquidations and calls to provide or withdraw stake from the stability pool to revert.

If the new curve is higher than the old curve at the current point in time, all additional issuance will be consumed in the next call to *issueHog*.

Code corrected:

The `CommunityIssuance.issueHOG()` function now returns 0 instead of causing an underflow when the new curve is lower than before. This prevents failures in liquidations and staking:

```
uint issuance = latestTotalHOGIssued > totalHOGIssued
    ? latestTotalHOGIssued.sub(totalHOGIssued)
    : 0;
```

A new function, `CommunityIssuance.setTotalHOGIssued()` has been added, to modify the previous point on the curve (totalHOGIssued).

Note that when increasing the issuance, the function *setTotalHogIssued()* must be called called first (or calls to *setHOGSupplyCap*, *setISSUANCE_FACTOR*, and *setTotalHogIssued* are batched). Otherwise, an *issueHog()* call might use all the extra issuance before *setTotalHogIssued()* is called.

6.28 Chosen Values for Gas Compensation and Minimum Debt Are Low

Design | Medium | Version 1 | Code Corrected

CS-HOG-015

The parameters for gas compensation and minimum net debt are set in the file `deployConfig.ts`:

```
export const deployConfig: DeploymentConfig = {
    ...
    gasComp: "100000",
    minNetDebt: "100000",
    ...
};
```

Both `gasComp` and `minNetDebt` represent amounts in the debt token, `BaseFeeLMAToken`, which uses 6 decimals. Therefore, gas compensation and minimum net debt are set to low amounts, corresponding to `0.1 BaseFeeLMA`.

Assuming a liquidation consumes 500,000 gas, the gas price on Arbitrum is about 1/5 of the base fee, and 1e6 `BaseFeeLMAToken` are worth 1 base fee (6 decimals), the cost of the liquidation expressed in `BaseFeeLMAToken` is:

$$gasCostInBaseFeeTokens = (500.000/5) * 1e6 = 100.000 * 1e6$$

A low minimum net debt enables gas griefing attacks by lowering the costs to create a large number of troves. Similarly, the gas compensation is very low, and it makes liquidations less attractive, hence increasing the risks of unhealthy troves. Troves with minimum net debt are not worth liquidating as the reward is lower than the gas costs of executing a liquidation.

Version 2:

The codebase had been updated to use 18 decimals for the BaseFeeLMA token. Thus, the constant variables set in the contract HedgehogBase represent very small amounts (less than 1 BaseFeeLMA token):

```
// HEDGEHOG UPDATES: Decreased to 100k wei
// Amount of BaseFeeLMA to be locked in gas pool on opening troves
uint public constant BaseFeeLMA_GAS_COMPENSATION = 100000;

// HEDGEHOG UPDATES: Decreased to 350000000 BFE
// Minimum amount of net BaseFeeLMA debt a trove must have
uint public constant MIN_NET_DEBT = 350000000;
```

Code corrected:

In Version 3 the BaseFeeLMA_GAS_COMPENSATION has been updated to 100.000 * 1e18. Hedgehog considers this amount as fair compensation although it might not cover always the gas costs of liquidations:

```
The gas compensation amount is based on the ratio between L2 and Ethereum gas price.
The Hedgehog liquidation transaction consumes app. 417-673K of L2 gas. The L2/L1 gas
price ratio range was estimated between 1:8 and 1:3 (Optimism L2, September 2023).
The present peak values (March 16, 2024) are app. 1:3.75. The compensated value is
then 52K-84K (1:8 ratio) or 139K-224K (1:3 ratio) L1 gas units, equivalent to the same
number of BaseFee token fractions. The 100K value is taken as a moderate rounded value
of the required compensation, given that the price ratio is often much lower than the
estimated peaks
```

The MIN_NET_DEBT has been changed to 50.000.000 * 1e18 or around 20.000 USD at a base fee of 100 Gwei and 4000 USD/ETH (50 mio * 100e-9 * 4000 = 20.000 USD). The minimum debt for a trove scales linearly with the base fee. Hence, a decrease of base fee at 1 Gwei, results in a minimum debt of 200 USD (50 mio * 100e-9 * 4000 = 200 USD). Therefore, the griefing attack that inject troves in the linked list to make user transactions such as openTrove() or redeemCollateral() revert, are more likely when base fee is low.

In Version 4 the BaseFeeLMA_GAS_COMPENSATION has been increased to 300.000 * 1e18 and the MIN_NET_DEBT has been increased to 100.000.000 * 1e18.

6.29 Closing Troves Requires Borrowers Having Larger Balance Than Needed

Correctness Medium Version 1 Code Corrected

CS-HOG-016

When opening a new trove, its debt is set to the gross amount of debt which includes the borrowing fee and the gas compensation:

```
function openTrove(..., _BaseFeeLMAAmount, ...) {
    ...
    vars.netDebt = _BaseFeeLMAAmount;
    ...
    vars.compositeDebt = vars.netDebt;
    ...
    contractsCache.troveManager.increaseTroveDebt(msg.sender, vars.compositeDebt);
    ...
}
```

The gas compensation is held by the gas pool address and should be refunded to users when closing a trove.

However, the function closeTrove implements a check that requires borrowers to have enough balance to repay the whole debt (including gas compensation):

```
uint debt = troveManagerCached.getTroveDebt(msg.sender);
_requireSufficientBaseFeeLMABalance(
    baseFeeLMATokenCached,
    msg.sender,
    debt // Hedgehog Updates: do not deduct gas comp anymore
);
```

Code corrected:

The function `closeTrove` now deducts the gas compensation from the debt before checking the user's balance:

```
uint debt = troveManagerCached.getTroveDebt(msg.sender);
_requireSufficientBaseFeeLMABalance(
    baseFeeLMATokenCached,
    msg.sender,
    debt.sub(BaseFeeLMA_GAS_COMPENSATION)
);
```

6.30 Distribution Functions in FeesRouter Use Wrong Configs

Design | Medium | Version 1 | Code Corrected

CS-HOG-017

Functions `distributeDebtFee()` and `distributeCollFee()` in the contracts `FeesRouter` use a wrong formula when retrieving a fee configuration:

```
FeeConfig memory config = FeeConfigs[(((_fee * 100) / _debt) % 5) * 5];
```

The intermediate result `((_fee * 100) / _debt) % 5)` in the formula above returns a value between `0` and `4`. Multiplying this intermediate result with `5` can produce five possibilities for the fee percentages: `0, 5, 10, 15, 20`.

This behavior is counter intuitive as for a fee percentage of 6%, 11% or 16%, the config corresponding to the 5% range is used. While for a fee percentage of 7%, 12%, and 17%, the config for 10% is used.

Code corrected:

A new internal function `_getPctRange` has been added in Version 2 that computes the closest multiplier of 5 given an amount of debt and the respective fee.

6.31 Function _getUSDValue Computes Wrong Value

Correctness | Medium | Version 1 | Specification Changed

CS-HOG-018

The function `BorrowerOperations._getUSDValue` uses the following formula to compute the return value:

```
function _getUSDValue(uint _coll, uint _price) internal pure returns (uint) {
    uint usdValue = _price.mul(_coll).div(DECIMAL_PRECISION);
```

```
    return usdValue;
}
```

Collateral amount `_coll` is in `WStETH` while `_price` is for the pair `BaseFeeLMA:ETH`, hence the returned value does not represent the value of collateral in USD. This function is unused in the current version of the codebase.

Specification changed:

The function `BorrowerOperations._getUSDValue()` function has been deleted from the codebase.

6.32 Gas Compensation Not Accounted on Redemption Hints

Correctness Medium Version 1 Code Corrected

CS-HOG-019

The function `HintHelpers.getRedemptionHints()` does not account correctly for the gas compensation linked to a trove. The net debt of trove is computed as follows:

```
uint netBaseFeeLMADebt = _getNetDebt(troveManager.getTroveDebt(currentTroveuser))
    .add(troveManager.getPendingBaseFeeLMADebtReward(currentTroveuser));
```

Differently from Liquity, function `_getNetDebt()` does not subtract the gas compensation, hence it remains included in the amount `netBaseFeeLMADebt`.

In case the trove should be closed during the redemption, the gas compensation is paid by the redeemer instead of refunded from the gas pool:

```
if (netBaseFeeLMADebt > remainingBaseFeeLMA) {
    ...
} else {
    remainingBaseFeeLMA = remainingBaseFeeLMA.sub(
        netBaseFeeLMADebt
    );
}
```

Code corrected:

The function `HedgehogBase._getNetDebt()` has been revised in Version 2 to subtract the gas compensation from a debt.

6.33 Inconsistent Definition of Redemption Share

Correctness Medium Version 1 Code Corrected

CS-HOG-020

The `TroveManager` contract in the codebase defines the redemption share inconsistently across two of its functions.

In the function `TroveManager._calcRedemptionRate()`, the redemption share is defined as the division of redeemed collateral by the sum of collateral in the active pool and the default pool.

$$redemptionShare = RedemptionEth/(CollateralinActivePool + CollateralinDefaultPool)$$

However, in the function `TroveManager._updateRedemptionBaseRateFromRedemption()`, the redemption share is calculated differently. Here, it is defined as the proportion of the redeemed collateral to the collateral in the active pool only:

$$redemptionShare = RedemptionEth/CollateralinActivePool$$

The NatSpec comments above the function `TroveManager._calcRedemptionRate()` imply the identical alternate formula that ignores the collateral in the default pool.

This inconsistency in the redemption share definition could lead to external parties misunderstanding the redemption mechanism and the calculation of the redemption rate. It is recommended to harmonize the definition of the redemption share in both functions and to update the NatSpec comments accordingly.

Code corrected:

The function `TroveManager._updateRedemptionBaseRateFromRedemption()` was updated to calculate the redemption share as the division of the redeemed collateral by the sum of collateral in the active pool and the default pool.

$$redemptionShare = RedemptionEth/(CollateralinActivePool + CollateralinDefaultPool)$$

The NatSpec comments above the function were also updated to reflect the corrected formula.

The function `TroveManager._calcRedemptionRate()` no longer adds the redemption share to the redemption rates, since the share would be double counted (see: Redemption rate double counts the redemption share).

6.34 Mismatch of NICR Specifications With Implementation

Correctness **Medium** **Version 1** **Code Corrected**

CS-HOG-022

The NatSpec description of the constant `NICR_PRECISION` in library `LiquityMath` states:

```
This value of 1e20 is chosen for safety: the NICR will only overflow for numerator > ~1e39 WStETH
```

The implementation of function `_computeNominalCR` computes the nominal individual collateralization ratio as follows:

```
return _coll.mul(NICR_PRECISION).div(_debt);
```

The amount `_coll` is in 18 decimals (WStETH), while `_debt` is in 6 decimals (BaseFeeLMA), hence the result is in 30 decimals. This conflicts with the specification of `NICR_PRECISION`.

Code corrected:

The debt token (BaseFeeLMA) has been changed to use 18 decimals. Therefore `_debt` and the result of the multiplication are in 18 decimals places. This is in line with the specification of `NICR_PRECISION`.

6.35 Price Feed Compares Timestamp to Blocknumber

Correctness | Medium | Version 1 | Code Corrected

CS-HOG-024

The `Function PriceFeed._backupOracleIsBroken()` compares the `block.number` of the response to the current `block.timestamp` to determine if the backup oracle is broken.

```
// Check for an invalid timeStamp that is 0, or in the future
if (
    _response.blockNumber == 0 ||
    _response.blockNumber > block.timestamp
) {
    return true;
}
// Check for zero price
if (_response.answer == 0) {
    return true;
}

return false;
```

As the `block.timestamp` is always larger than the `block.number` (incremented every 12 seconds), the function returns false for block numbers in the future. Note that `block.number` returns an estimate of the L1 block number on Arbitrum.

Code corrected:

The function `PriceFeed._backupOracleIsBroken()` has been updated to compare the `response.blockNumber` to the current `block.number` instead of the `block.timestamp`.

6.36 PriceFeed Does Not Check if Main Oracle Recovers

Design | Medium | Version 1 | Code Corrected

CS-HOG-025

The function `PriceFeed.fetchPrice()` does not check if the main oracle has recovered when prices are retrieved from the backup oracle (Case 2):

```
// --- CASE 2: The system fetched last price from Backup ---
if (status == Status.usingBackupMainUntrusted) {
    if (
        _priceChangeAboveMax(
            backupOracleResponse,
            prevBackupOracleResponse,
            decimals
        )
    ) {
        _changeStatus(Status.bothOraclesUntrusted);
        return lastGoodPrice;
```

```
    }
    ...
}
```

In case the main oracle recovers and returns a similar price to backup, but the backup oracle reports two prices that deviate more than allowed, the code marks both oracles as untrusted and returns the last stored price.

Code corrected:

The function `fetchPrice()` has been revised to first check if the main oracle has recovered (Case 2) by performing later the checks in the code snippet above.

6.37 Redemption Share Is Rounded to Zero

Design | Medium | Version 1 | Specification Changed

CS-HOG-026

The redemption rate is calculated based on the share of collateral that is redeemed:

$$RedRate = RedFloor + RedBaseRate + RedemptionEth/Collateral$$

In the function `TroveManager._calcRedemptionRate()`, the share of collateral is rounded to zero in the intermediate calculation `_redemptionColl.div(activePool.getWStETH() + defaultPool.getWStETH())`.

Both the numerator and denominator have 18 decimals and the denominator is always larger than numerator. As a result of the rounding error, the redemption rate does not increase when large amounts of collateral are redeemed, and large amounts of Troves can be redeemed against during base fee spikes.

Specification changed:

The function `TroveManager._calcRedemptionRate()` no longer adds the redemption share to the redemption rate (see: Redemption rate double counts the redemption share).

6.38 Unusual Decimals Used for Values in StabilityPool

Correctness | Medium | Version 1 | Code Corrected

CS-HOG-028

The debt token `BaseFeeLMA` in Hedgehog uses 6 decimals, which has side effects in the calculations in the contract `StabilityPool`. Several state variables store values in unusual decimals that deviate from Liquity and are not properly documented.

Although we did not identify concrete issues related to decimals in the formulas used in StabilityPool, the specifications should be extended to clarify the intended decimals for variables.

For instance, the values stored in mappings `epochToScaleToSum` and `epochToScaleToG` use 48 decimals, while memory variables `HOGPerUnitStaked`, `WStETHGainPerUnitStaked` and `BaseFeeLMALossPerUnitStaked` use 30 decimals.

Code corrected:

The debt token `BaseFeeLMA` has been revised to use 18 decimals of precision. This removes the side effects, and the state variables are stored in the same decimal numbers as the specification.

6.39 Wrong Value Used to Calculate the Borrowing Rate With Decay

Correctness Medium Version 1 Code Corrected

CS-HOG-029

The function `TroveManager.getBorrowingRateWithDecay()` incorrectly calculates the borrowing rate of a Trove. It uses the decayed redemption base rate instead of the decayed borrowing base rate.

```
function getBorrowingRateWithDecay(
    uint _issuedBaseFeeLMA
) public view returns (uint) {
    return
        _calcBorrowingRate(
            _calcDecayedRedemptionBaseRate(),
            _issuedBaseFeeLMA
        );
}
```

While this function is not directly used by any contract, it can be utilized by front ends to calculate the maximum fee a user is willing to pay for borrowing baseFeeLMA tokens.

If the value returned by _calcDecayedRedemptionBaseRate() is lower than the value returned by _calcDecayedBorrowingBaseRate(), the expected borrowing rate will be lower than possible. This could lead to a user selecting a value for the maximum fee that is too low, causing borrowing operations to revert.

Code corrected:

The function `getBorrowingRateWithDecay()` has been revised to use the decaying borrowing base rate:

```
return _calcBorrowingRate(_calcDecayedBorrowBaseRate(), _issuedBaseFeeLMA);
```

6.40 Collateralization Ratio Is Rounded Down

Design Low Version 6 Code Corrected

CS-HOG-084

The function `LiquityMath._computeCR()` rounds down the result of the intermediate calculation, since it divides the collateral by the debt before multiplying with `DECIMAL_PRECISION` again.

```
function _computeCR(
    uint _coll,
    uint _debt,
    uint _price
) internal pure returns (uint) {
    if (_debt > 0) {
        uint newCollRatio = (((_coll * DECIMAL_PRECISION) / _debt) *
            DECIMAL_PRECISION) / _price;

        return newCollRatio;
    }
    // Return the maximal value for uint256 if the Trove has a debt of 0. Represents "infinite" CR.
    else {
        // if (_debt == 0)
        return 2 ** 256 - 1;
    }
}
```

As a result `_computeCR` calculates a result that is too small. The relative error is small based on current base fees. However theoretically, the error is such that a trove could be liquidated if the correct CR is 1500000000000000001, because it is incorrectly computed as 1499999999500000000.

Code corrected:

In Version 7, the precision of function `computeCR` has been increased by multiplying the collateral with `DECIMAL_PRECISION**2` before dividing by the debt.

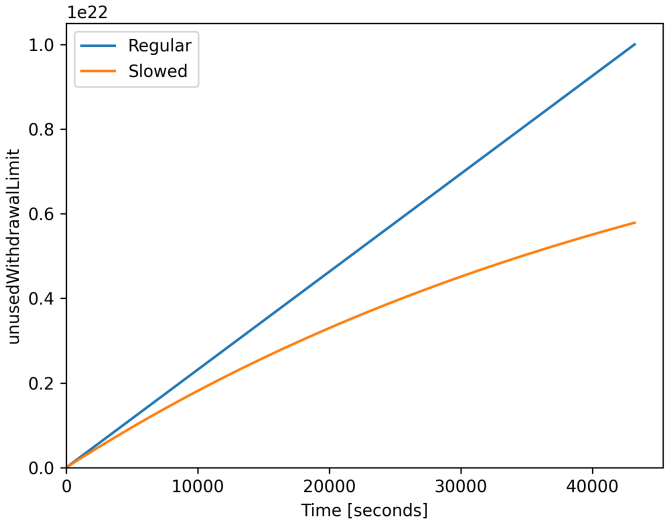
Note that the intermediate computation can theoretically overflow, but for that to happen `_coll > 10**77 / 10**36 = 10**41` (10**77 is roughly the limit of uint256). A collateral value of 10**41 would imply that the system holds 10**23 WStETH. However, currently there are only roughly 10**8 ETH in existence.

6.41 Adversary Can Slow the Recovery of Withdrawal Limit

Security | Low | Version 5 | Code Corrected

CS-HOG-074

Once the withdrawal limit is small, an adversary can slow down its recovery by simply withdrawing a single wei of collateral in each block. Hence, the collateral requirement for the attacker is negligible. The attacker pays for transaction costs of the withdrawals. Keep in mind that transaction costs are fairly low on Base chain. Below we have simulated an example where the `unusedWithdrawalLimit` drops to 0 and the `ActivePool` has a balance of 20,000 WstETH:



As you can see, the adversary can roughly half the recovery of the withdrawal limit over the current recovery window of 12 hours compared to regular execution.

Code corrected:

The withdrawal limits have been removed in Version 6. Hence, the issue is resolved.

6.42 Outdated Specification for `_handleWithdrawalLimit`

Correctness | Low | Version 5 | Code Corrected

CS-HOG-076

The comment before the function `_handleWithdrawalLimit` inside the `BorrowerOperations` is outdated. In particular, it mentions specification items like a `Condition Check When Collateral is Added to the System`. These items do not correspond to the current specification.

Code corrected:

The withdrawal limit has been removed in Version 6. Hence, the issue is resolved.

6.43 Precision Issue in Withdrawal Limit Calculation

Correctness | Low | Version 5 | Code Corrected

CS-HOG-077

To calculate the withdrawal limit the code is using the following calculation:

```
uint256 DENOMINATOR = 100000;

// First, we calculate how much time has passed since the last withdrawal
```

```
uint256 minutesPassed = block.timestamp - _lastWithdrawTimestamp;

// We calculate the percentage based on the time diff between last withdrawal and current moment
uint256 percentageToGet = minutesPassed > _expandDuration
    ? DENOMINATOR
    : (minutesPassed * DENOMINATOR) / _expandDuration;

additionFromNewColl =
    ((totalCollBasedLimit - _unusedWithdrawalLimit) *
        percentageToGet) /
    DENOMINATOR;
```

Note that in Version 5 `_expandDuration` is 720 minutes and hence has a value of `43200`. The issue is that the `percentageToGet` can have a fairly big rounding error. In case only two seconds have passed since the last calculation (which is the current block interval of Base chain), then the `percentageToGet` will be `4`. This is due to the chosen precision, when the correct value would have been roughly `4.63`. Hence, the percentageToGet is off by roughly 14%. Therefore, also the subsequent calculation of `additionFromNewColl` will be off by roughly 14% and the withdrawal limit will recover slower than it was intended.

Code corrected:

The withdrawal limits have been removed in Version 6. Hence, the issue is resolved.

6.44 Unclear Specification Regarding Oracle Decimals

Design | Low | Version 5 | Specification Changed

CS-HOG-078

The `BaseFeeOracle` serves to inform the system about mainnet BaseFee values. Its documentation says:

```
* A custom oracle that's used to feed real world (LogMA50(BaseFeePerGas) * WstETH / ETH ratio)
* value to the system onchain

...

int256 answer; // LogMA50(BaseFeePerGas) * WstETH / ETH ratio in wei

...

uint8 public constant decimals = 18;
```

Reading this suggests that if the `LogMA50(BaseFeePerGas)` were 20 GigaWei and the `WstETH / ETH ratio` would be 1.1, then the feed would have a value of `22 * 10**9 * 10**18`. This would report the logarithmic BaseFee with a precision of 18 decimals.

However, according to our understanding, the feed would have the value `22 * 10**9`. Hence, the specification should be updated to avoid incorrect integrations.

Specification changed:

The comment has been clarified.

6.45 Withdrawal Limit Function Has Discontinuities

Design Low Version 5 Code Corrected

CS-HOG-079

The withdrawal limits have been corrected to be more smooth in Version 5. However, the withdrawal limit function still has some discontinuities close to the limit:

1. If the balance is below the threshold (currently 10 ETH), full withdrawals are possible, regardless of the unusedWithdrawalLimit.
2. If the activePool balance is slightly above the threshold, weird behavior can happen: If the activePool holds 12 ETH and the unusedWithdrawalLimit is 0, then the withdrawal of 1 ETH will be blocked, but the withdrawal of 2 ETH will be allowed.
3. If the activePool balance is between threshold and threshold * 2, then the activePool can be emptied in two steps irrespective of the value of unusedWithdrawalLimit. Concretely, if the balance is 20 ETH and the unusedWithdrawalLimit is 0, it is allowed to withdraw 10 ETH and directly 10 ETH again.

Code corrected:

The withdrawal limits have been removed in Version 6. Hence, this issue is resolved.

6.46 Magic Value for Expand Duration

Correctness Low Version 4 Code Corrected

CS-HOG-067

The contract `BorrowerOperationsArb` uses magic value 720 minutes in `_updateWithdrawlLimitFromCollIncrease` and `setAddresses` instead of referring to the constant `EXPAND_DURATION` of the same value. This makes the code harder to read and maintain.

Code corrected:

The magic value of 720 minutes has been replaced by the constant `EXPAND_DURATION` of the same value in Version 5.

6.47 Missing Event When Changing Withdrawal Limit

Correctness Low Version 4 Code Corrected

CS-HOG-068

In contract `BorrowerOperationsArb` the functions `_handleWithdrawlLimit()` and `_updateWithdrawlLimitFromCollIncrease()` do not emit an event when updating the withdrawal limit or the time of the last withdrawal limit update.

This can make it hard for external users to know the current withdrawal limit or its change over time.

Code corrected:

In Version 5, the event *WithdrawalLimitUpdated* has been added. This event is emitted whenever the withdrawal limit is updated by the functions *_handleWithdrawalLimit()* and *_activePoolAddColl()*.

6.48 Withdrawal Threshold Can Be Circumvented by Splitting Transactions

Design Low Version 4 Code Corrected

CS-HOG-069

Code comments in the function `BorrowerOperationsArb._handleWithdrawalLimit()` state that a single transaction should only be able to withdraw 80% of the withdrawable amount. However, a user can bypass this restriction by splitting the withdrawal of collateral across multiple Troves. By withdrawing 80% of the remaining withdrawable amount in each transaction, a user can effectively withdraw nearly all available collateral in the system. For example, with just 4 Troves, a user can withdraw 1 - (1 - 0.8)^4 = 99.84% of the limit.

Code corrected:

The withdrawal limits have been removed in Version 6. Hence, the issue is resolved.

6.49 Unnecessary Limitation When Opening a Trove

Correctness Low Version 2 Code Corrected

CS-HOG-053

The function `BorrowerOperations.openTrove()` enforces the check:

```
if (_BaseFeeLMAAmount <= vars.BaseFeeLMAFee + BaseFeeLMA_GAS_COMPENSATION) {
    revert("BO: Fee exceeds gain");
}
```

In Version 2, the gas compensation is not included in `_BaseFeeLMAAmount`, hence the check above sets an unnecessary limit.

Code corrected:

The check has been changed to enforce that

```
if (_BaseFeeLMAAmount <= vars.BaseFeeLMAFee) {
    revert("BO: Fee exceeds gain");
}
```

Note that `BaseFeeLMAFee` can be at most `_BaseFeeLMAAmount`, since it is calculated in the previous step as:

$$BaseFeeLMAFee = BaseFeeLMAAmount * maxFeePercentage$$

and `maxFeePercentage` can be at most 100%.

6.50 Event BorrowBaseRateUpdated Is Emitted Twice

Correctness | Low | Version 1 | Code Corrected

CS-HOG-030

When an asset is borrowed, the `BorrowBaseRateUpdated` event is triggered twice.

First, it is emitted in `TroveManager.decayBaseRateFromBorrowing()` and the decayed old base rate is logged. Second, it is emitted in `TroveManager.updateBaseRateFromBorrowing()` to log the updated borrow base rate.

Since the deprecated rate is logged first, external integrators might consume outdated data.

Code corrected:

The `BorrowBaseRateUpdated` is now emitted only once in `TroveManager.updateBaseRateFromBorrowing()`.

6.51 Excess Fee Distribution in FeesRouter

Correctness | Low | Version 1 | Code Corrected

CS-HOG-031

The function `FeesRouter.distributeDebtFee()` divides the fee between three addresses and settles any rounding error with address A.

```
function distributeDebtFee(
    ...
    uint256 totalAmounts = amountA + amountB + amountC;
    if (totalAmounts < _fee) {
    ...
    } else if (totalAmounts > _fee) {
        amountA = amountA + totalAmounts - _fee;
    }
```

However, if the total amount of distributed fees is higher than the generated fee, the rounding error will be added to the amount sent to address A instead of being deducted from it. This doubles the rounding error.

The same problem exists in the function `FeesRouter.distributeCollFee()`.

Code corrected:

Both functions `distributeDebtFee()` and `distributeCollFee()` have been revised to settle the error caused from rounding down in `_calculateAmount()` with address A:

```
if (totalAmounts != _fee) {
    amountA = amountA + _fee - totalAmounts;
}
```

6.52 Function _findPriceBelowMCR Can Be Improved

Design Low Version 1 Specification Changed

CS-HOG-032

The function `LiquityMath._findPriceBelowMCR()` lacks clear specifications and is called only from external functions. `_findPriceBelowMCR()` takes as input a collateral amount `_coll`, debt amount `_debt`, a starting price `_startPrice`, and a collateralization ratio `_mcr`. The function uses an iterative method to find the target price such that the CR of a position with collateral `_coll` and debt `_debt` matches the input `_mcr`.

The function could be improved if using an analytical solution instead of the iterative one.

Specification changed:

The function now uses an analytical solution to find the target price.

6.53 Immutable Parameters Should Be Constants

Correctness Low Version 1 Code Corrected

CS-HOG-033

The accounting of the core contracts works only if the system-wide parameters are equal among all contracts. Furthermore, parameters such as minimum collateralization ratio (MCR) or critical collateralization ratio (CCR) are predefined and should not change on deployment. Therefore, the state variables `BaseFeeLMA_GAS_COMPENSATION`, `MIN_NET_DEBT` and `CCR` in the contract `HedgehogBase` should be constants.

Code corrected:

The listed variables are now declared as constants in the contract `HedgehogBase`.

6.54 Incomplete Error Message

Design Low Version 1 Code Corrected

CS-HOG-034

The function `_requireCallerIsBOorTroveMorSPorFRoute()` in `ActivePool` does not include the fee router in the error message:

```
ActivePool: Caller is neither BorrowerOperations nor TroveManager nor StabilityPool
```

Code corrected:

The error message has been updated to include the fee router:

```
ActivePool: Caller is neither BO nor TM nor FRouter
```

6.55 Incorrect Validation of Repayments

Correctness Low Version 1 Code Corrected

CS-HOG-036

The internal function `_requireValidBaseFeeLMARepayment` in `BorrowerOperations` should limit debt repayments in a trove to its current `debt` minus gas compensation. However, the function does not consider the gas compensation, hence permitting larger repayments:

```
require(
    _debtRepayment <= _currentDebt,
    "BorrowerOps: Amount repaid must not be larger than the Trove's debt"
);
```

Note that if the gas compensation is larger than minimum net debt, the consequences of this issue are severe and could break redemptions. An attacker can lower their debt below `BaseFeeLMA_GAS_COMPENSATION` and a deduction of the gas compensation from user debt will underflow in `TroveManager._redeemCollateralFromTrove()`.

Code corrected:

The gas compensation is now removed from the debt repayment:

```
require(
    _debtRepayment <= _currentDebt.sub(BaseFeeLMA_GAS_COMPENSATION),
    "BorrowerOps: Amount repaid must not be larger than the Trove's debt"
);
```

6.56 Initial Stake Rounds Down to Zero

Design Low Version 1 Code Corrected

CS-HOG-037

The function `StabilityPool._getCompoundedStakeFromSnapshots()` calculates the compounded stake of a trove based on an initial stake and a snapshot. The function implements an if-condition to check if the compounded stake is less than a billionth of the original stake and should return `0` if it is the case:

```
if (compoundedStake < initialStake.div(1e9)) {
    return 0;
}
```

However, given that `initialStake` uses 6 decimals (BaseFeeLMA), the result of `initialStake.div(1e9)` rounds down to zero for stakes smaller than `1_000 * 10**6`.

Code corrected:

BaseFeeLMA token uses 18 decimals in Version 2, which resolves the issue described above.

6.57 Missing Event When Increasing Balance

Correctness Low Version 1 Code Corrected

CS-HOG-039

The function `StabilityPool._increaseBalance()` does not emit the event `StabilityPoolWStETHBalanceUpdated` when updating the wstETH balance. This makes it hard for integrators and dApps to track the wstETH balance of the Stability Pool.

Code corrected:

The function `_increaseBalance()` has been updated to emit the event.

6.58 Missing Sanity Checks

Design Low Version 1 Code Corrected

CS-HOG-040

- The function `PriceFeed._backupOracleIsBroken()` does not check that `roundId` is non-zero and price is positive, which is different from the checks performed for the main oracle.
- The function `FeesRouter.setAddresses()` does not check for non-zero addresses for parameters `_borrowersOp` and `_troveManager`.

In Version 4:

- The function `PriceFeedArb._backupOracleIsBroken()` does check that price is non-zero, but not that it is positive.

In Version 5:

The contract `PriceFeedArb` has been removed from scope.

Code corrected:

The missing sanity checks have been added to the contracts `PriceFeed` and `FeesRouter` in Version 3.

6.59 Misleading Variable Name in BorrowerOperationsArb

Informational Version 4 Code Corrected

CS-HOG-070

The temporary variable `singleTxWithdrawable` in function `BorrowerOperationsArb._handleWithdrawlLimit()` has a misleading name since it represents the amount of collateral in a single *call* and not transaction. Multiple withdrawal requests removing the collateral from multiple troves can be batched in a single transaction.

Code corrected:

The relevant function has been removed. Hence, this finding is resolved.

6.60 Misleading Variable Name in LiquityMath

Informational | Version 4 | Code Corrected

CS-HOG-071

The local variable `minutesPassed` in `_checkWithdrawlLimit` is denominated in seconds and not minutes as the name suggests.

Code corrected:

The relevant function has been removed. Hence, the issue is resolved.

6.61 Withdrawal Limit Does Not Take Collateral From Redistributions Into Account

Informational | Version 4 | Code Corrected

CS-HOG-081

Withdrawal limits are calculated based on the collateral locked in the system. However, only the collateral in the active pool is considered. Collateral from liquidated troves that has been redistributed to other troves is not included. Instead, this redistributed collateral is locked in the default pool.

When users adjust their trove, they realize their pending rewards, moving collateral from the default pool to the active pool. However, this does not immediately increase their withdrawal limit; instead, the limit gradually increases through the recovery mechanism.

Code corrected:

This finding is resolved, as withdrawal limits have been removed.

6.62 Remaining ToDos

Informational | Version 3 | Code Corrected

CS-HOG-058

The following ToDo comment is present in contracts `TroveManager` and `TroveManagerArb`:

```
* HEDGEHOG UPDATES:
* 1) Now passing _calcDecayedBorrowBaseRate instead of _calcDecayedBaseRate
     function to calculate the decayed borrowBaseRate
* TODO: Write test
```

Code Corrected:

The TODO has been removed.

6.63 Gas Optimizations

Informational | Version 1 | Code Corrected

CS-HOG-042

1. Function `HOGToken._transfer()` redundantly checks that `recipient` is not address zero, which is already checked in external functions `transfer()` and `transferFrom()`.
2. The state variable `feesRouter` in contract `BaseFeeLMAToken` can be declared as immutable.
3. The function `BaseFeeOracle.feedBaseFeeValue()` could save a SLOAD operation by setting `latestRound` to `round`.
4. The check `_BaseFeeLMAAmount <= vars.BaseFeeLMAFee + BaseFeeLMA_GAS_COMPENSATION` in `BorrowOperations.openTrove()` can be moved up to fail early.
5. Function `_requireCallerIsActivePool()` in the contract `DefaultPool` remains unused in the codebase.
6. The state variable `feeCount` in the contract `FeesRouter` is unused.
7. Several contracts inherit `SafeMath` library although solidity version `0.8.19` is used.
8. Function `StabilityPool.withdrawWStETHGainToTrove()` triggers redundant execution of `getDepositorWStETHGain()` in its internal function calls.
9. The state variable `ISSUANCE_FACTOR` is set upon declaration and then written again in the constructor of `CommunityIssuance`.
10. Event `BaseFeeLMATokenBalanceUpdated` in the contract `BaseFeeLMAToken` remains unused in the codebase.
11. The constant `BETA` in the contract `TroveManager` remains unused in the codebase.
12. The constant `ONE_YEAR_IN_SECONDS` in the contract `HOGToken` remains unused.
13. The immutable `BOOTSTRAP_PERIOD` in the contract `TroveManager` can be defined as constant.

Version 3:

14. Contract `StabilityPool` inherit `LiquitySafeMath128` library although solidity version `0.8.19` is used.

Version 4:

15. Function `BorrowOperationsArb._adjustTrove()` could use the boolean `vars.isCollIncrease` to check if collateral has been withdrawn.
16. Function `BorrowOperationsArb._updateWithdrawlLimitFromCollIncrease()` could use the value of `newColl` to calculate `newLimit`.
17. The local variable `DENOMINATOR` in `_checkWithdrawlLimit` could be defined as constant.
18. Function `BorrowerOperationsArb._handleWithdrawlLimit()` calls redundantly `getWStETH()` in the active pool.

Code corrected:

The optimizations were implemented.

6.64 Incorrect Interfaces

Informational Version 1 Code Corrected

CS-HOG-043

The function `setAddresses` defined in the interfaces `ICommunityIssuance`, `IBorrowerOperations`, `ICollSurplusPool`, `IStabilityPool`, and `ITroveManager` accepts fewer arguments than their respective implementations in `CommunityIssuance`, `BorrowerOperations`, `CollSurplusPool`, `StabilityPool`, and `TroveManager`.

To maintain consistency between an interface and its corresponding contract, it is considered best practice to have the contract inherit its interface.

Code corrected:

The listed contracts have been updated to inherit the respective interfaces.

6.65 Misleading Variable Name in BaseFeeOracle

Informational Version 1 Code Corrected

CS-HOG-044

The struct `Response` in contract `BaseFeeOracle` has a variable named `currentChainBN`. This variable is set to `block.number` in function `feedBaseFeeValue()`. However, `block.number` in Arbitrum returns the estimated Layer 1 block number, which is different from the block number in the current chain as the name suggests.

A more detailed description of `block.number` can be found in the official docs.

Code corrected:

A new contract `BaseFeeOracleArb` was introduced in Version 3 that stores Arbitrum block numbers in the variable `currentChainBN`.

6.66 Misleading Variable Name in FeesRouter

Informational Version 1 Code Corrected

CS-HOG-045

The input argument `_debt` in function `FeesRouter.distributeCollFee()` is misleading. The input argument represents a collateral amount.

Code corrected:

The variable has been renamed as `_coll`.

6.67 Misleading Variable Name in TroveManager

Informational | Version 1 | Code Corrected

CS-HOG-046

The temporary variable `redeemedBaseFeeLMAFraction` in function `TroveManager._updateRedemptionBaseRateFromRedemption()` has a misleading name. The variable `redeemedBaseFeeLMAFraction` is set to the fraction of collateral that is redeemed and not the share of `BaseFeeLMAToken` redeemed.

Code corrected:

The variable has been renamed to `redeemedCollFraction` which is in line with the amount of the token it stores.

6.68 Vulnerable Dependency

Informational | Version 1 | Code Corrected

CS-HOG-049

The Hedgehog's contract makes use of the OpenZeppelin Contracts Library (Version 4.9.3) with a known vulnerability. More details can be found here: https://github.com/advisories/GHSA-9vx6-7xxf-x967

While the contracts in scope do not use the vulnerable function, it is considered best practice to upgrade to a patched version of the library.

Code Corrected:

The dependency has been upgraded.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Incorrect Comments About Subsequent Base Fee LMA Prices

Informational **Version 5**

CS-HOG-082

The `MAX_PRICE_DEVIATION_PERCENTAGE_FROM_PREVIOUS_ROUND` is currently set to 17.6% (176) in the code. According to the code comments, the oracle triggers a price updates as soon as the price deviates by more than 5%. Thus, Hedgehog expects that the price can change by 12.5% between two updates leading to a maximum deviation of 17.6%.

```
// HEDGEHOG UPDATES: decreased to 176
// Maximum deviation allowed between two consecutive main oracle prices.
// Hedgehog oracles trigger updates when the price deviation exceeds 5%.
// Thus, the maximum possible deviation between rounds is 17.6%.
uint public constant MAX_PRICE_DEVIATION_PERCENTAGE_FROM_PREVIOUS_ROUND = 176;
```

Yet, if the Base Fee LMA token could increase by 12.5% in this scenario, then a threshold of 17.6% would not be enough, since deviations are multiplied and not added together. Consider the case in which the previous oracle price is 1 Gwei, and the current price is 1.0499 Gwei (just below the 5% threshold). Then the new price would be:

$P = 1.0499 * 1.1250 = 1.181 > 1.176$

However, the scenario outlined is not possible, as the current specification of the LMA price of the base fee would not allow it. The LMA base fee for block t is calculated as the logarithmic moving average of the base fee over the last 50 blocks:

$$LMA_t = \frac{\sum_{i=0}^{49} baseFee_{t-i} \times w_i}{\sum_{i=0}^{49} w_i}$$

In this formula, the smallest weight is assigned to the most recent block, while the largest weight is assigned to the oldest block within the window. Weights are defined as:

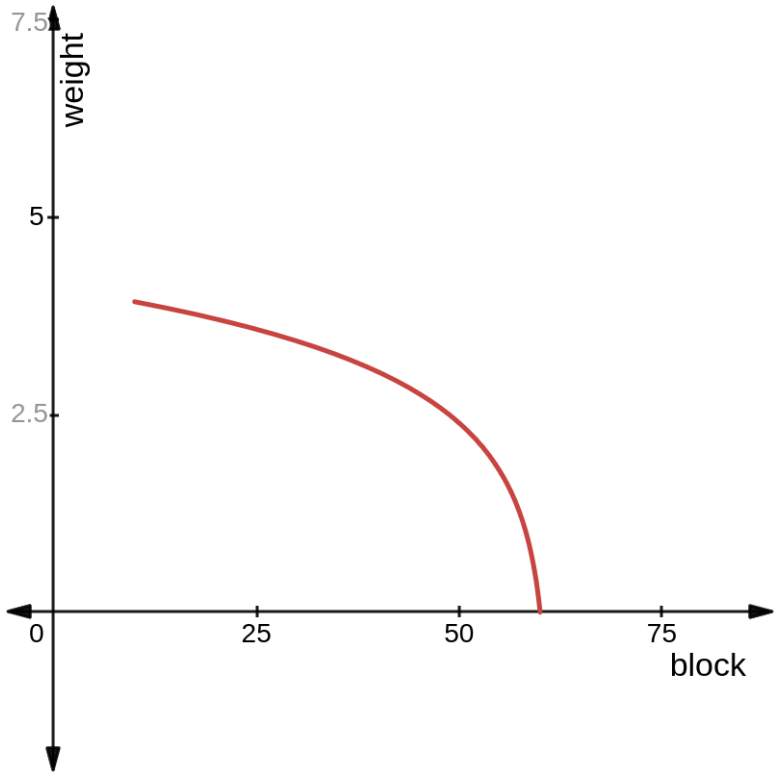
$w_{t,i} = \ln(t + 1 - i)$

So in the calculation of the Base Fee LMA at block 60, the base fee at block 60 has the smallest weight, and the base fee at block 11 has the largest weight:

For the Base Fee LMA to increase by 12.5% from block 60 to block 61, the base fee would need to increase by 12.5% in block 12 to block 61:

$$baseFee_{t+1} = 1.125 \times baseFee_t = \frac{\sum_{i=0}^{n-1} 1.125 \times baseFee_{t-i} \times w_i}{\sum_{i=0}^{len-1} w_i}$$

Note that blocks 12 to 60 not only affect the Base Fee LMA at block 61, but also the LMA at previous block 60. The primary differences between the LMA's at block 60 and block 61 are that block 11 is included only in block 60's LMA, and block 61 is included only in block 61's LMA.



As illustrated in the graph above, the weight of any single block is small relative to the aggregate weights, meaning a 2 observation difference is not enough to change the value significantly. Hence, the base fee at block 60 itself would also need to be approximately 12.5% higher than the base fee at block 59. Such an increase would violate the assumption that the price deviation remains within the 5% threshold, since any deviation exceeding 5% would immediately trigger an oracle price update, resetting the deviation to 0%. Note that if the weighting scheme is altered - for instance, by a huge weight to the last block — the contribution of a single block (i.e. block 11) could become large enough to allow the Base Fee LMA to increase by 12.5% in this scenario.

7.2 Incorrect Error Message in Recovery Mode

Informational **Version 5**

CS-HOG-080

In Hedgehog's code recovery mode has no effect on the amount of borrowing fees charged. However, function `BorrowerOperations._requireValidMaxFeePercentage` has a special case in recovery mode allowing a user to set a max fee lower than the fee floor in recovery mode instead of reverting.

```
function _requireValidMaxFeePercentage(
    uint _maxFeePercentage,
    bool _isRecoveryMode
) internal pure {
    if (_isRecoveryMode) {
        require(
            _maxFeePercentage <= DECIMAL_PRECISION,
            "Max fee percentage must less than or equal to 100%"
        );
    } else {
        require(
```

```
            _maxFeePercentage >= BORROWING_FEE_FLOOR &&
                _maxFeePercentage <= DECIMAL_PRECISION,
            "Max fee percentage must be between 0.5% and 100%"
        );
    }
}
```

Note that the execution will still revert later during the execution of `_requireUserAcceptsFee`, but the error message will not be as informative as it could. The error message will not reflect whether the user passed an incorrect fee value (below the fee floor) or if the user passed a fee value that is lower than the current borrowing fee.

7.3 Base Fee Oracle Is Incompatible With Chainlink Interface

Informational **Version 1**

CS-HOG-041

The contract `BaseFeeOracle` uses similar function names as Chainlink but it is not compatible with Chainlink's interface. For instance, the declaration of function `getRoundData()` is:

```
function getRoundData(uint80 _roundId) public view returns (int256, uint256, uint256, uint80);
```

while the Chainlink's interface has the following:

```
function getRoundData(uint80 _roundId) external view returns (uint80, int256, uint256, uint256, uint80);
```

Updates in **Version 2**:

The function `getRoundData()` has been revised to follow the Chainlink interface, however the `decimals()` are of type `uint256` instead of `uint8`.

Updates in **Version 5**:

Technically, the function `getRoundData()` does not follow the Chainlink interface, however the difference in the value ranges is unlikely to become an issue.

Integrators should be aware that not all functions like `description` and `version` are implemented and `getRoundData` returns the (L1) block number and not the block timestamp.

7.4 Race Conditions When Opening Troves

Informational **Version 1**

CS-HOG-047

In Hedgehog's code, borrow base rates increase with each additional borrow before decaying back to zero. If two transactions are pending, the first transaction executed will pay a lower fee than the second transaction.

Similar to Liquity, a user can specify the value `_maxFeePercentage` to limit the percentage fee they are willing to pay. However, racing other users is considerably easier in Hedgehog's protocol, as borrowing

fees increase with each borrow, while they depend on redemptions in Liquity. Note that redemptions are costly to perform under the wrong market conditions.

Furthermore, Hedgehog removed the 5% cap on borrowing fees from the codebase. It is important that the changes are well documented so that users are aware of setting an acceptable value for `_maxFeePercentage` to limit the fees paid.

7.5 Slightly Larger Collateral Reported

Informational **Version 1**

CS-HOG-048

Theoretically, the function `HedgehogBase.getEntireSystemColl()` can return `1` wei more than the actual collateral due to `ActivePool.getWStETH()` returning `1` when the actual collateral is `0`.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Bridged Version of HOG Lacks Permit

Note Version 5

The `HogToken` in scope of the review is intended to be deployed on Ethereum Mainnet and then bridged to Base Chain with the native bridge.

As a result of this bridging process, the `HogToken` on the Base Chain will be deployed via the `OptimismMintableERC20Factory` bridge, making it an `ERC20Bridged` token. However, the `ERC20Bridged` does not retain all the features of the original token—most notably, it does not include a `permit` function.

This limitation may affect any operations or integrations that rely on gasless approvals via EIP-2612.

8.2 Dependency on EIP-1559 Specification

Note Version 5

The price of `BaseFeeLMAToken` is computed as the logarithmic moving average of the base fee over the last 50 Ethereum mainnet blocks.

The Ethereum Base Fee mechanism is specified in EIP-1559.

If the base fee pricing mechanism is modified by a future Ethereum upgrade, it could potentially break Hedgehog's protocol. We have identified at least two ways in which the protocol could fail:

1.The base fee is expected to change by a maximum of 12.5% per block. If this limit is exceeded, the maximum deviation threshold of the price feed could get exceeded and the protocol would stop to accepting new prices.

2.The EIP-1559 implicitly enforces a minimum base fee of 7 wei. If this limit would change in the future and the base fee drops to a value as low as 1 wei, then any trove with a collateralization ratio below 10 could be rounded down by `LiquityMath._computeCr()`. Below we argue why the current limit is 7 wei:

Note that the base fee decreases when less gas is used than the target gas. So when no gas is used we have gas_used_delta = parent_gas_target - 0 = parent_gas_target. The increase of the next base fee can then be shown to be parent_base_fee_per_gas // BASE_FEE_MAX_CHANGE_DENOMINATOR (= 8):

```
base_fee_per_gas_delta = parent_base_fee_per_gas * gas_used_delta // parent_gas_target // BASE_FEE_MAX_CHANGE_DENOMINATOR

base_fee_per_gas_delta = parent_base_fee_per_gas * parent_gas_target // parent_gas_target // 8

base_fee_per_gas_delta = parent_base_fee_per_gas // 8
```

Now, if `parent_base_fee_per_gas` falls below 8 wei, then `base_fee_per_gas_delta` becomes zero as the expression rounds down to zero. Hence, the limit is 7 wei.

8.3 Incorrect Permit Functions

Note Version 5

The `permit` functions of the `HOGToken` and the `BaseFeeLMAToken` allow to set an ERC20 approval based on a signature. They use `ecrecover` for the signature check, but do not check whether the result is zero. Hence, anyone can give ERC20 approvals in the name of the Zero-Address for these tokens. However, as transfers to and from the Zero-Address are blocked, this should not have an impact.

8.4 Liquidations Can Incur Losses to Stability Pools

Note Version 2

The minimum collateralization ration (MCR) in Hedgehog is set to `150%`. Any trove with a collateralization ratio (CR) lower than MCR is eligible to be liquidated. The liquidations are expected to happen when a trove's CR is above 100%, hence the stability pool makes a profit. The stability pool should make a loss only when a trove is liquidated when its CR is below 100%.

As highlighted in Pegging Mechanisms Are Less Strict, Hedgehog does not enforce a hard upper bound on the price of the debt token `BaseFeeLMA`. It is possible that `BaseFeeLMA` can trade in secondary markets at 150% (or above) of the oracle price. Therefore, even if liquidations happen when a trove's CR above 100%, the stability pool might incur losses if the price in the secondary market is high.

8.5 Low Redemption Fees Due to High Overcollateralization

Note Version 2

The redemption fees are determined by the proportion of collateral redeemed relative to the system's total collateral.

The Hedgehog team anticipates a high overcollateralization ratio between 750% and 1000%. As a result, redemptions removing large amounts of the debt supply only remove a small fraction of the total collateral and hence pay a low redemption fee.

Using the formulas for the redeem collateral (baseFeeDebt * price) and the TCR (TotalCollateral / TotalDebt / price), we can derive the redemption share depending on the baseFeeDebt and the TCR:

$$RedemptionShare = \frac{RedeemCollateral}{TotalCollateral} = \frac{RedeemDebt * price}{TotalDebt} * \frac{TotalDebt}{TotalCollateral} = \frac{RedeemDebt}{TotalDebt} * \frac{1}{TCR}$$

For example, with a 1000% overcollateralization ratio and a base rate of 1%, redeeming 10% of the supply would incur a fee of 2.5%

$$RedemptionShare = 0.5\% + 1\% + 10\% * \frac{1}{10} = 0.5\% + 1\% + 1\% = 2.5\%$$

and increase the base rate to 2% for the next redemption.

8.6 Minimum Debt Value of a Trove

Note Version 1

In Version 1, the minimum debt amount for a trove is 50 million `baseFeeLMA` tokens. The value of the minimum debt scales linearly with the base fee on mainnet. If we consider 3 values from the typical range of baseFee on mainnet, the value of min debt changes significantly:

- base fee @ 1 Gwei -> min debt value is 0.05 ETH
- base fee @ 50 Gwei -> min debt value is 2.5 ETH
- base fee @ 100 Gwei -> min debt value is 5 ETH

The historical data from Ethereum mainnet shows that the gas price can spike further than 100 Gwei, which would cause the minimum debt value to be even larger.

Conversely, if the base fee drops to 1 Gwei, the minimum debt decreases to 0.05 ETH. Therefore, the griefing attack that inject troves in the linked list to make user transactions such as `openTrove()` or `redeemCollateral()` revert, are more likely when base fee is low.

Hedgehog is aware of this behavior and considers it to work according to the system design.

Changes in Version 4:

The minimum debt amount of a trove has been increased from 50 to 100 million base fee tokens.

8.7 Pegging Mechanisms Are Less Strict

Note Version 1

Hedgehog uses similar mechanisms as Liquity to maintain the pegging of `BaseFeeLMA` to the actual base fee in Ethereum mainnet. However, important system-wide parameters have been altered, resulting in less stringent pegging mechanisms. Consequently, `BaseFeeLMA` can fluctuate in a wider price range:

- Upper bound: The minimum collateralization ratio (MCR) enforces the upper limit of the price fluctuation for the debt token. If `BaseFeeLMA` is priced high enough in a secondary market, an arbitrage opportunity is opened as users can open undercollateralized troves and sell the debt token in a secondary market to make a profit. Hedgehog sets MCR to 150%, meaning that `BaseFeeLMA` can trade in a secondary market up to 150% of its actual value in mainnet before the pegging mechanism (arbitrage opportunity) kicks in to limit further price increase. Liquity sets MCR to 110%.
- Lower bound: Redemptions enforce that the price of the debt token does not fall below a certain limit in the secondary markets. If the price of `BaseFeeLMA` drops below this limit, redemptions open an arbitrage opportunity as one can buy `BaseFeeLMA` tokens from the market and redeem them at their face value in Hedgehog. A fee is charged on redemption, which influences the lower limit.

Moreover, in Hedgehog, redemptions do not increase the borrowing fee. This is because the borrowing fees are tied to a borrow base rate and not the (redemption) base rate as in the case of Liquity. When the `BaseFeeLMA` token trades below its face value, redemptions occur. Since borrowing fees remain unaffected, the price experiences further downward pressure due to the issuance of additional tokens.

A full review of the soundness of the financial model was not in scope of this review.

8.8 Returned Price When Both Oracles Are Untrusted

Note Version 5

The function `PriceFeed.fetchPrice()` checks if the backup oracle is broken or its last two prices deviate more than allowed (Case 1), and returns the last good price if one of the conditions is satisfied:

```
// If Backup is broken, both oracles are untrusted, and return last good price
if (
    _backupOracleIsBroken(backupOracleResponse) ||
    _priceChangeAboveMax(
        backupOracleResponse,
        prevBackupOracleResponse,
        decimals
    )
) {
    _changeStatus(Status.bothOraclesUntrusted);
    return lastGoodPrice;
}
```

This behavior is different from Liquity which accepts the Chainlink price if both oracles report a similar price. Imagine a scenario where the price increases by 12.5% in two subsequent blocks and the backup is currently used. With the current contract design Block 102 would return the last good price, instead of the main oracle price:

Ethereum Block 100:

1. Base fee is 1
2. Main oracle price is broken
3. Backup oracle price is 1
4. price is 1

Ethereum Block 101:

1. Base fee is 1.1
2. Main oracle price is broken
3. Backup oracle not updated
4. price is 1

Ethereum Block 102:

1. Base fee is 1.2
2. Main oracle report 1.2
3. Backup oracle report 1.2
4. price is 1

Ethereum Block 103:

1. Base fee is 1.2
2. Main oracle report 1.2
3. Backup oracle report 1.2
4. price is 1.2

Similarly, if the backup oracle is used when the main oracle is broken (Case 2), and the main oracle has not recovered, then `PriceFeed.fetchPrice()` will check if the backup oracle response is more than the allowed deviation away from the previous oracle response. If the deviation is greater than the allowed deviation, the function will return the last good price otherwise it accepts the backup price. In Liquity's code the backup price would be accepted even when the backup oracle response is more than the allowed deviation away from the last good price.

8.9 State Variable totalHOGIssued Might Deviate From Actual HOG Issued

Note **Version 1**

The state variable `CommunityIssuance.totalHOGIssued` is public and can be queried externally to get the amount of HOG tokens that have already been issued. However, callers should be aware that this value can be updated by privileged accounts and might not reflect the actual amount of issued HOG tokens:

```
function setTotalHogIssued(
    uint _newHogIssued
) external onlyRole(DISTRIBUTION_SETTER) {
    totalHOGIssued = _newHogIssued;
    emit TotalHogIssuedManuallyUpdated(_newHogIssued);
}
```

8.10 Tokens Can Get Stuck When the Issuance Is Reduced

Note **Version 3**

An address with `DISTRIBUTION_SETTER` role can change the overall token issuance with functions `setHOGSupplyCap()`, `setISSUANCE_FACTOR()`, or `setTotalHogIssued()`. The privileged addresses that can call these functions are responsible to ensure that the changes are atomic, and the new issuance curve is correct, hence any change of these parameters should be evaluated carefully.

Note that there is no mechanism to withdraw tokens from the contract, so if the deployer initially transfers 1 million tokens to the contract and subsequently decides to reduce the total issuance, a portion of the tokens may become irretrievably stuck.

8.11 Trove Changes Are Forbidden in Specific Scenarios

Note **Version 1**

Trove modifications, such as removing collateral or closure, are forbidden in case the total collateralization of the system goes below the critical collateralization ration (200%) afterwards. Therefore, it is possible that a borrower cannot close its trove or withdraw collateral in specific scenarios.