Code Assessment

of the Sulu Extensions IV Smart Contracts

March 07, 2024

Produced for

enzyme

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Mona and Sean,

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

Avantgarde Finance implements a new version of the Curve price feed and adapted the Curve liquidity and and Convex Curve adapters while minor updates to the ParaswapV5 adapter have been made. Moreover, external positions for lending on Maple, borrowing on Liquty, vote-locking for Convex, and delegating on The Graph were implemented. Also, a shares splitting contract for splitting fees, including its surrounding architecture, were implemented.

The most critical subjects covered in our audit are functional correctness, access control and integration with external systems. Security regarding all the aforementioned subjects is high.

The general subjects covered are upgradeability, documentation, specification, gas efficiency, trustworthiness. Security regarding all the aforementioned subjects is high.

In summary, we find that the codebase provides a high level of security.

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	0
High-Severity Findings	1
• **Code Corrected**	1
Medium-Severity Findings	2
• **Code Corrected**	2
Low-Severity Findings	4
• **Code Corrected**	4

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 Sulu Extensions IV 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	18 April 2022	ea6c11903cc48e1a6cbb36dedcfab85ed50a2833	Initial Version
2	23 April 2022	2ab92f4a89800b772f9d50bc0ffee9f702a2955f	Curve related Fixes
3	09 May 2022	422808344fdc117be2c7df4f39c39a2d35f93143	Final Version
4	07 March 2024	f143dfc5fe98f5e2f0b02ff946a2272b60b095f4	Report Update

For the solidity smart contracts, the compiler version `0.6.12` was chosen.

The following contracts where in scope:

Curve Price Feed and Adapters:

```
* contracts/release/infrastructure/price-feeds/derivatives/feeds/CurvePriceFeed.sol
* contracts/release/interfaces/ICurveLiquidityPool.sol
* contracts/release/interfaces/ICurvePoolOwner.sol
* contracts/release/interfaces/ICurveRegistryMain.sol
* contracts/release/interfaces/ICurveRegistryMetapoolFactory.sol
* contracts/release/extensions/integration-manager/integrations/utils/bases/CurveLiquidityAdapterBase.sol
* contracts/release/extensions/integration-manager/integrations/adapters/CurveLiquidityAdapter.sol
* contracts/release/extensions/integration-manager/integrations/adapters/ConvexCurveLpStakingAdapter.sol
* contracts/release/extensions/integration-manager/integrations/utils/actions/CurveGaugeV2RewardsHandlerMixin.sol
```

Maple External Position:

```
* contracts/release/interfaces/IMapleMplRewards.sol
* contracts/release/interfaces/IMapleMplRewardsFactory.sol
* contracts/release/interfaces/IMaplePool.sol
* contracts/release/interfaces/IMaplePoolFactory.sol
* contracts/release/extensions/external-position-manager/external-positions/maple-liquidity/IMapleLiquidityPosition.sol
* contracts/persistent/external-positions/maple-liquidity/MapleLiquidityPositionLibBase1.sol
* contracts/release/extensions/external-position-manager/external-positions/maple-liquidity/MapleLiquidityPositionDataDecoder.sol
* contracts/release/extensions/external-position-manager/external-positions/maple-liquidity/MapleLiquidityPositionLib.sol
* contracts/release/extensions/external-position-manager/external-positions/maple-liquidity/MapleLiquidityPositionParser.sol
* contracts/release/utils/AssetHelpers.sol
```

Global config and shares splitter:

```
* contracts/persistent/shares-splitter/SharesSplitterFactory.sol
* contracts/persistent/shares-splitter/SharesSplitterLib.sol
* contracts/persistent/shares-splitter/SharesSplitterProxy.sol
* contracts/persistent/shares-splitter/TreasurySplitterMixin.sol
* contracts/persistent/global-config/GlobalConfigLib.sol
* contracts/persistent/global-config/GlobalConfigProxy.sol
* contracts/persistent/global-config/bases/GlobalConfigLibBase1.sol
* contracts/persistent/global-config/bases/GlobalConfigLibBaseCore.sol
* contracts/persistent/global-config/utils/GlobalConfigProxyConstants.sol
* contracts/persistent/global-config/utils/ProxiableGlobalConfigLib.sol
* contracts/persistent/global-config/interfaces/IGlobalConfig1.sol
* contracts/persistent/global-config/interfaces/IGlobalConfigVaultAccessGetter.sol
```

ParaswapV5 Adapter Changes:

```
* contracts/release/extensions/integration-manager/integrations/adapters/ParaSwapV5Adapter.sol
* contracts/release/extensions/integration-manager/integrations/utils/actions/ParaSwapV5ActionsMixin.sol
* contracts/release/interfaces/IParaSwapV5AugustusSwapper.sol
```

Convex Voting External Position:

```
* contracts/release/interfaces/IConvexBaseRewardPool.sol
* contracts/release/interfaces/IConvexCvxLockerV2.sol
* contracts/release/interfaces/IConvexVlCvxExtraRewardDistribution.sol
* contracts/release/interfaces/ISnapshotDelegateRegistry.sol
* contracts/release/interfaces/IVotiumMultiMerkleStash.sol
* contracts/release/extensions/external-position-manager/external-positions/convex-voting/IConvexVotingPosition.sol
* contracts/release/extensions/external-position-manager/external-positions/convex-voting/ConvexVotingPositionDataDecoder.sol
* contracts/release/extensions/external-position-manager/external-positions/convex-voting/ConvexVotingPositionLib.sol
* contracts/release/extensions/external-position-manager/external-positions/convex-voting/ConvexVotingPositionParser.sol
```

Liquidity Borrowing external positions:

```
* contracts/release/interfaces/ILiquityBorrowerOperations.sol
* contracts/release/interfaces/ILiquityTroveManager.sol
* contracts/release/extensions/external-position-manager/external-positions/liquity-debt/ILiquityDebtPosition.sol
* contracts/release/extensions/external-position-manager/external-positions/liquity-debt/LiquityDebtPositionDataDecoder.sol
* contracts/release/extensions/external-position-manager/external-positions/liquity-debt/LiquityDebtPositionLib.sol
* contracts/release/extensions/external-position-manager/external-positions/liquity-debt/LiquityDebtPositionParser.sol
```

The Graph delegations:

```
* contracts/release/interfaces/ITheGraphStaking.sol
* contracts/release/extensions/external-position-manager/external-positions/the-graph-delegation/ITheGraphDelegationPosition.sol
* contracts/release/extensions/external-position-manager/external-positions/the-graph-delegation/TheGraphDelegationPositionDataDecoder.sol
* contracts/release/extensions/external-position-manager/external-positions/the-graph-delegation/TheGraphDelegationPositionLib.sol
* contracts/persistent/external-positions/the-graph-delegation/TheGraphDelegationPositionLibBase1.sol
```

In the review of the commit for V4, only the changes related to the Liquity EP described in Changes in V4 were in scope.

2.1.1 Excluded from scope

Curve is not part of this review and expected to work correctly as documented. The Maple Protocol is not part of this review and expected to work correctly as documented. ParaswapV5 is expected to work as expected in the main audit report for that adapter. The Convex Voting contracts are expected to work as documented. Moreover, Votium and the CvxCRV contract are expected to work as documented. Liquity is not part of this review and is assumed to function as described in the documentation. The Graph Protocol is not part of this review and expected to work correctly as documented.

The potential future use cases of the TreasurySplitterMixin are unknown and out of scope.

2.2 System Overview

This system overview describes the initially received version ($\boxed{\text{Version 1}}$) of the contracts as defined in the Assessment Overview.

At the end of this report section we have added subsections for each of the changes accordingly to the versions.

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

Avantgarde Finance offers updates to the current Curve price feed which implies also changes for the Curve integrations. Also minor changes to the ParaswapV5 adapters have been made. New external positions are introduced, namely it is now possible to lend and stake on Maple, borrow on Liquity and vote-lock convex and delegate the votes. Additionally, a shares splitter contract was implemented; it can be deployed by fund managers and specified as a fee recipient so fees can be split among multiple

parties at a constant ratio. For that, some global config was introduced as a part of the surrounding global architecture.

2.2.1 Curve price feed and Curve integrations

Curve is a decentralized exchange with which Enzyme integrates. Users can deposit liquidity and earn exchange fees. The Curve integration allows depositing and removing liquidity from Curve as well as staking the LP tokens to Curve's gauges. Additionally, the ConvexCurve integration, which allows staking through Convex, has been updated. Many changes on the integrations result from changes made to the Curve price feed.

Compared to the previous version, the most notable changes are

- Avantgarde Finance adds a view-reentrancy protection mechanism to the Curve price feed to validate the current virtual price of an LP token by calling a reentrancy protected function on the Curve pool, if the current virtual price deviates too much from the last validated price.

- The Curve price feed now supports metapools by using the metapool factory as an additional registry. The ConvexCurve and Curve adapters have been adapted to query the price feed for the registry information (instead of Curve directly). Hence, the adapters also support metapools.

- The reward claiming mechanism for Curve has been adapted to support non-Ethereum chains by skipping the direct minting of CRV rewards. In contrast to Mainnet, these are not minted directly to users but are distributed as regular rewards.

For a general description of the integrations see our previous audit reports. Regarding the updated price feed we will give a description of the changes.

The Curve price feed supports the following external price feed functions used in the Enzyme system:

- `calcUnderlyingValues`: Given a derivative token (Curve LP) and its amount, it approximates the value in the proxy asset (one of the underlying assets) by using Curve's `get_virtual_price()` function. Note that this functions also protects from view-reentrancies if the virtual price deviation is too high by calling `withdraw_admin_fees(pool)` on the Curve admin contract.

- `isSupportedAsset`: Defines whether an asset is supported by the price oracle.

Furthermore, `updateValidatedVirtualPrices()` allows updating the last validated virtual price.

The following admin functionality is implemented:

- `addGaugeTokens(WithoutValidation)`: Adds Gauge tokens. Without validations they are not checked against the registry or the metapool factory.

- `addPools(WithoutValidation)`: Adds LP tokens. Without validations they are not checked against the registry or the metapool factory.

- `removeDerivatives()`: Removes the derivative and makes it unsupported.

- `removePools()`: Removes the price feed.

- `updatePoolInfo()`: Updates the information on the pool.

Note that the separation of removal logic of `removeDerivatives()` and `removePools()` allows for a complete removal of the pool but also adds flexibility.

2.2.2 Maple lending

Maple is a DeFi protocol offering under-collateralized borrowing for institutions / whitelisted actors. Lenders can earn interests and rewards by providing capital. Capital deposited is subject to a minimum lockup period defined by the pool before it can be withdrawn. Furthermore, lenders may stake to earn MPL tokens as a reward. For more information please refer to their documentation: https://maplefinance.gitbook.io/maple/

A new type of external positions will enable vaults of Enzyme to lend capital into Maple pools. For an overview of how external positions work in Enzyme, please refer to section 2.2.2 External Positions of the system overview of the Sulu report.

After opening such an external position for a fund, the following actions are available to interact with the maple protocol:

- `Lend`: Allows lending the given amount to the specified pool, calls `pool.deposit()`.

- `Stake`: Allows staking the given amount of the given pool into the given staking rewards contract. Executes calls to `pool.increaseCustodyAllowance()` and `rewardsContract.stake()`.

- `LendAndStake`: Wrapper to lend and stake in one call.

- `IntendToReedeem` Signals intention to redeem, starts cool-down period after which redemption is possible. Calls `pool.intendToWithdraw()`.

- `Redeem`: Redeems the given amount from the given pool using `pool.withdraw()`.

- `Unstake`: Unstakes the given amount from the given rewards contract. Calls `rewardsContract.withdraw()`.

- `UnstakeAndRedeem`: Wrapper to unstake and redeem in one call.

- `ClaimInterest`: Allows claiming accrued interests. Calls `pool.withdrawFunds()` and transfers the interests in form of the liquidity asset to the vault.

- `ClaimRewards`: Allows claiming the rewards from staking in form of the reward token (understood to be the MPL token). Calls `getReward()` on the rewards contract and transfers the tokens onwards to the vault.

The Maple protocol is fully trusted to work correctly as described. Note that several of the Maple contracts can be paused, which inhibits interactions, including withdrawals.

2.2.3 Global Config and Shares Splitter

Avantgarde Finance introduces a global config contract to allow sharing storage among different contracts in the system. It introduces a proxy contract and an implementation following EIP-1822 and EIP-1967. The proxy initializes itself on creation. The library offers the following additional functionality:

- `getDispatcher()` to get the dispatcher.

- `setGlobalConfigLib()` to change the global config library and `getGlobalConfigLib()` to get the current library.

- `isValidRedeemSharesCall()` to validate whether a redeem shares call is valid. If a fund is a V4 fund, it checks the validity of the redeem function and the contract on which it is intended to be called. Furthermore, it can optionally check whether the recipient and the amount match the argument that will be provided to the redemption call. In case the validation is passed, `true` is returned. Otherwise, `false` is returned. Note that the integration with this function requires correct passing of arguments.

Leveraging the functionality described above, an optional shares splitting contract is introduced that could be specified as the fee recipient to allow splitting fees among multiple users at constant ratios.

To implement this logic a more generic abstract contract TreasurySplitterMixin was implemented. Per token the totally claimed fee amount of all users and per user are tracked. Each specified user has a percentage share. It exposes the following functionality to its descendants:

- External functions `claimTokens()` and `claimTokenAmountTo()`: The first one claims the full amount claimable of the token to the `msg.sender` according to his share. The second claims a given amount, if possible, to a predefined recipient. Note that both utilize `__claimTokens()` that is also exposed to descendants that contains the generic logic for both externally exposed functions.

- Internal function `__claimTokenWithoutTransfer()` which is the claiming logic without the transfer.

- External getters to get for a given token the claimed fees per user and totally and the split ratio per user. Internal and external getter to calculate the claimable amount for a user.
- `__setSplitRatio` that writes the percentages for a given set of users.

Given the mix-in, a more derived SharesSplitterLib contract is implemented. It allows redeeming shares, which could be non-transferrable, through a chosen method. However, it requires validation on the global config proxy contract. It implements a function `redeemShares()` by claiming without a transfer, validating the redemption call (with optional validation on the redemption amount activated) and finally redeeming shares.

Note that the the shares splitter is a proxy contract which extends `NonUpgradeableProxy` without any modifications (see previous audit reports). A factory SharesSplitterFactory offers a `deploy()` function to deploy a proxy that references the SharesSplitterLib as its implementation.

2.2.4 ParaswapV5 adapter changes

The following changes have been made to the ParaswapV5 adapter (see previous reports):

- The adapter allows deployment to specify the fee partner and the fee percent. The code has been adapted such that calls to ParaswapV5 do not have fixed arguments `0x0` and `0` for those calls to have a more flexible architecture.
- Now, `multiSwap` is used instead of `protectedMultiswap`.

2.2.5 Convex vote-locked CVX external position

Holders of CVX can lock their tokens into the CvxLockerV2 of Convex finance to receive vlCVX (non transferrable with ERC-20 functions). Note that the CVX tokens will remain locked for 16 epochs. vlCVX holder are eligible to vote on Snapshot under the `cvx.eth` snapshot id and to claim CvxCRV and other extra rewards.

A new type of external positions will enable vaults of Enzyme to lock CVX tokens, to relock them after they are unlocked, to withdraw the CVX tokens once they are unlocked, to delegate their votes to a delegate, and to claim rewards. Additionally, the position can claim rewards from Votium (if the delegation has been made to Votium).

After such an external position is opened, the following actions are available:

- `Lock`: locks CVX into the vlCVX locker contract.
- `Relock`: relocks all unlocked CVX in the vlCVX locker contract.
- `Withdraw`: withdraws all unlocked CVX in the vlCVX locker contract to the vault proxy.
- `ClaimRewards`: Optionally claims rewards CVX from the locker, optionally claims the extra rewards from the extra rewards contract, optionally claims rewards from Votium, optionally withdraws from the CvxCrv staking contract (since other users could claim rewards and stake for the position), and transfers the full balance of a given array of tokens to the vault proxy. Since there are no checks on the token addresses, tokens could remain in the position.
- `Delegate`: Delegate the position's voting power to another address.

Since no debt is taken, `getDebtAssets()` will be empty. `getManagedAssets()` will return the sum of the locked CVX and the currently held CVX.

2.2.6 Liquity borrowing

Liquity is a permissionless DeFi protocol offering interest-free borrowing where ETH can be provided as collateral to mint the LUSD stablecoin. However, a one-time fee is paid when borrowing. Users interacting with the system have a so-called active trove on which they operate. Most notably, Liquity has two execution modes. The first one is the normal mode where the minimum collateralization ratio is 110%, while the second one is the recovery mode - active when the global collateralization falls below 150% - where troves can be liquidated if the collateralization ratio drops is below 150%.

A new type of external positions will enable vaults of Enzyme to borrow LUSD on Liquity. For an overview of how external positions work in Enzyme, please refer to section 2.2.2 External Positions of the system overview of the Sulu report.

After opening such an external position for a fund, the following actions are available to interact with Liquity:

- OpenTrove: Opens a trove on Liquity by depositing ETH and withdrawing LUSD.

- AddCollateral: Adds ETH to the active trove as collateral.

- RemoveCollateral: Removes ETH from the active trove.

- Borrow: Draws more debt from the active trove by withdrawing LUSD.

- RepayBorrow: Repays some LUSD debt.

- CloseTrove: Repays all LUSD debt and withdraws all ETH from the trove. The trove becomes inactive.

`getDebtAssets()` will return the LUSD debt. `getManagedAssets()` will return the ETH held by the trove.

2.2.7 The Graph Delegations

The Graph is a protocol for indexing and querying where indexers are node operators in the network that stake Graph Tokens in order to provide indexing services. As an incentive, indexers earn rewards from query fees and indexing rewards. Other parties, namely delegators, can receive a share of the fees earned by indexers by delegating to the corresponding indexers.

A new type of external positions will enable Enzyme vaults to delegate to several indexers on The Graph. After such an external position is opened, the following actions are available:

- Delegate: Delegate an amount of The Graph tokens to an indexer.

- Undelegate: Undelegate an amount of pool share from an indexer. Tokens remain locked in the staking contract for the unbonding period. However, tokens from earlier undelegations (for that indexer) could be unlocked.

- Withdraw: Withdraw The Graph from a delegation once the unbonding period has passed and the tokens are unlocked.

Since no debt is created, `getDebtAssets()` will be empty while `getManagedAssets()` will return the sum of all locked tokens and all actively delegated tokens.

Note that the unbonding period is documented to be 28 days. For further details, see The Graph documentation.

2.2.8 Changes in V2

For the Curve integrations and the Curve price feed following changes have been made:

- `updateValidatedVirtualPrices()` was removed as validation was always triggered by `calcUnderlyingValues` if necessary.

- Functionality to handle a possible pool ownership transfer was added.

2.2.9 Changes in V4

Avantgarde Finance reported an issue with the Liquity EP. The issue was mistakenly included in another report and the error went unnoticed. V4 introduces a fix to the issue by introducing an action `ClaimCollateral` that calls `claimCollateral()`. Further, it modifies the valuation by accounting for the unclaimed collateral (balance available) in the collateral surplus pool.

2.2.10 Trust Model and Roles

Please refer to the main audit report for a general trust model of Sulu.

The Fund Manager is fully trusted given the reasons detailed in the previous reports. Note that with the addition of the time-locked tokens for some external positions, e.g. vote-locked Convex and The Graph tokens, the Fund Manager yields even greater power over funds. Hence, the fund managers are fully trusted to keep the funds balanced in such a way that users can withdraw their shares.

All external systems are expected to be non-malicious and work correctly as documented. Fund managers are expected to not only behave honestly but also to understand the systems they are interacting with. This includes choosing appropriate parameters, e.g., for the slippage protection.

For actions on Liquity troves we expect the fund manager to keep the collateralization ratio above 150% at all times.

For pauseable systems such as Maple, we assume that they will be paused only temporarily.

In general we assume Enzyme only interacts with normal ERC-20 tokens without any special behavior including rebasing, multiple entry points or callbacks. This assumption also applies for for the TreasurySplitterMixin of the SharesSplitter.

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 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:

- 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	0

Low -Severity Findings	0

6 Resolved Findings

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

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

`Critical`-Severity Findings	0

`High`-Severity Findings	1

- Conversion Errors When Computing Underlying Graph Token Value `Code Corrected`

`Medium`-Severity Findings	2

- Incorrect Argument Order for lowerHint and upperHint `Code Corrected`
- Liquity: Lack of Support for claimCollateral() `Code Corrected`

`Low`-Severity Findings	4

- Lending Pools Array Read Twice From Storage in getManagedAssets `Code Corrected`
- Pool Owner Could Change `Code Corrected`
- LendAndStake Stakes the Full Balance `Code Corrected`
- __curveGaugeV2GetRewardsTokensWithCrv Is Unused `Code Corrected`

6.1 Conversion Errors When Computing Underlying Graph Token Value

`Correctness` `High` `Version 1` `Code Corrected`

Each delegation contains two values relevant for the totally managed assets per indexer: the delegated amount denominated in shares and the locked amount denominated in The Graph tokens. To compute the value in The Graph tokens assigned currently to an indexer pool, `getDelegationGrtValue()` implements the following logic:

```
(uint256 delegationShares, uint256 tokensLocked, ) = GRAPH_STAKING_CONTRACT.getDelegation(
    _indexer,
    address(this)
);
(, , , , uint256 poolShares, uint256 poolTokens) = GRAPH_STAKING_CONTRACT.delegationPools(
    _indexer
);
if (delegationShares > 0) {
    return delegationShares.mul(poolTokens).div(poolShares).add(tokensLocked);
}
return tokensLocked;
```

Note, however, that the `view` function `delegationPools` returns the following struct:

```
struct DelegationPool {
    uint32 cooldownBlocks; // Blocks to wait before updating parameters
    uint32 indexingRewardCut; // in PPM
    uint32 queryFeeCut; // in PPM
```

```
    uint256 updatedAtBlock; // Block when the pool was last updated
    uint256 tokens; // Total tokens as pool reserves
    uint256 shares; // Total shares minted in the pool
    mapping(address => Delegation) delegators; // Mapping of delegator => Delegation
}
```

The `poolShares` return value corresponds to the `tokens` value in the struct (similar for `poolTokens`). Hence, the return values are not used correctly when `delegationShares > 0` holds since the shares' value in GRT will be computed with the inverse of the actual exchange rate.

The tests leave this issue undiscovered since for the delegation pool used in the test case `tokens` equals `shares` which hides the issue.

Ultimately, `getManagedAssets()` will incorrectly estimate the position.

Code corrected:

The `poolTokens` and `poolShares` values are now retrieved in the correct order from `delegationPools`.

6.2 Incorrect Argument Order for `lowerHint` and `upperHint`

Correctness   Medium   Version 1   Code Corrected

In `LiquityDebtPositionLib.sol`, the order in which the `lowerHint` and `upperHint` arguments are passed is incorrect in several locations.

This includes private calls (from `receiveCallFromVault` to the action in question), external calls (from the action to `ILiquityBorrowerOperations`). Furthermore, there are mix-ups in the tests as well.

Passing the hints in the wrong order generally does not result in the call to Liquity to fail. However, as the hint is unusable, the execution spends more gas to find the right location.

Below is a summary of whether these function calls are made with a correct argument order for each action within the smart contract:

- `__openTrove`:

    - Private call: Incorrect
    - External call: Incorrect

- `__addCollateral`:

    - Private call: Correct
    - External call: Incorrect

- `__removeCollateral`:

    - Private call: Correct
    - External call: Incorrect

- `__borrow`:

    - Private call: Incorrect
    - External call: Correct

- `__repayBorrow`:

  - Private call: Incorrect

  - External call: Correct

The convention that Liquity seems to follow is to pass `upperHint` before `lowerHint`. In Liquity's `SortedTroves.sol`, a different naming (`_prevId` and `__nextID`) is used. Since troves are sorted in descending order, `_prevId` actually corresponds to `upperHint`. This is inconsistent with how hints are interpreted/named in `LiquityDebtPosition.test.ts`. In some test cases hints are in switched as well.

One example is the implementation and the test case for `repayBorrow`: The arguments are switched in the smart contract code and in the corresponding test. Two wrongs make a right and the hints are passed correctly.

When an uneven number of such mistakes are made, the hints are useless and the gas consumption of the call increases.

After switching all the hints arguments in the tests, the gas consumption of the above actions is as follows:

- `__openTrove`: Higher (761598 vs. 748240)

- `__addCollateral`: Higher (516155 vs. 417178)

- `__removeCollateral`: Lower (549037 vs. 562395)

- `__borrow`: Lower (1181526 vs. 1194884)

- `__repayBorrow`: Higher (490531 vs. 391554)

Overall, the arguments `upperHint` and `lowerHint` should be rechecked and corrected everywhere to ensure useful hints are passed to Liquity and gas used is minimized.

Code corrected:

Hints are now always passed in the same order, i.e., `upperHint`, `lowerHint`. The code was also improved to more explicitly identify these two arguments. Furthermore, the tests were updated to use a collateralization ratio that will avoid placing the trove at an extremity of the sorted list to validate the correct passing of hints.

6.3 Liquity: Lack of Support for `claimCollateral()`

Design Medium Version 1 Code Corrected

The technical documentation of Liquity writes the following:

```
claimCollateral(address _user): when a borrower's Trove has been fully redeemed from and closed,
or liquidated in Recovery Mode with a collateralization ratio above 110%, this function allows the
borrower to claim their ETH collateral surplus that remains in the system (collateral - debt upon
redemption; collateral - 110% of the debt upon liquidation).
```

However, the Liquity position library does not support such an action. Hence, it could be possible that funds could become stuck in some situations (e.g. liquidations in recovery mode).

Code corrected:

The issue has been resolved by adding a `ClaimCollateral` action and including the unclaimed collateral in the valuation.

Note:

Note that this issue has been inadvertently included in another report previously.

6.4 Lending Pools Array Read Twice From Storage in `getManagedAssets`

`Design` `Low` `Version 1` `Code Corrected`

In `MapleLiquidityPositionLib.getManagedAssets()` the length of the used lending pools is queried by copying the full array from storage into memory. Next, the `for` loop iterates over the array and reads the elements from storage. Hence, gas consumption could be reduced by caching the array in memory.

Code corrected:

The pools are now cached into memory and no longer read from storage repeatedly.

6.5 Pool Owner Could Change

`Design` `Low` `Version 1` `Code Corrected`

To prevent the manipulation of a pool, reentrancy is checked through the `withdraw_admin_fees` function of the pool owner contract. However, the pool owner address could change and hence such calls on the pool could fail not due to reentrancy but due to access control. The price feed stores the address as an immutable and, thus, could become unusable in the aforementioned scenario of changing ownerships.

Code corrected:

The pool owner address is not immutable anymore. Now, it can be changed by governance.

6.6 `LendAndStake` Stakes the Full Balance

`Design` `Low` `Version 1` `Code Corrected`

`LendAndStake` is an action wrapping the lending and the staking action. First, it lends an amount of liquidity assets to the Maple pool. Next, it stakes LP tokens to the rewards contract to get some extra rewards:

```
function __lendAndStakeAction(bytes memory _actionArgs) private {
    (
        address pool,
        address rewardsContract,
        uint256 liquidityAssetAmount
    ) = __decodeLendAndStakeActionArgs(_actionArgs);
```

```
    __lend(IMaplePool(pool).liquidityAsset(), pool, liquidityAssetAmount);
    __stake(rewardsContract, pool, ERC20(pool).balanceOf(address(this)));
}
```

The argument passed to the internal `__stake` function is the full balance of the pool token. Note that it is also possible to lend the underlying without staking. Consider now the following scenario:

1. 100 tokens are lent into the pool. 100 LP tokens are received.

2. Later, lend and stake is used with 100 underlying tokens.

3. The full balance, namely 200 LP tokens, will be staked.

Such behavior could be unexpected for fund managers.

Code corrected:

The code of `__lendAndStakeAction()` has been changed and now only stakes the amount of tokens received.

6.7 `__curveGaugeV2GetRewardsTokensWithCrv` Is Unused

`Design` `Low` `Version 1` `Code Corrected`

The internal function `__curveGaugeV2GetRewardsTokensWithCrv` is unused and could be removed to reduce deployment cost.

Code corrected:

The function has been removed.

7 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.

7.1 Derived Contracts Could By-Pass the Invariant on the Sum of Shares

`Note` `Version 1`

The TreasurySplitterMixin is an abstract contract that allows splitting funds at constant ratios (which should sum up to 100%) among users. The only possibility to modify the split ratio in a more derived contract is through the internal function `__setSplitRatio`.

```solidity
function __setSplitRatio(address[] memory _users, uint256[] memory _splitPercentages)
    internal
{
    uint256 totalSplitPercentage;
    for (uint256 i; i < _users.length; i++) {
        // ... duplicate and non-zero validation
        userToSplitPercentage[_users[i]] = _splitPercentages[i];
        totalSplitPercentage = totalSplitPercentage.add(_splitPercentages[i]);
        emit SplitPercentageSet(_users[i], _splitPercentages[i]);
    }
    require(totalSplitPercentage == ONE_HUNDRED_PERCENT, "__setSplitRatio: Split not 100%");
}
```

This function is agnostic to the current storage of the contract. Hence, the following scenario could occur:

1. A more derived contract sets the split ratio with `__setSplitRatio` to 100% for user A. Hence, `userToSplitPercentage` for A will be 100% while no invariants are violated.

2. In another step, the more derived contract tries to add user B to the sharing mechanism. It passes only user B and 50% to the function.

3. Now, the `userToSplitPercentage` is set to 50% for B.

4. The sum of all user split percentages is 150% which violates the invariant.

Hence, the current implementation is only suited for one-time setting of split ratios.

With the current usage, this is not an issue as the shares splitter contract will set the ratio only once upon creation. However, future contracts inheriting from the TreasurySplitterMixin could require some additional logic to prevent the invariant violations described above.