PUBLIC

Code Assessment

of the Ripe Finance Smart Contracts

January 17, 2024

Produced for

by

CHAINSECURITY

Contents

1 Executive Summary

Hightop Financial implements an investment platform that additionally allows borrowing their stablecoin called Green (jUSD before Version 4 of the code). Ripe acts like a bank. Users can deposit their assets into strategies. Strategies could be a simple vault-like strategy or an investment in e.g., a Yearn v2 vault. These deposits can be used as collateral to borrow Green. To keep Green stable, Ripe implements multiple mechanisms described in the System Overview. Additionally, Ripe rewards users with Ripe tokens and Green as well as an internal loyalty program with so-called juice points (providing users with better conditions).

Overall, the number of issues relative to the size of the projects found is relatively moderate and the communication with the team was always excellent. However, the project is one of the biggest vyper projects and the audit was spread over a very long time. Hightop Financial implemented one of the most through testing and development practices we have seen. Yet, due to the size of the project and the many iterations, remaining issues might be present and we recommend to carefully roll-out and launch the product.

The most critical subjects covered in our audit are peg stability, gas consumption and system solvency. In Version 3 of the code, we focused on new governance system.

Ripe implemented many tools to maintain the peg. However, some of the stability mechanisms rely on the actions of privileged actors and parameters set by the governance. This might lead to a situation where the peg is hard or impossible to maintain due to the nature of the events. The system should always be monitored accordingly and adjusted if necessary.

Many issues discovered during the audits are related to high gas consumption and possible denial-of-service scenarios. Most of them were fixed to ensure, that the actions of users do not compromise critical system functions such as liquidations. However, individual users can still encounter high gas costs and potentially DoS themselves.

The security regarding system solvency is high. The system is designed to be over-collateralized and has a liquidation mechanism in place. However, the system relies on governance actions and parameters set by the governance to maintain the USD to Green peg. E.g., governance should not emit more value in bonds than the system can handle.

Green stablecoin peg stability is achievable, but the system should be monitored and adjusted if necessary.

The system is highly interconnected, and logic is separated into many different contracts. Call-paths are complex and hard to follow. It is hard to formulate invariants due to many cross-calls and storage operations in many different contracts. All this makes the system hard to audit (especially, regarding invariants for certain contracts) and maintain. When operating the system many parameters can be set and the implications and dependencies might not be obvious immediately. As mentioned in the beginning, we would always recommend extensive testing and a careful roll-out of the product.

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	4
• Code Corrected	4
High-Severity Findings	7
• Code Corrected	6
• Risk Accepted	1
Medium-Severity Findings	14
• Code Corrected	13
• Risk Accepted	1
Low-Severity Findings	30
• Code Corrected	25
• Specification Changed	2
• Risk Accepted	3

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 Ripe Finance repository based on the documentation provided by Ripe on notion. The scope of the audit is limited to the smart contracts within the `contracts` subfolder, if not mentioned in Excluded from scope. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	10 Jan 2024	aeb3ba4d563a3a83c6a8e1e01d858645db8ab7dc	Initial Version
2	17 Mar 2024	7840cf17349c3c5c477c2d2fbace83b0b1aa9836	Version with fixes
3	10 Jun 2024	d791f645ccc87df47e5dfe310a0a73c72f5dd188	New governance system
4	24 Jun 2024	ae9928da90f1436ab2221fe6edfebf0f83b11a48	Renaming and refactoring
5	15 Jul 2024	a7ca2785ada23c8bb021dc661f404a1e156c1177	Fixes and refactoring
6	23 Jul 2024	3346b0faaea774b6e0d4fa72204fed76981bc476	Fixes
7	06 Aug 2024	7138c5d583d482584f824d6cae9c04193edc4ebf	Fixes
8	03 Sep 2024	2041ba7d97c7ca84a15a9e81c98296f6cbf5b8e0	Fixes
9	17 Jan 2025	98a2099d6a732e9fd7cd52e03ff25de9cfdbaacc	Fixes

For the Vyper smart contracts, the compiler version `0.3.10` was chosen.

2.1.1 Excluded from scope

- All files outside of the `contracts` subfolder
- All files in `contracts/mock`
- All files in `contracts/config`
- Economic model of the project. The sanity of parameters is out of scope.

This assessment did not include an economic modeling analysis, such as testing the sanity of parameters or conducting economic stress tests with scenarios simulating price changes or other economic fluctuations. Ripe system heavily depends on parameters and their relevance. Overtime the parameters will need to be updated to reflect the changes in the market. We assume all parameter updates to be done correctly and tested sufficiently.

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.

Ripe finance is a Defi lending and investment protocol allowing users to take loans in Ripe's stable coin jUSD (pegged to USD). Users can deposit and allocate their funds into various investment strategies and special strategies (e.g., governance and stability pools). Some deposits can be used as collateral, while earning yield, to take out a loan in jUSD. jUSD is an algorithmic stablecoin that tries to improve capital efficiency. Like most DeFi loans, the borrowed funds are over-collateralized. Thus, users can only borrow up to a specified fraction of the collateral value. E.g., 100 token collateral worth 1000 USD might only allow the user to take a 700 USD loan. Ripe's jUSD stablecoin is backed by the collateral of the users as it is minted when users borrow it. Additionally, jUSD will be minted against protocol profits. Ripe took various measures to stabilize their stablecoin and improve capital efficiency. Through this system, Ripe acquires additional funds to back jUSD and secure the overall system. As the major share of jUSD's value is backed by the collateral that users provide, a critical part is the efficient liquidation of unhealthy user debt positions.

2.2.1 The Headquarters

Ripe's contracts are structured like a bank with different departments and a board. The departments', the board's, the jUSD's and RIPE's addresses, and some top-level configs are managed in the `Headquarters` contract. It is the core and definition of the protocol as it stores the addresses of the departments and the BoardRoom contract and cannot be replaced/upgraded. It is the only contract that is allowed to mint new Ripe emissions or jUSD (when `mintRipe` or `mintJusd` is called). Top-level access belongs to Chairman — the primary bootstrapping governance wallet/contract that can update these contracts. The chairman's responsibilities can be transferred to the BoardRoom contract. The Headquarters contract stores and has the setters for the boardroom contract (`setBoardRoom`), the Departments contract (`setDepartments`), the jUSD stablecoin (`setJusdToken`), the Ripe token (`setRipeToken`) and the chairman change with a commit accept scheme (`acceptChairmanship` and `setPendingChairman`). All setters are permissioned and can only be called by the `chairman`. Additionally, the contract can be paused by eligible boardroom members by calling `setIsPaused`. This set the `isPaused` flag to `true` and paused most of the interaction with the protocol. Another flag `isActivated` can be set to disable the `Headquaters` contract itself from executing most actions (Chairman change, minting, setting new department or board room). The last callable function is `didUpdateBoardRoomOrDepts` to broadcast a new department or board room address.

2.2.2 The BoardRoom

The `BoardRoom` is a central part of Ripe's access permission. It holds the view function to check permissions on critical operations that subdaos and board members are allowed to execute.

2.2.3 Departments

The Departments contracts hold Ripe's internal system addresses e.g., the address of the Teller and CreditUnion. It also checks the minting rights of certain system contracts. It is used for the automated part of the system as access control in this case whereas the Boardroom checks for external accounts or smart contracts the permissions.

The omnipresent `didUpdateBoardRoomOrDepts` in the ripe system would update and propagate new addresses through the Departments and the BoardRoom contracts by querying the immutable Headquarters address for the Departments and BoardRoom address and from there ask the registries to get all their potential child contacts and propagate the address changes to them.

2.2.4 Further Registries

Besides the BoardRoom which stores external addresses with certain permissions and the Departments that store the system's addresses, four other contracts allow the management of external and internal needed addresses:

- The PriceDesk: It manages all price feeds the system can use.
- The RoloDex: It manages the DEX integrations that the system can use.
- The StaratMap: It manages the strategies the system can use. E.g., it holds the addresses for the stability pool, governance and yearnV2 strategies users can deposit in.
- HumanResources: Manages system contributors. It is split into three contracts. A config contract `HumanRessourcesConfig` where e.g., vesting, cliff and compensations are set. `HumanRessources` provides the logic to add new entries and manage them. The `Contributors` contract has the logic for contributors to interact with the system and receive their compensations.

All four registries use the `PhoneBookData` contract for storing their state.

2.2.5 The Endaoment

The Endaoment manages the protocol's funds and tries to keep jUSD stable while generating profits for Ripe through investing, liquidations, and stabilization measures. To facilitate this the following features are present:

- Users can swap tokens for jUSD
- Users can buy discounted jUSD through bonding
- The Stablizer is acting on exchanges (stabilizer pools) to keep DEX rates pegged and generate profits
- Invest in whitelisted strategies (including the stability pool)
- Offer jUSD redemptions

The amount of Endaoment assets available to invest is the net of Endaoment inflows and outflows. For jUSD to be invested, it first needs to be swapped to Tokens via the speed-swap mechanism. To keep jUSD stable the Endaoment burns jUSD. The Endaoment collects assets and revenue from:

- The yield generated by the Endaoment strategies. Endaoment Assets can be invested to generate yield if:
    - Governance has agreed to do so
    - An underlying strategy integration that can generate yield exists
- The bonding actions (see Treasury)
- Liquidations fees (See Liquidations)
- Borrowing (origination) fees
- Borrow rate (interest)
- Endaoment redemption fees

The measures the Endaoment can take to stabilize jUSD are:

- Seizing funds in liquidation
- Sell discounted jUSD over bonding (Ripe token bonding is also possible).
- Collecting other tokens by allowing so-called speed swaps (users can swap jUSD for tokens the Endaoment accepts with no fees).

- EndaoReduce: A mechanism allowing users to redeem their jUSD for positions held by the Endaoment (if enabled - usually only in stressed situations).
- Directly act on exchanges via the stabilizer contract

2.2.6 Stability Pool

The stability pool holds funds to buy collateral and, therefore, helps to stabilize jUSD. In the case of fungible token liquidation, it will be the first measure to liquidate a loan and buy the collateral. The stability pool is implemented as a special strategy. Tokens deposited into the stability pool strategy are considered staked and are eligible for rewards. These rewards come from:

- The purchase of liquidated assets at a discount
- jUSD Endaoment Staking Rewards: a specific % allocation of the yield generated from the Endaoment.
- RIPE Rewards from the total RIPE supply
- Ability to participate in Token and NFT liquidation auctions (only users who staked can participate in auctions)

To acquire the rewards of the stability pool (jUSD, RIPE and liquidated assets), users need to claim them. When claimed, purchased liquidation assets are automatically deposited into the strategies they were liquidated from. Users need to claim their share actively (also possible via the Janitorial).

In case of a liquidation, the stability pool will buy the collateral assets position with jUSD or stable whitelisted assets (assuming 1 stable token equals 1 USD) at a discount. The position of the user will not be made solvent by taking it out of the strategy but the position in the strategy will now be owned by the stability pool. The stables will be paid into the Endaoment wallet, to back the jUSD still out in fluctuation.

- Collateral assets purchased can then be claimed by the stability pool stakers relative to their stability pool contribution.
- When claimed, the user will own the position. The underlying asset is never withdrawn from the strategy in this process. The ownership of the asset in the strategy simply changes.
- All Ripe and jUSD rewards allocated to stability pool users need to be claimed separately in the Lootbox.

Ripe's contracts interact with the stability pool mainly when credit is repaid, or funds are used to buy collateral in liquidation over the auction house smart contracts. In case of liquidations, the stability pool's jUSD funds are used to buy fungible collateral assets that are in liquidation, generating profits for the stability pool.

2.2.7 Treasury

The Treasury is another part of the system, which should help to stabilize jUSD by providing funds for liquidations. This measure provides another lever for Ripe to finance the Endaoment without increasing jUSD liquidity and adding buy pressure. Unlike the stability pool's funds, these funds are not staked user funds but are owned by the system. The system acquires them by offering a limited amount (Bond Capacity) of discounted vested jUSD (in the future maybe also Ripe and other tokens but the assessment was limited to jUSD and Ripe bonds) in exchange for specific assets (Bonded Assets) for a predetermined period (Epoch Time) to users. Ripe calls this bonding as the users will need to pay immediately but can only claim the vested jUSD later (specified by the Vesting Period). The Janitorial will claim bond proceeds for a user if desired. The treasury is closely coupled to the Endaoment as bonding is a way to finance the Endaoment liquidity and is theoretically independent of the overall protocol functioning. The bonding terms are specified by the needs of the protocol to grow the Endaoment that seeks to maximize assets and protocol-owned liquidity while accounting for costs and constraints. Different dynamics are at play such as what assets Endaoment wants, the capacity it's willing to sell, and the maximum discount it is ready to offer. All are based on jUSD liquidity and peg constraints, jUSD demand, its risk profile, and portfolio management strategy. The bonding process is designed for pricing/discount to be completely market-driven, with competitive dynamics leading buyers to have to balance price & availability. The jUSD discount varies throughout the Epoch, going from a Min Bond Discount (0%) to a Max Bond Discount (eg. 10%). Additionally, Ripe rewards are given to further incentivize purchasing jUSD, and these also increase over time.

2.2.8 Governance & Gov Strategy

Ripe has decentralized governance that users can participate in by staking Ripe or Ripe LP tokens into a dedicated governance strategy `RipeGovStrat`. Overtime, users locked Ripe tokens will receive power, accounted in `RipreGovStratPoints` contract. At any moment, users can delegate their voting power to another user, effectively increasing the voting power of the delegate, while reducing their own. Having staked into the governance strategy will reward a user with Ripe and jUSD token rewards. At the time of the assessment, the Ripe government can vote on the following parameters:

- Deposit Rewards:

  Users vote on the Ripe and jUSD allocation for all users who deposited funds into Ripe.

- Ripe Rewards

  Vote on the allocation of the new Ripe token emission between stakers, borrowers, vote depositors and general depositors.

- Endao Yield

  Vote on the allocation of the Yield earned by the Endao and how it is distributed between vote depositors, general strategy depositors and stakers.

Users will have a voting power based on their stake that is defined by a function of the time they staked. According to Ripe, building the right governance system is more an art than a science and will be a continuous and iterative process. Changes might be upcoming regularly and not be part of the code assessed in this assessment.

2.2.9 Investing/Strategies

Users can deposit their capital into Ripe and invest it into different strategies and/or borrow jUSD by using their deposit as collateral. Deposits are handled by the Teller department. The Governance strategy and Stability Pool strategy are exceptional and covered in other parts of the system overview in more detail. Ripe wants to support a wide range of tokens including NFTs and RWAs to be deposited, invested, and used as collateral to borrow jUSD. Even assets that are not allocated to a strategy will earn yield from the Endaoment deposit rewards. Ripe will offer different strategies some use third-party protocols like Yearn or custom Ripe strategies. The main functionalities that strategies need to support are depositing, withdrawing, position transfers and providing an accurate estimation of their fund's value.

Currently, there are six strategies implemented (besides the governance and stability pool strategy):

- floorErc21

  This strategy is for ERC271 NFTs. No investment is done. The strategy simply is a vault holding the users' NFTs. The user deposit is valued at the lowest oracle price for the collection (if Chainlink is used).

- GranularErc721

  Same strategy as floorErc21 but it tries to value the NFT individually based on the id in the collection.

- simpleErc20

  The simple ERC20 strategy is a trivial vault that holds the deposited funds but does not invest them anywhere.

- rebaseErc20

  The rebase ERC20 strategy is the same as the simple ERC20 strategy but for rebasing tokens. The compatibility of a rebasing token needs to be evaluated and tested carefully before using it.

- yearnV2

  The yearnV2 strategy acts as an adapter a vault for the yTokens. Funds a user allocates to this strategy are forwarded and invested into the corresponding Yearn v2 vault. The user will be credited a share of the strategy's yearn shares. When withdrawing from the strategy, the tokens will be withdrawn from the yearn v2 vault, too.

- yieldErc4626

  This strategy is a generalized strategy to invest in ERC4626 vaults by using the standard interfaces.

2.2.10 Borrowing, Repaying and Redeeming

After depositing funds, users can use these funds (if the deposited asset is supported) as collateral to borrow jUSD (up to a maximum borrow capacity and a user-specific limit). Users need to pay a borrow fee (origination fee) and dynamic interest (borrow rate) on the borrowed jUSD. Over time the user's debt (jUSD) will increase based on the borrowing rate. To avoid liquidation, users need to keep their debt positions healthy (maintain a predefined minimum debt-to-collateral ratio / LTV). The Credit department handles the borrowing process. Users can have deposits in multiple strategy-asset combinations and use them as collateral. The borrow terms are individually set for each asset in each strategy separately and aggregated to check the health of a user. The aggregated borrowing terms are calculated by computing the weighted average of the borrowing terms of each strategy-asset combination. The `LTV * collateralValue` is used as a weight. Borrow-specific parameters are stored and managed in the Communal department. These parameters include e.g. the loan to value (LTV), liquidation threshold, liquidation fee, redemption threshold and borrow rate for the asset in a specific strategy. E.g., the LTV defines the maximum amount of jUSD a user can borrow against their collateral.

Users can always repay their jUSD loan (incl. borrow rate). Their collateral will stay managed by a strategy in the name of the user while having the loan and after repaying as long as the user does not withdraw it from the strategy or gets liquidated. Repayments are also handled in the credit department. A user can repay at any time. Even when the funds are up for auction because of a liquidation.

The collateral deposits of users that have not yet reached the liquidation threshold but do have an unhealthy position that reached the redemption threshold, can be redeemed by other users. A redemption fee needs to be paid by the reedemer/buyer of the collateral. The buyer will buy the collateral from the user and pay the value of the redeemed collateral in jUSD to the Endaoment.

In case the borrower's position falls under the liquidation threshold, the user can be liquidated. Liquidations are managed by the AuctionHouse contracts.

2.2.11 Liquidations

Redemptions try to lower the risk of debt to be liquidated. But in case a user's debt is too high (liquidation threshold reached), the debt can be liquidated. Liquidations differ between fungible tokens and non-fungible tokens. Non-fungible collateral will be auctioned. To bid on an auction the bidder needs to have the bidding value staked in the stability pool. In case a user wins an auction, invested funds are not withdrawn but the position of the strategy asset combination changes the owner. Fungible collateral will be liquidated in different steps:

1. The Stability Pool tries to buy out the collateral with its funds.
2. If the stability pool does not have enough funds to liquidate the position, the Endaoment tries to buy the collateral with its jUSD reserve.
3. If the Endaoment cannot cover the liquidation process, the collateral is auctioned via Ripe's fungible token auction. This auction is based on a Dutch auction which decreases the price over time until some buyer decides to take it.

Liquidations are handled by the so-called Auction house smart contracts. To trigger the liquidation of a user, an account calls `AuctionHouseCollections.liquidateUsers`. The auction mechanisms (like bidding and auction management) are provided by the contracts `AuctionHouseNFT` and `AuctionHouseFungible` for the respective asset type.

2.2.12 Rewards

The system distributes Ripe emission (time-dependent) and jUSD rewards (based on the Endaoment profits) back to the users. In the case of Ripe emissions, there is a fixed emission rate that is distributed over time. In the case of jUSD rewards, first, the Endaoment profits need to be determined. The protocol funds are held by the Endaoment wallet. The Endaoment wallet's funds are invested in a set of approved strategies. In case these strategies generate profits, a matching amount of jUSD is minted and distributed as a reward for the profits. In case the Endaoment has jUSD at the point in time, when the respective function (`moneyPrinterGoBrr`) is called, parts of this jUSD might be burned instead of distributed and minted to the users. The protocol rewards are mainly managed and distributed by the Lootbox smart contracts logic. Distribution is divided into two stages. First, between the different stakeholders (depositors, voters, etc.). For each user group, there is a share that can be set by a governance voting process. The second stage is the share of a specific user in such a group. The users' share depends on so-called Lootbox points. Lootbox points essentially reflect the time and amount the user's funds are invested in Ripe. Last but not least Lootbox contracts include the functions users can call, to claim their rewards.

Ripe Rewards can be distributed to:

- stakers (i.e. jUSD, RIPE or corresponding LP tokens)
- borrowers
- general depositors (distributed based on deposit USD value)
- vote depositors (distributed based on governance/voting)

Jusd Rewards (endaoment yield) is distributed primarily to:

- impact subdao
- ops subdao
- stakers
- general depositors
- vote depositors

2.2.13 Juice Score

The juice score is a different reward system compared to the Lootbox. The juice score resembles a loyalty reward program that gives users better conditions when performing specific protocol interactions. It is a score between 0 and 100 and allows perks such as:

- Lower Borrow rate
- Improved LTV
- Lower Origination fee
- Higher Bond discount
- Lower Liquidation fee
- Lower Swap/Redemption fee

The juice score is cumulative and depends on the total value of the action and the action itself. Actions that can increase a user's Juice Score are e.g.:

- Borrowing: total Outstanding Debt
- Bonding: total Bonds Purchased recently (decay time)
- Staking: total jUSD + RIPE staked balances

2.2.14 Communal Data

The communal data contracts store important system configuration parameters e.g., the debt terms, auction configuration, allow lists and asset configurations.

2.2.15 Janitorial

The system has a so-called janitorial contract. It triggers important system updates and allows (if desired) a user to simultaneously claim rewards in a transaction.

2.2.16 Ledger

The Ledger department of the system is responsible for the accounting of the system. It tracks the system's assets, borrowing safety data (SafetySlips) and the strategies users are participating in.

2.3 Version 2 changes

Numerous changes were made to the system. The most significant changes are:

- the `AuctionHouseNFT` locks the funds of the best bidder in `AuctionHouseEscrow` contract. If the bid was in jUSD, the jUSD is deposited for the bidder and a position is transferred to them. In case the bid was in another stable asset, the asset will be sent to the `EndaoLiqWallet` and jUSD will be minted for it. This jUSD is, again deposited for the user as before.
- `LedgerLocker` contract was added to Ledger department. Each `StratEngage` contract now tracks the user's unlock time. `LedgerLocker` contract can compute the updated values for the users unlock time and combine them.

2.4 Version 3 changes

The Governance system received a major overhaul.

2.4.1 New Governance overview

As before, users can stake Ripe or Ripe LP tokens into the governance strategy `RipeGovStrat` contract. Over time, users locked Ripe tokens will receive power, accounted in `RipreGovStratPoints` contract. As before, users can vote on reward allocations in dynamic voting contracts (e.g. `DynGovEndaoPositions` and `DynGovLootAssets`). But in addition to the dynamic voting, the new governance system supports creating and voting on proposals that allow to change most of the system parameters.

2.4.1.1 Proposals

Each proposal in general has the following stages:

- Pending: Proposal is created and waiting for the voting to start
- Voting: Proposal is in the voting stage
- Weight Adjustment: Votes are set, but users can adjust their delegated and allocated vote power
- Timelock: Proposal is accepted, but waiting for the timelock to expire to be executed
- Executed: Proposal is executed
- Expired: Proposal is not executed and expired
- Defeated: Proposal did not receive enough votes to be executed
- Vetoed: Proposal is vetoed

Depending on the proposal's parameters, certain stages can be skipped.

Any proposal (general or committee) can be vetoed. Proposals can have up to 3 thresholds: Ripe points threshold, Committee threshold and Alternative committee threshold.

- Ripe points threshold is a percentage of total Ripe points that need to veto the proposal.
- Committee threshold is a certain number of committee members that need to veto the proposal. It is defined in % of total committee members at the start of the proposal.
- Alternative committee threshold is the same as committee threshold, but only oversights committee members' vetos are counted.

Proposals can be of 2 types: General proposals and Committee proposals. General proposals are voted on by all stakers, while Committee proposals are voted on by the committee members.

2.4.1.2 Committees

The committee membership is tracked and managed by `CSuite` contract. The voting power of committee members comes from the users who allocate their Ripe point power to the committee members. Compared to Ripe points delegation, this allocation does not reduce the voting power of the user.

A committee with enough members can be designated as one of the following types:

- Technical
- Legal
- Policy
- Oversight
- Community

Committee proposals can be created and voted on only by the committee members from committees designated for this exact proposal type.

The committee members are elected by the stakers for a period of time via `ElectionHall` contract.

Any committee member can resign at any time, resignation immediately starts a special election for a new committee member.

2.4.1.3 `BootStrapper` *contract and* `TrainingWheels` *role*

After the deployment of the entire Ripe system, the `BootStrapper` contract will used together with the `TrainingWheels` role holder to manage the system until the governance is fully operational. These 2 entities will have the power to set up initial parameters via setters.

2.5 Version 4 changes

In this version of code the `jUSD` token was renamed to `Green`.

2.6 Assumptions:

- Stability pool and governance are not an Endaoment strategy.
- Stability pool tokens all have 18 decimals.
- Stablecoins must be stable compared to USD, not stable to other currencies.
- Partner liquidity for stabilizer shall never be an LP token the stabilizer can hold anyway.
- As there are a lot of "manual" configurations that can be done, we assume all configuration parameters are tested and well-chosen to work together as intended. It is assumed that over time the parameters are monitored and adjusted to keep the system stable.
- Stable tokens do not de-peg so quickly that the system cannot react
- The system interacts with a variety of assets (e.g., strange or unintended token behavior). All assets are tested and work as intended with the system.
- The stabilizer's pools are monitored closely and any changes in e.g., the pool balanced state is identified, and measures are taken accordingly
- `isEndaoStable` will not be `true` for jUSD
- Price oracles return their prices in 18 decimals, or an appropriate conversion is performed
- The pools the stabilizer acts on are checked to be compatible and will always be more valuable when balancing the pool.
- The system will never hold any debt against jUSD and Ripe tokens as collateral.
- Endaoment will never invest in any strategy that yields profits in jUSD.
- Governance strats will not have debt terms, and so cannot hold debt.
- The system will be working with ERC20 tokens that have Chainlink oracle feeds.
- The system will not be working with reentrant ERC20 tokens, like ERC777.
- Rebasing ERC20 tokens won't be used as assets in the strategies.
- `BootStrapper` and `TrainingWheels` will not setup the system in a way that it can't be changed by the governance. The initial setup is assumed to be correct and the system will be able to change it over time.

In case collateral drops too fast (causes could be manifold like successful price manipulations) and, thus, preventing a proper liquidation process or in case other issues (e.g., too high gas costs) prevent proper liquidation, the system will accumulate bad debt.

3 Limitations and use of this 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 assessed 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 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:

- 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	1

- RipeGovStratPoints Delegatee Can DoS Delegators Risk Accepted

Medium-Severity Findings	1

- Proposals for Liquidity Operations Might Revert or Lose Value Risk Accepted

Low-Severity Findings	3

- Function rebalanceAssetDebtAllUsers() Gas Cost Risk Accepted
- Inconsistent Transfer and Accounting Risk Accepted
- Redemption Payback Buffer Changes Borrow Conditions Risk Accepted

5.1 `RipeGovStratPoints` Delegatee Can DoS Delegators

Design High Version 3 Risk Accepted

CS-RIPE-062

Users can delegate their Ripe points to other users. After such adjustment in `RipeGovStratPoints.delegateTo()` function, A call is made to `Janitorial.cleanHouseAfterGovParticipation()` function. This will trigger `RupeGovStrat.updateGovPowerFromJanitorial()` function. The `BoardRoom.govPowerDidChangeForUser()` will be triggered for the delegator and every delegatee. The `govPowerDidChangeForUser()` is a function that can consume a lot of gas. It will try to adjust users voting power for all user votes in:

- `PropHouse` votes
- `CSuite` committee allocations
- `ElectionHall` votes
- All dynamic voting parameters

A malicious delegatee can maximize the number of needed calculations by participating in all votes. As a result, the delegator will have to pay a lot of gas to adjust the voting power of the delegatee. This effectively allows a single delegatee to DoS all of its delegators. Also this situation can arise during a normal operation, when a delegatee participates in a lot of votes.

Risk accepted:

Hightop Financial is aware of the issue and plans to handle this case as follows:

- If there is a malicious user doing this, governance can lock their account: *LedgerManager.setUserIsLocked()*
- As props/votes that the delegatee voted for expire (or are executed), then their user vote data will be deleted/removed. As that data is removed, the gas/cost for the delegator will slowly be eliminated.

Also, Hightop Financial added a set of special functions to facilitate storage cleaning.

5.2 Proposals for Liquidity Operations Might Revert or Lose Value

Design | Medium | Version 8 | Risk Accepted

CS-RIPE-055

Such `EndaoGovB` has proposals that trigger functions like: `addLiquidityTwoCoins`, `addLiquidityOneCoin`, `removeLiquidityOneCoin`, `swapAsset`. These functions are used to manage liquidity pools and swap assets between them. However, proposal needs to be created with parameters that can be invalid or not optimal, and the proposal can be rejected or executed with a loss.

In addition, malicious attackers can intentionally sandwich proposals and maximize the loss for such proposals.

Risk accepted:

The issue is hard and complex to mitigate. Each proposal execution should be carefully assessed to be aware of the worst-case scenario. E.g., an attacker who sandwiches the execution to extract value.

5.3 Function rebalanceAssetDebtAllUsers() Gas Cost

Design | Low | Version 1 | Risk Accepted

CS-RIPE-012

The function `CreditBorrow.rebalanceAssetDebtAllUsers()` is a restricted function that tries to rebalance the debt of all users. This is an extremely high gas cost operation and will not fit gas block limits for a reasonable number of users. The complexity of the operation is `O(numUser * numStrats * numAssets)`.

Risk accepted:

Hightop Financial responded:

We also have `rebalanceAssetDebtForUsers()` for when we want to pass in an array of specific users. So, if/when `rebalanceAssetDebtAllUsers` gets too expensive, we can just run that. Being

able to always run `rebalanceAssetDebtAllUsers()` is not a requirement for the system to be healthy.

5.4 Inconsistent Transfer and Accounting

Correctness | Low | Version 1 | Risk Accepted

CS-RIPE-014

When withdrawing tokens from strategies, the user's balance gets reduced by calling `reduceBalance` on the strategy data contract. This deducts the amount from the corresponding user balance mappings. When calling `recallTokens` afterward, it is not guaranteed, that this amount deducted before is the amount that will be sent to the users. Currently, the protocol should be protected but users might get less than they desired.

Risk accepted:

Hightop Financial responded:

Yes it's possible that the user will not get the exact amount they expect. Should only be dust difference tho.

5.5 Redemption Payback Buffer Changes Borrow Conditions

Design | Low | Version 1 | Risk Accepted

CS-RIPE-018

The Borrow terms are computed on aggregated positions of the user. When the user is redeemed, the `AuctionHouseConfig.ltvPaybackBuffer` is used to compute the `targetLtv`. The goal is to make sure the user is no longer redeemable after the redemption. However, the repayment assets withdrawn from user are not pro rata. `CreditRedeem._redeemUserAssets()` tries to get the maximum from the first to the last user's asset. As a result, the `BorrowData.ltv` will change. It can become lower or higher than the `targetLtv` redemption tries to achieve. The user position can become redeemable immediately after the redemption. It can also become liquidatable, depending on the leftover assets of the user. The opposite scenario is also possible, the user can become non-redeemable after the redemption, but with a higher ltv than the `targetLtv`.

Risk accepted:

Hightop Financial responded:

Code has been re-worked to change the ordering of redemptions. It will start with non priority strats/assets first. And then it'll go thru priority strats/assets in reverse order — similar to debt asset reduction in CreditByAssets.vy. So while the finding is not totally resolved, the main risks should be mitigated. Typically (but not guaranteed), the lowest LTVs will be correlated with the lowest priority strats/assets. Therefore, the common case will be that as redemptions occur, overall LTV should increase, therefore helping make the user's debt position healthier. Now redemptions that lead to worse debt health, or liquidations should be extremely rare.

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	4

- Voting Power Double Counting Problem Code Corrected
- DoS Due to refreshStabilityPoolShares() Code Corrected
- Gas Exhaustive Loops Code Corrected
- RebaseERC20 and StabilityPool Inflation Attack Code Corrected

High-Severity Findings	6

- PropHouse Signature for Vote Casting Can Be Replayed Code Corrected
- CurveDexData.calcProfitForStabilizer() Inconsistency in Unit Types Code Corrected
- GovLootboxLiquid Deposit Rewards Allows DoS Code Corrected
- Endao Redemptions Can DoS User Liquidations Code Corrected
- Ripe and jUSD Bonds Can DoS the User Liquidations Code Corrected
- Slippage on Asset Transformation Does Not Protect From Sandwich Attacks Code Corrected

Medium-Severity Findings	13

- Committee Power Update Can Be Skipped Code Corrected
- Fasttrack Committee Proposals Temporal Vote Double Accounting Code Corrected
- Proposal General Veto Power Temporal Double Counting Code Corrected
- Proposal Cannot Be Executed Code Corrected
- Wrong ParamType Is Used for Some Proposals Code Corrected
- PropHouse Missing Ecrecover Empty(Address) Check Code Corrected
- ERC20 Boolean Return Values Not Handled Correctly Code Corrected
- Inconsistent Fund Aggregation in Endao Wallet Code Corrected
- Inconsistent Voting Power Code Corrected
- Liquidations Require Position in a Stability Pool Code Corrected
- Stability Pool Profits and Losses Distribution Code Corrected
- TreasuryConfig.numRipeBondPrograms Gas Cost Code Corrected
- TreasuryConfig.setRipeBondConfig() Wrong Access Control Code Corrected

Low-Severity Findings	27

- Committee Size Threshold Is Shifted by 1 Due to Inconsistency Code Corrected
- Wrong Check Is Enforced for Some Proposals Code Corrected
- Function LedgerLocker.releaseLockEarly() Code Corrected
- Superfluous Asset Data Validation Code Corrected
- Add and Reduce Debt Consistency Code Corrected

- Consistency of isEnabled Flag on jUSD and Ripe Bond Configuration Code Corrected
- CreditBorrow._getBorrowIntervalAvail Can Overflow Due to Config Change Code Corrected
- CurveDexData.STABILIZER_PROFIT_BUFFER Serves No Purpose Code Corrected
- HQ Interface Violation Code Corrected
- Incorrect Calculation of the Debt Amount to Repay Code Corrected
- Keeper Fee on Liquidating Users With Small Positions Code Corrected
- Leftover Approvals Might Break Some Tokens Code Corrected
- Open TODO Code Corrected
- Rebasing Tokens Transfer Corner Case CODECORRECRebasing Tokens Transfer Corner Case Code Corrected
- Redundancy in CreditByAsset._reduceDebtFromAssetIfAvail() Code Corrected
- Ripe Bond Config With Index 0 Code Corrected
- StabilityPool.refreshStabilityPoolShares Rounding Error Code Corrected
- TellerDeposit desiredDepositAmount Wrong Units Code Corrected
- TreasuryConfig RipeBondConfig.minEntitledRatio Cannot Be Altered Code Corrected
- TreasuryConfig vestingLength Unchecked Setters Code Corrected
- TreasuryConfig.canClaim Is Ignored by Janitorial Code Corrected
- Unclear Purpose of lastModified Specification Changed
- Unnecessary Minimum Calculation Code Corrected
- Unnecessary Parameters in Calldata and Function Parameters Code Corrected
- Unused Functions Code Corrected
- Unused Storage Variables Specification Changed
- CreditData.totalUserDebtByAsset Is the Same as UserDebt.principal Code Corrected

Informational Findings	14

- Gas Optimizations Code Corrected
- Lacking Sanitization for Input Data of Proposals Code Corrected
- Proposal With Failed Execution Does Not Emit Event Code Corrected
- Check of Balance After Transfer From TellerDeposits Code Corrected
- CreditBorrow.borrowForUser Check isActivated Code Corrected
- Dev Comment Bigger Vs. >= Code Corrected
- Dust in Ripe Bond Programs Code Corrected
- Incorrect Comments Code Corrected
- Partner Could Block Disabling a Stabilizer Pool Code Corrected
- Potential Vulnerable Voting Code Corrected
- Redemption Fee Can Be Less Than Designed Code Corrected
- Transfer to Self in StratData Contracts Code Corrected
- Unused Function transferBalance in Stability Pool Code Corrected
- Unused Return Value isSafeLtv Code Corrected

6.1 Voting Power Double Counting Problem

Design | Critical | Version 3 | Code Corrected

CS-RIPE-052

6.1.1 `CSuite` *problem*

Any user of the system can call `CSuite.setCommitteeAllocs()` to give a committee member voting power in committee proposals. If a user's governance power changes in any way, `CSuite.govPowerDidChangeForUser()` will be triggered to update the user's committee allocations.

There is a gas optimization in current implementation that skips updating the committee allocations if the user's governance power has already been updated this block:

```
# already done
if self.lastUpdate[_user][i] == block.number:
    continue
```

Consider the following scenario:

- Alice is a member of the committee.
- Bob has 1 governance power and allocates 1 governance power to Alice.
- Charlie has 1 governance power and allocates 1 governance power to Alice as well.
- Eve controls 2 accounts, one with 1 governance power and one with 100 governance power. Let's call the Eve's EvePoke and EveMain.

Alices committee allocation is 2 at this point.

Eve performs the following actions in two blocks:

Block with number N:

1. EveMain delegates(via `RipeGovStratPoints.delegateTo()`) 100 governance power to Bob. - This will set `lastUpdate[Bob][Alice] = N`

   Alice's committee allocation is now 102.

Block with number N+1:

1. EvePoke delegates 1 governance power to Bob. - This will set `lastUpdate[Bob][Alice] = N+1`

   Alice's committee allocation is now 103.

2. EveMain changes 100 delegation from Bob to Charlie. The Bobs `CSuite.govPowerDidChangeForUser()` will be triggered, but power allocated to Alice will not be updated because it was already updated in this block. Charies `CSuite.govPowerDidChangeForUser()` will be triggered and Alice's committee allocation will be updated to 203

This way Alice's committee allocation will be 203 instead of 103. The Eve's governance power will be counted twice.

Until `govPowerDidChangeForUser()` is called for Bob, Alice's committee allocation will be counted twice. Eve can create more accounts that allocate to Alice and increase Alice's committee allocation even more.

Eve need "poke" accounts to avoid the `Ledger.touch()` protection that does not allow to change the delegation in the same block twice.

6.1.2 `ElectionHall` problem

Similar problem arises in `ElectionHall` contract, due to this skip:

```
# already done
if self.lastUpdate[_user][electionId] == block.number:
    continue
```

6.1.3 `DynGov` problem

Similar problem arises in `DynGovLootAssests` and `DynGovJuiceAssests` contracts, due to the same skip.

Effectively, in `CSuite`, `ElectionHall` and both `DynGov` contracts, important invariant is broken:

- The committee allocation (or vote) of a user is not greater than the governance power of the user.

Code corrected:

This skip was removed from the code allowing the committee allocations to be updated in the same block.

6.2 DoS Due to refreshStabilityPoolShares()

Design | Critical | Version 1 | Code Corrected

CS-RIPE-001

The `StabilityPool.refreshStabilityPoolShares()` function first iterates over all the assets in the pool, and then iterates over all the users for the asset. This is a quadratic operation, and will DoS the system if the number of users and assets is large. `AuctionHouseCollections.liquidateUsers` calls `StabilityPool.refreshStabilityPoolShares` in every call. Thus, with a large number of users and assets, no liquidations can be performed. Liquidators will be unable to have profitable liquidations (due to extremely high gas costs), and the system will be unable to liquidate undercollateralized positions.

Code corrected:

The `StabilityPool` mechanism was redesigned to fix this issue. The shares no longer need to be refreshed. The price is now calculated based on the claimable value and the stability pool asset balance.

6.3 Gas Exhaustive Loops

Design | Critical | Version 1 | Code Corrected

CS-RIPE-002

The uses multiple loops (e.g., over strategies and assets or users). These loops might do storage operations and also often include calls to other contracts in the system. Depending on the call path some loops are also executed multiple times (e.g., when Janitorial also loops over strategies and assets another time). This can be extremely gas-expensive and exhaust the block-gas limit.

In some cases, this will DOS the system for sure in the long run. Particular examples are the ever-increasing variables `numUserEndaoRedemptions`, `numDepositRewardsSums`, `numVotersDepositRewards`, `numVotersRipeRewards` or `numVotersEndaoYield`. Loops use

these variables as the upper bound. Over time this will inevitably become a DOS vector with too many redemptions. Some of the variables like the vote-related variables can be easily increased to DOS the system by users.

Example of a loop mentioned in Gas Exhaustive Loops is a `CreditByAssets._reduceDebtPerAsset()` and `_reduceDebtFromStratIfAvail()` functions. They contain the loops over all the strategies and assets of the user. If number of strategies and assets is large, the gas cost of these functions can be high. These functions are called when the `AunctionHouseCollections.liquidateUsers()` and `CreditRedeem.redeemOnDebtPosition()` are called. Thus, the DoS on liquidations and redemptions is possible.

Code corrected:

The data accessed by the loops is now cleaned up in most cases, thus in theory, the loops should not be able to DOS the system anymore. Some loops still can be costly to run, but they should not be able to DOS system critical scenarios like liquidations or redemptions anymore.

6.4 `RebaseERC20` and `StabilityPool` Inflation Attack

Security | Critical | Version 1 | Code Corrected

CS-RIPE-003

The way how `_calcSharesOnDeposit()` function works in `RebaseERC20` and `StabilityPool` contracts can be used by an attacker to front-run depositors and steal part of their deposits. Example that demonstrates the issue:

1. Attacker sees a big deposit of Y tokens into `RebaseERC20`. He sandwiches the deposit
2. Attacker deposit X tokens. X is `newShares`. X needs to be high enough to fulfill the config min deposit
3. Attacker withdraws X-1 tokens. 1 share remains
4. Attacker donate Y/2 tokens to `stratWallet`. `totalBalance` becomes (Y/2 + 1).
5. Execute the deposit from the victim. Victim gets: `newShares = _amount * _totalShares / totalBalance = Y * 1/(Y/2 + 1) = 1`. Error will be Y/2 - 1. `totalBalance` becomes Y* 3/2 + 1.
6. Attacker withdraws the 1 share. Gets ~Y*3/4 > Y/2. Profit of Y/4 from the rounding error of the victim deposit

Related blog post: https://blog.openzeppelin.com/a-novel-defense-against-erc4626-inflation-attacks

---

Code corrected:

Virtual shares and assets based on `YieldBox` approach are used to prevent the attack. Both `RebaseERC20` and `StabilityPool` contracts have 1 virtual asset and 10^8 virtual shares. This makes the inflation attack practically impossible: https://docs.openzeppelin.com/contracts/5.x/erc4626#defending_with_a_virtual_offset

6.5 `PropHouse` Signature for Vote Casting Can Be Replayed

Security High Version 3 Code Corrected

CS-RIPE-072

The signature used in `castManyVotesWithSignatures()` function of PropHouse contract, does not have a nonce or any other mechanism to prevent replay attacks. Thus, a vote submitted by a user can be replayed by an attacker to cast the same vote again. This can be a problem if a user has changed his mind and voted for a different outcome of the proposal. Attacker then can resubmit the same signature again and revert the vote of the user.

In addition, the EIP-712 Domain separator is computed once during contract deployment and is not updated. This means that in case of a hard fork, the domain separator will remain the same and the signature can be replayed on the new chain.

Code corrected:

Hightop Financial responded:

Nonce was added to the signature to prevent replay attacks.

Domain separator is now computed on the fly with the relevant chainId.

6.6 `CurveDexData.calcProfitForStabilizer()` Inconsistency in Unit Types

Correctness High Version 2 Code Corrected

CS-RIPE-050

In Version 2, the Leftover balance of the `EndaoLiqWallet` is included in the profit calculation. The final profit calculation is as:

```
(_lpBalance - lpDebt) + netJusdBal
```

The `_lpBalance` and `_lpDebt``is measured in the LP shares, while the ``netJusdBal` is measured in the jUSD tokens. For reference, the `lpDebt` is calculated as:

```
lpDebt = netJusdDebt * (10 ** 18) / CurvePool(_pool).get_virtual_price()
```

With time the virtual price of Curve pool will keep growing, and thus more jUSD tokens will be worth 1 LP share. Thus, the stabilizer will start refusing stabilizations, where the leftover balance of the `EndaoLiqWallet` decreases. The stabilization mechanism will not be able to keep the peg, if the jUSD appreciation grows compared to the other assets in the pool.

Code corrected:

`netJusdBal` is now converted to LP token via `netJusdBal * (10 ** 18) / virtualPrice` and named `netJusdBalInLp` before adding it in the calculation.

6.7 `GovLootboxLiquid` Deposit Rewards Allows DoS

Design | High | Version 2 | Code Corrected

CS-RIPE-051

A user can use `setDepositRewardsAllocs` to set the weight of the ripe and jUSD rewards for a strategy and asset pair. If the startegy+asset pair is voted the first time, the vote deposit result is registered, and `GovLootboxLiquidData.numVoteDepositResults` is increased by 1. Two loops exist that iterate over the `numVoteDepositResults`:

1. `applyVoteDepositRewards` is called to apply the rewards to the strategy+asset pair. The strat+asset is not configured for deposit, it is skipped.
2. `refreshEligibleVoteDepositAssets` cleans up the start+asset pairs that are not eligible for rewards.

The issue is that users can spam the `setDepositRewardsAllocs` to increase the `numVoteDepositResults`. This is a potential DoS vector, that would prevent the call of `applyVoteDepositRewards` and thus the change of reward allocations.

Code corrected:

The issue applies to the old governance structure. The fix does theoretically still allow a DoS attack depending on the parameters set. We will assume that the parameters are set in a way that the attack is not feasible. The new code of the governance is up to be audited in the next iteration.

6.8 Endao Redemptions Can DoS User Liquidations

Design | High | Version 1 | Code Corrected

CS-RIPE-004

Every time a user redeems jUSD on the `EndaoReduce`, the `EndaoData.numUserEndaoRedemptions` is incremented. The `getRedeemJuice()` will iterate all the user redemptions to calculate the juice points. This can DoS all the system functions that rely on the juice score, similar to the Ripe and jUSD bonds can DoS the user liquidations issue. The liquidations are affected by this issue as well.

Code corrected:

The list of endao redemptions is removed. Now there is one data object that aggregates all the user redemptions.

6.9 Ripe and jUSD Bonds Can DoS the User Liquidations

Design | High | Version 1 | Code Corrected

CS-RIPE-005

Every time user purchases bonds, `TreasuryData.addJusdBond()` or `addRipeBond()` is called. The `numJusdBonds[_user]` or `numRipeBonds[_user]` is increased by 1. The `getBondJuice()` will

iterate all the bonds of the user. With time active or malicious users can have big number of bonds and the `getBondJuice()` will be very expensive to execute. Hence, the `JuiceStandScores.getJuiceScore()` will be very expensive to execute. This can Dos all the system functions that rely on the juice score. Among them:

- `AuctionHouseCollections.liquidateUsers()`
- `CreditRedeem.redeemOnDebtPosition()`

The execution of these functions due to DoS will be very expensive and liquidators will opt to not liquidate users due to the high cost of the transaction.

Code corrected:

Following steps were performed to limit the max gas usage:

- Single bond per epoch (per user), instead of multiple bonds per epoch
- Max num bonds per user
- Data clean-up on all user interactions with protocol (removing bonds that are past decay, or removing “most decayed” when user reaches max)

6.10 Slippage on Asset Transformation Does Not Protect From Sandwich Attacks

Security | High | Version 1 | Code Corrected

CS-RIPE-006

Following contracts have a `transformAsset` function that has a `maxSlippage` parameter:

- `EndaoManager` (just calls the `EndaoWallet`)
- `EndaoWallet`
- `TellerDeposit`
- `TellerWithdraw`

This parameter is intended to be used as a way for users to protect themselves from sandwich attacks. However, contracts like `CurveDexSwap`, `CurveDexAddLiq`, `CurveDexRemoveLiq` compute `minAmountOut` onchain from this parameter. This would not protect the user from front-running. The front-runners will change the onchain state of pool to make the conversion at an unfavorable rate. The `minAmountOut` parameter should be either computed off-chain or bound to an on-chain oracle value.

In addition, function `TellerWithdraw.reallocateTokens()` has a `maxSlippage` parameter that is used for two different trades. One from `_fromAsset` to `_withDrawAsset` and another from `_withDrawAsset` to `_toStratAsset`. The `maxSlippage` parameter is used in both trades, but different swaps need to have different slippage values. Misconfiguring the slippage value for at least one of the trades can lead to a sandwich attack.

Code corrected:

`maxSlippage` parameter is replaced with `minAmountOut` parameter. This offchain computed `minAmountOut` is passed directly to the contracts that check the slippage. This way the sandwich will not be able to execute the trade at an unfavorable rate, since the desired output is fixed.

6.11 Committee Power Update Can Be Skipped

Design | Medium | Version 6 | Code Corrected

CS-RIPE-074

If a user calls `CSuite.setCommitteeAllocs()`, if its Ripe points were already updated in the same block, the propagation of allocated Committee power to Proposals will be skipped.

`CSuite.setCommitteeAllocs()` relied on `Janitorial` contract to trigger `CSuite.govPowerDidChangeForUser()`. However, this only happens, if `RipeGovStratPoints._updateEarnedGovPowerForUser()` has `shouldUpdateGovVoting` set to `true`. This won't be the case, if the user's Ripe points were already updated in the same block.

This can be abused in the following way:

1. Alice holds and allocates 100 Points to Bob and 1 to Charlie.

In a single block:

2. Any user delegates/removes delegation to/from Alice, just to trigger an update of Alice's Ripe points.
3. Alice calls `CSuite.setCommitteeAllocs()`. The propogation of allocated Committee power to Proposals will be skipped, since the `RipteGovStratData.updateEarnedGovPower()` will return `shouldUpdateGovVoting` as false. Alice sets Bob's power to 1 and Charlie's to 100.

In another block, Charlie votes again on a Proposal, and thus his new power of 100 will be used. Bob's proposals will keep using his old power of 100. Points are effectively double counted.

Code corrected:

`RipteGovStratPoints` now trigger `govPowerDidChangeForUser()` updates without `shouldUpdateGovVoting` check. `shouldUpdateGovVoting` flag is not returned by `updateEarnedGovPower()` anymore.

6.12 Fasttrack Committee Proposals Temporal Vote Double Accounting

Design | Medium | Version 6 | Code Corrected

CS-RIPE-065

Assume Alice has 101 Points and allocates 1 to Bob and 100 to Charlie. There exist a fasttrack proposal for that committee that requires 150 Points to pass. Both Bob and Charlie vote "Yes" for the proposal.

Assume Alice decided to call `CSuite.setCommitteeAllocs()` and change the allocation to 100 to Bob and 1 to Charlie.

At the end of `setCommitteeAllocs()` call, `Janitorail.cleanHouseAfterGovParticipation()` is called. Eventually, `CSuite.govPowerDidChangeForUser(Alice)` is called.

This will trigger a loop, where following order of can happen:

1. `CSuite._updateVotingPower(Bob)`
2. `CSuite._updateVotingPower(Charlie)`

`_updateVotingPower()` will call `PropHouse.committeePowerDidChange()` and update the committee power of the user in all proposals.

If proposal `_canBeFastTracked()` check passes, Proposal execution block number will be immediately set to current one.

The problem is, the order of 1) and 2) actions has basically following effect on the proposal:

1. Add 99 points to the proposal
2. Remove 99 points from the proposal

Effectively, between 1) and 2), the proposal will have 200 points, which is enough to fasttrack the proposal. The times will be set to the same block number, and the proposal can be executed.

Thus, there is basically a double accounting due to the way the committee power is updated.

Note that reverse order is also possible, where first Charlie is updated and then Bob. This will not trigger a problem.

Code corrected:

Two stage process was introduced on `PropHouse` to avoid fasttracking proposals before all updates are done.

`_updateVotingPower()` will save proposals that pass fasttrack requirements in a special transient list `doubleCheckFastTrackProps`. After all updates are done, new `PropHouse.doubleCheckFastTrackPropsFromJanitorial()` function will be called by `Janitorial` contract and proposals will be actually fasttracked.

6.13 Proposal General Veto Power Temporal Double Counting

Design | Medium | Version 6 | Code Corrected

CS-RIPE-075

Similar to Fasttrack committee proposals temporal Vote double accounting, proposals can be vetoed based on the general veto power criteria, due to how the update loop applies the delegation changes.

Assume Alice has 101 Points and allocates 1 to Bob and 100 to Charlie. Both Bob and Charlie veto some proposal. Extra 50 points are required to veto the proposal, based on the veto requirement `generalPowerRatio`.

Alice decides to call `RipeGovStratPoints.delegateToMany()` and change the delegation to 100 to Bob and 1 to Charlie.

At the end of `delegateToMany()` call, `Janitorail.cleanHouseAfterGovParticipation()` is called.

This will trigger a loop, where following order of can happen:

1. `RipeGovStratPoints._updateVotingPower(Bob)`

   `_updateVotingPower()` will call `PropHouse.govPowerDidChangeForUser()` and update the general veto power of the user in all proposals via `PropHouse._adjustVetoWeight()`. `PropHouseData.adjustVetoWeight()` will be called and set the `didVeto` flag to `true` since now extra 99 points were added to the proposal veto power.

2. `RipeGovStratPoints._updateVotingPower(Charlie)`

   Similarly, Charlie's veto power will be updated and the proposal veto power will be decreased by 99 points. However, the `didVeto` flag will not be reset to `false` since the proposal will also have `isFinished` flag set to `true` as well during the 1) update.

Thus, there is basically a temporal double accounting that can transition the state of the proposal based on intermediate state.

Code corrected:

The issue was mitigated by using transient storage variables to correctly facilitate the state transition.

6.14 Proposal Cannot Be Executed

Correctness Medium Version 4 Code Corrected

CS-RIPE-063

The proposal of type `GRANULAR_721_DISABLE_TOKEN_PRICE` in `VariousNftStratsGov` cannot be executed because `_executeProposal()` function does not support this enum variant. Execution of such a proposal will raise the `Type not supported` error.

6.15 Wrong `ParamType` Is Used for Some Proposals

Correctness Medium Version 4 Code Corrected

CS-RIPE-064

A wrong `ParamType` is used for some proposals.

1. `CommunalGovA.proposeIsNftCollection` uses `ParamType.CAN_DIRECT_DEPOSIT_ASSET`.
2. `StratMapGov.proposeSetStabilityPoolId` uses `ParamType.SET_RIPE_GOV_STRAT_ID`.
3. `VariousPriceSourcesGov.proposeDisableCurvePrice` uses `ParamType.CURVE_SET_FEED`.

6.16 `PropHouse` Missing Ecrecover Empty(Address) Check

Security Medium Version 3 Code Corrected

CS-RIPE-071

The `ecrecover` function can return empty(address) if the signature is invalid. `PropHouse._isValidVoteSig` does not check for this case, which can lead to a vote being accepted even if the signature is invalid.

Code corrected:

A check has been added to ensure that the recovered address is not empty.

6.17 ERC20 Boolean Return Values Not Handled Correctly

Design | Medium | Version 1 | Code Corrected

CS-RIPE-007

According to the ERC20 standard, some of the functions return a boolean value. E.g. the `transfer` function returns `True` if the transfer was successful.

```
function approve(address _spender, uint256 _value) public returns (bool success)
function transfer(address _to, uint256 _value) public returns (bool success)
function transferFrom(address _from, address _to, uint256 _value) public returns (bool success)
```

However, some tokens do not return a boolean value but instead return nothing. Ripe's code has several places where this case is attempted to be handled using the `default_return_value` parameter.

```
ERC20(asset).transfer(to, value, default_return_value=True)
```

However, this is not correct, as the `default_return_value` parameter is only used if the function returns no value at all. If the function returns `False`, the `default_return_value` parameter is ignored, but the execution of the code continues. The correct way is to assert that the return value is `True`.

In cases when a known token like jUSD is used, the `default_return_value` parameter is not needed at all, as the token is known to always return `True` or revert otherwise.

In addition, in some cases, the calls to potentially non-bool returning tokens are not handled at all. For example:

1. `CreditRepay._fundRepayment()`
2. `AuctionHouseFungible._handlePayment()`

Code corrected:

`assert` statement was added before calls to ERC20 functions to ensure that the return value is `True`. `default_return_value=True` used consistently to prevent reverts in case token does not adhere to ERC20 standard.

6.18 Inconsistent Fund Aggregation in Endao Wallet

Design | Medium | Version 1 | Code Corrected

CS-RIPE-037

If `EndaoStabilizer.disableStabilizerPool()` is called on some pool, EndaoWallet contract receives the LP tokens of the pool. These tokens could be burned later to receive the jUSD from the pool. However, the funds from the stabilizer are not tracked in case the LP tokens are converted to underlying token. The funds would mix with the jUSD from other sources in the EndoWallet e.g fees.

The jUSD that comes from stability pool LP shares can be seen as backed by the other assets in the pool. But upon LP token unwrapping, the jUSD cannot be seen as profits of the Endaoment. In general, funds should be accounted correctly according to their origin to avoid incorrect fund allocation, unintended swapping or burning.

Code corrected:

A new `EndaoLiqWallet` contract is introduced to handle any tokens related to LP (jUSD and Ripe LP). This wallet is separated from the main Endaoment reserves wallet.

6.19 Inconsistent Voting Power

Correctness | Medium | Version 1 | Code Corrected

CS-RIPE-008

The voting power in governance votes is calculated when a user votes but is only used when it is emitted as part of the event (e.g., in `CastVoteDepositRewards` the `govPower` variable). When applying the vote result it is re-calculated and used to evaluate the voting shares. But there is no guarantee that this voting power stays constant nor that the user who voted in the beginning still has voting power.

Code corrected:

The two-stage voting process is now consistent in the way it calculates the voting power. Once the user submits a vote, but his voting power is changed, the voting power is updated. When the vote is applied, the up-to-date voting power is used.

6.20 Liquidations Require Position in a Stability Pool

Design | Medium | Version 1 | Code Corrected

CS-RIPE-038

Liquidators that want to buy an auction need to hold a position in a stability pool. This leads to multiple problems: First, stability pool positions are used as a first mean of liquidation, so potentially their position can be seized before they can buy an auction. Second, due to the `Ledger.touch` lock, the liquidators cannot deposit and buy the auction and withdraw in the same block. This limits the ability of liquidators to use flash loans as a mean to get needed liquidity. The need for liquidators to hold position in the system adds costs to the liquidation process. Consider two approaches:

1. Current approach: liquidator needs to submit 3 tx across 3 blocks to get pay jUSD, get collateral, and withdraw it out of the Ripe
2. Single transaction liquidation approach: liquidator pays jUSD and directly same Tx gets underlying collateral.

In 1, compared to 2 the liquidators need to account for all 3 tx gas costs and potential deprivation of collateral value over 2 blocks time period. This lowers the health of positions that are liquidated on average.

Code corrected:

A new flag `_fromStabilityPool` is added to functions that handle bidding or buying auctions. If this flag is set to true, the liquidators funds from the stability pool are used to buy the auction. If this flag is set to false, the funds will be transferred from the `msg.sender`.

6.21 Stability Pool Profits and Losses Distribution

Correctness | Medium | Version 1 | Code Corrected

CS-RIPE-009

The distribution of funds in the stability pool is not checkpointed and all losses or profits are socialized. In consequence, a user that deposited 100 dai worth 100$ in `t0` and his funds were used in `t1` to buy positions in liquidation to generate profits valued. Let's assume we bought 100 tokens for the 100 dai, valuing the position at 100$. Over `t1` to `t10` the funds generated yield valued (this might be by rebasing or through claiming mechanisms which would complicate the issue more). Let's assume the 100 tokens are now worth 110$. In `t10` another user stakes into the stability pool and their funds are used to buy assets in liquidation. The second user also stakes 100 DAI. They are now used to buy the asset which is now valued at 110$. Hence, we receive around 90 tokens. Both users are now eligible to withdraw 50% of the collateral bought by the stability pool. The second user would receive a share of the position gains of the first user. The first user would lose as his profits were "socialized".

To summarize, late depositors will profit from yields of prior depositors.

Code corrected:

With a new `StabilityPool` shares mechanism, the users who join or leave pool will get their fair share of the pool's value. They won't be able to get the profits or losses of the previous users.

6.22 TreasuryConfig.numRipeBondPrograms Gas Cost

Design | Medium | Version 1 | Code Corrected

CS-RIPE-010

Every time a bond program is added, the number of bond programs is increased by 1. When the epoch is refreshed, the `TreasuryEpochs._applyEpochRefresh()` will iterate over all the ripeBondPrograms. Each iteration of the loop will need to query the `TreasuryConfig`. With a great number of Ripe bond programs, the gas cost will be high. The epoch refresh is done in `Janitorial.cleanHouse()`, that is called in:

- `AuctionHouseCollections.liquidateUsers()`
- `CreditRedeem.redeemOnDebtPosition()`
- `AuctionHouseFungible.buyMultipleFungibleAuctions()`

Those operations should have as little gas cost as possible. Elevated costs might lead to the accumulation of bad debt.

Hightop Financial responded:

- Added `removeRipeBondProgram()` to TreasuryConfig to remove bond programs that are not actively being used.
- The Janitorial functions that used to call `refreshBondEpoch()` is no longer called for a user except when the user herself is interacting directly with the protocol. So for mission critical things like liquidations or redemptions, that loop will not occur in the flow.
- The check to see if all bonds are depleted (which would iterate over all Ripe Bond Programs) is no longer occurring in the `refreshBondEpoch`. Instead, it's occurring at the end of every bond purchase, which calls `refreshAfterBondDepletion()`.

- When a new epoch is created, there's no way around iterating over all Ripe bond programs to reset the data for each bond program. However, this is no longer happening in the key flows (liquidation), and will mostly likely just piggy back off the bond purchase flow when it's depleted.

6.23 TreasuryConfig.setRipeBondConfig() Wrong Access Control

Correctness Medium Version 1 Code Corrected

CS-RIPE-011

The `TreasuryConfig.setRipeBondConfig()` function is callable only by `TreasutyRipeBonds` contract. However, the `TreasuryEpochs` contract needs to call this function to reset total if the ripe bond config is changed. Without this call, the ripe bond programs cannot be renewed. The `RipeBondConfig.tragetAmount` cannot not be changed after the ripe bond program is created.

Code corrected:

The check was corrected to allow the `TreasuryEpochs` contract to call the `setRipeBondConfig()` function.

6.24 Committee Size Threshold Is Shifted by 1 Due to Inconsistency

Correctness Low Version 3 Code Corrected

CS-RIPE-073

`Csuite._canBecomeDesignated` checking the number of members against `MIN_COMMITTEE_SIZE` without considering that the number of members is already shifted by 1 due to the way it's built.

Code corrected:

The greater than or equal to check is replaced by a greater than check.

6.25 Wrong Check Is Enforced for Some Proposals

Correctness Low Version 3 Code Corrected

CS-RIPE-066

Some proposals have a wrong check enforced:

1. `JuiceStandGovernor.proposeJuicePositionAllocs` checks `isEligibleJuiceAssetPositionToRemove`
2. `AuctionHouseFungibleGovernor.proposeRestartManyFungibleAuctions` check `isValidFungibleAuctionStart`

Code corrected:

The checks were corrected.

6.26 Function `LedgerLocker.releaseLockEarly()`

Design Low Version 2 Code Corrected

CS-RIPE-053

Most of the functions that calculate new release terms for users (deposit, transfer, withdrawal) release the lock if the configuration terms become less favorable compared to the user's lock terms. However the function `LedgerLocker.releaseLockEarly()` does not release the lock if the terms become unfavorable for the user. In addition, the exit fees that are taken from the user do not come from the user's lock terms, but from the newest configuration. Thus, in some cases, it will be more profitable to withdraw the funds and get immediate unlock, rather than calling `LedgerLocker.releaseLockEarly()`.

Code corrected

In `releaseLockEarly` before checking if a user can exit and calculate the fees, the new lock terms are queried. In case they changed the user can exit without fees.

6.27 Superfluous Asset Data Validation

Design Low Version 2 Code Corrected

CS-RIPE-054

In `CommunalAsset.setCanSwapInStabPool` the boolean parameter `_canSwap` is set for a given strategy and asset combination. After setting this parameter, the function `_validateDebtTerms` is called for configuration validation. However, this function does not validate anything related to `_canSwap` parameter. Hence, invoking this function uses additional gas without contributing to parameter validation. The only meaningful check for the parameter is done in the body of the `setCanSwapInStabPool` itself: `assert config.canSwapInStabPool != _canSwap`. The same applies for `_canAuction` in `setCanAuctionInstantly`

Code corrected:

The superfluous validation checks were removed.

6.28 Add and Reduce Debt Consistency

Design Low Version 1 Code Corrected

CS-RIPE-059

When `CreditByAssets._addDebtPerAsset()` is called, the debt is added in following way:

1. Add debt to priority strats from first to last
2. Add debt to non-priority strats of user from first to last

When `CreditByAssets._reduceDebtPerAsset()` is called, the debt is reduced in following way:

1. Reduce debt from priority strats of user from last to first

2. Reduce debt from non-priority strats of user from first to last

This is technically not consistent. The `_addDebtPerAsset()` aims to maximize the debt in priority strats and minimize the debt in non-priority strats. The `_reduceDebtPerAsset()` aims to minimize the debt in priority strats and maximize the debt in non-priority strats.

Similarly, the `CreditByAssets._addDebtToStratIfAvail()` adds debt to assets in following way:

1. Add debt to priority assets from first to last
2. Add debt to non-priority assets from first to last

The `CreditByAssets._reduceDebtFromStratIfAvail()` reduces debt from assets in following way:

1. Reduce debt from priority assets from last to first
2. Reduce debt from non-priority assets from first to last

This is same kind of inconsistency as mentioned above.

Changed reduction to first go thru non-priority strats/assets, then it goes reverse order thru priority strats/assets. This is now consistent.

Code corrected:

Hightop Financial responded:

Changed reduction to first go thru non-priority strats/assets, then it goes reverse order thru priority strats/assets. This is now consistent.

6.29 Consistency of isEnabled Flag on jUSD and Ripe Bond Configuration

Design | Low | Version 1 | Code Corrected

CS-RIPE-060

In `TreasuryEpochs._applyEpochRefresh()` the Ripe bond programs with the `isEnabled=False` are not being reset. However, the jUSD program is being reset even if `isEnabled=False`. Please confirm if this is the expected behavior.

Code corrected:

Both jUSD and Ripe bond programs are now being reset only if `isEnabled=True`.

6.30 CreditBorrow._getBorrowIntervalAvail Can Overflow Due to Config Change

Design | Low | Version 1 | Code Corrected

CS-RIPE-039

If user has borrowed the maximum amount of credit, and the maximum amount of credit is then lowered, the available credit can overflow.

```
available = maxBorrowPerInterval - _userBorrowInterval.amount
```

While this issue does not lead to problems in the system, users can misunderstand the revert reason.

Code corrected:

A `min(_userBorrowInterval.amount, maxBorrowPerInterval)` is used instead of `_userBorrowInterval.amount` to prevent overflow.

6.31 CurveDexData.STABILIZER_PROFIT_BUFFER Serves No Purpose

Design | Low | Version 1 | Code Corrected

CS-RIPE-058

The `CurveDexData.STABILIZER_PROFIT_BUFFER` is set to 1e18. It is used in `calcProfitForStabilizer()` to offset the profits of the stabilizer.

The `stabilize()` function that relies on the `calcProfitForStabilizer()` output, but it relies on difference between two profits, so the offset buffer has no effect.

In addition, the checks for `stabilize()` profitability will basically be off until the real profits reach 1e18. This breaks the purpose of the checks.

Code corrected:

The `CurveDexData.STABILIZER_PROFIT_BUFFER` is removed.

6.32 HQ Interface Violation

Correctness | Low | Version 1 | Code Corrected

CS-RIPE-013

The `HeadQuarters` contract has functions `mintRipe` and `mintJusd` that return bool. However, the `HqInterface`, which many contracts use to call the `HeadQuarters` contract, has the functions `mintRipe` and `mintJusd` that do not return anything. While this technically will be accepted by the compiler and would not cause any issues in production, with more changes in the system it can lead to undesired problems.

Code corrected:

Missing return value added to the functions `mintRipe` and `mintJusd` in the `HqInterface` contract.

6.33 Incorrect Calculation of the Debt Amount to Repay

Correctness | Low | Version 1 | Code Corrected

CS-RIPE-015

`CreditRedeem._calcAmountToPay()` will return that all the user's debt is redeemable if the debt amount is smaller than `collValueAdjusted`. This is incorrect as this would mean, that the higher the collateral value of a user is the more debt can be redeemed. The collateral value is already adjusted by the target ltv. Consequently, if the condition: `_debtAmount <= collValueAdjusted` is `true`, no debt at all should be redeemable instead of all debt.

The severity of the issue is only rated low because before we only continue with the redemption of a user if the following condition is true:

```
_userDebt.amount * HUNDRED_PERCENT / borrowData.collateralVal < borrowData.redemptionThreshold
```

Assuming, the system's parameters are set correctly with:

```
redemptionThreshold > borrowData.ltv * (HUNDRED_PERCENT - _ltvPaybackBuffer) / HUNDRED_PERCENT
```

We should never end up in the condition mentioned at the beginning. Else, the issue would be more severe.

Same problem appears in `AuctionHouseCollection._calcAmountToPay()`.

Code corrected:

The unnecessary condition was removed from both functions.

6.34 Keeper Fee on Liquidating Users With Small Positions

Design | Low | Version 1 | Code Corrected

CS-RIPE-040

The keeper gets only `keeperFeeRation` of the debt position as a reward for calling `AuctionHouseCollections.liquidateUsers()`. If the user's total amount of debt to be liquidated is small, the keeper also has a small incentive to liquidate the debt. The keepers have almost constant gas costs for the liquidation and, hence, it might not be profitable to liquidate small positions. Such "dust" positions usually accumulate with a certain rate that must be accounted for.

Code corrected:

Hightop Financial responded:

- Added `minKeeperFee` param (AuctionHouseConfig.vy) to make sure even on small debt positions, the liquidator will get a minimum payment amount.
- Also added `minDebtAmount` param (CreditConfig.vy) to reduce risk of user ever having small/dust debt position — and reduce risk of protocol eating the `minKeeperFee` cost (if user debt/collateral is so small)

The risk for small/dust amount remaining is reduced with these measures but not completely eliminated.

6.35 Leftover Approvals Might Break Some Tokens

Design | Low | Version 1 | Code Corrected

CS-RIPE-016

Some tokens like USDT have built-in protection against the approval race conditions. This is done by requiring the allowance to be set to 0 before setting it to the new value. If the contracts give approval to a spender, but spender does not use the full allowance, the leftover allowance will be stuck in the contract. Next time the same call path will be blocked due to the allowance being non-zero. Some code places where such leftover allowance might occur are:

- `EndaoStabilizer.addPartnerLiquidity()` for the `_altAsset`
- approvals prior to `Rolodex.transforAsset()`
- approvals as in `YearnV2.depositTokens()`, if vault does not use the full allowance
- `CurveDexAddLiq` and `CurveDexRemoveLiq`

Code corrected:

Approval is set to 0 after usage.

6.36 Open TODO

Design | Low | Version 1 | Code Corrected

CS-RIPE-041

`AuctionHouseCollections` has an open TODO comment that needs to be addressed before the deployment of the contract.

```
# TODO: build auction mechanism for assets without reliable price feed (i.e. Ripe governance token)
```

Code corrected:

Comment was removed as the code is already implemented.

6.37 Rebasing Tokens Transfer Corner Case CODECORRECRebasing Tokens Transfer Corner Case

Design | Low | Version 1 | Code Corrected

CS-RIPE-017

Many rebasing tokens such as stETH, and AAVE aTokens have internal conversion between value and shares on transfers. This might lead to 1-2 wei less difference between value received and value sent:

https://docs.lido.fi/guides/lido-tokens-integration-guide/#1-2-wei-corner-case

The Ripe system in some scenarios has multiple hops in the transfer path, and the 1-2 wei difference might lead to a failure in the transfer. E.g. during deposit user might specify value X, but strat pool will receive X-1 wei, and would then not be able to forward X tokens to the strategy wallet.

Code corrected:

Balance checks before and after call were introduced to query the exact amount of tokens that were transferred.

6.38 Redundancy in `CreditByAsset._reduceDebtFromAssetIfAvail()`

Design | Low | Version 1 | Code Corrected

CS-RIPE-042

In this function the `debtAvailToReduce` is computed as:

```
debtAvailToReduce = borrowSlip.existingUserDebt - borrowSlip.maxDebtForAsset
```

This is the same as already computed `borrowSlip.debtNeedsRedistro`. This computation is redundant and complicated code readability.

Code corrected:

Redundant computation was removed.

6.39 Ripe Bond Config With Index 0

Design | Low | Version 1 | Code Corrected

CS-RIPE-019

The `TreasuryConfig.addRipeBondProgram()``assigns indexes starting with 1. The index 0 is not used. However, the ``TreasuryConfig.setRipeBondConfig()` that `TreasuryDirect` calls has no such restriction. The `TreasuryConfig.getRipeBondProgram()` also will return index 0 for all the bonds that are not set. Due to upgradable nature of the contracts, it is not guaranteed that the `TreasuryDirect` will never call `setRipeBondConfig` with index 0. Thus, the `TreasuryConfig` invariant that the index 0 is not used is not enforced.

Code corrected:

In `setRipeBondConfig()` a check to prevent the index of being 0 was added.

6.40 StabilityPool.refreshStabilityPoolShares Rounding Error

Design | Low | Version 1 | Code Corrected

CS-RIPE-043

The `refreshStabilityPoolShares()` function in the StabilityPool contract recomputes how much pool shares each user has.

```
newUserShares = assetBalance * prevUserShares / prevTotalShares
if newUserShares < minBalance:
    newUserShares = 0
```

However, this is done using integer division, which leads to rounding errors. The magnitude of the error is bigger in tokens with low decimals. For example, USDC with 6 decimals will have a big error and user with small share of the pool will be "rounded to 0", even if the worth of position is still significant. Having a precision tha depends on the decimals of the token is not a good practice.

Code corrected:

`refreshStabilityPoolShares()` function in the `StabilityPool` contract was removed.

6.41 TellerDeposit desiredDepositAmount Wrong Units

Design | Low | Version 1 | Code Corrected

CS-RIPE-020

In `TellerDeposit._depositTokens()` function the check in case, when the deposit is not direct, is wrong. Units of `bundle.minDepositSize` is always in an asset value, but `depositAsset` is not. So, the configuration of `DepositConfig.minDepositSize` correctly won't be possible if more than one asset is used for the deposit.

```
assert desiredDepositAmount >= bundle.minDepositSize # dev: deposit is too small
```

Code corrected:

A proper check was added to ensure that the deposit amount is greater than or equal to the minimum deposit size:

```
assert stratAmount >= bundle.minDepositSize
```

6.42 TreasuryConfig RipeBondConfig.minEntitledRatio Cannot Be Altered

Design | Low | Version 1 | Code Corrected

CS-RIPE-021

Once the Ripe bond program is added via `addRipeBondProgramm()`, all the config parameters except `asset` and `minEntitledRatio` cannot be altered via setters. Thus, after the asset is added, the `minEntitledRatio` cannot be changed. Since the Ripe bond programs are extended and never

recreated, this limits the flexibility of the configuration. In contrast, in the jUSD bond program, the `minEntitledDiscountRatio` config param can be changed.

Code corrected:

Now, `setMinEntitledRatio` allows to change the parameter.

6.43 TreasuryConfig vestingLength Unchecked Setters

Design | Low | Version 1 | Code Corrected

CS-RIPE-022

In treasury Ripe and Jusd bond programs the `juiceScoreDecay` should always be greater than `vestingLength` when the program is added. However, the setters in `TreasuryConfig` for the `vestingLength` param do not enforce this invariant. The setters are:

- `setJusdBondVestingLength()`
- `setRipeBondVestingLength()`

Code corrected:

In `_purchaseRipeBond` and `_purchaseJusdBond` the decay block is now set to `bond.decayBlock = maturityBlock + config.juiceScoreDecay`. Consequently, the `bond.decayBlock` is still not strictly greater as `juiceScoreDecay` can be zero.

6.44 TreasuryConfig.canClaim Is Ignored by Janitorial

Design | Low | Version 1 | Code Corrected

CS-RIPE-023

The `TreasuryClaims.claimAllBondProceeds()` function checks the `TreasuryConfig.canClaim` flag. However, this flag is ignored by `claimBondsFromJanitorial()`. Any user can claim bonds via the `Janitorial.claimAll()`, regardless of the `TreasuryConfig.canClaim` flag.

Similarly, the `TreasuryClaims.claimBondsForUser()` function ignores the `TreasuryConfig.canClaim` flag. But in this case, this is a privileged function.

Code corrected:

The function `claimBondsFromJanitorial` was removed. The call to `Janitorial.claimAll()` will not claim as in the nested call `_cleanHouse -> updateTreasuryDuringJanitorial -> claimBondsDuringCleanUp` will check if `TreasuryConfig(_treasuryConfig).canClaim()` is `true`. `TreasuryClaims.claimBondsForUser()` ignores this setting, it but is privileged function.

6.45 Unclear Purpose of `lastModified`

Design | Low | Version 1 | Specification Changed

CS-RIPE-044

Multiple structs define a field called `lastModified`:

- `EndaoStratConfig`
- `EndaoStableConfig`
- `VoteDepositConfig`
- `JuicePositionConfig`

The variable is set in the setter function but not used anywhere in the code. The only function using the field is `BoardRoom._acceptSubDao` in the `ContractInfo` struct. With `description` and `version` there are two other fields for informational purpose only. Which each taking a storage slot.

Specification changed:

Hightop Financial explained that they want to keep the field for off-chain viewers/apps/users to see when certain things were modified, without needing to look through events.

6.46 Unnecessary Minimum Calculation

Design | Low | Version 1 | Code Corrected

CS-RIPE-024

In `AuctionHouseFungible._getUserDiscount` checks if `auctionConfig.minEntitled != 0` and if so get the minimum between `_generalDiscount` and `_generalDiscount * auctionConfig.minEntitled / HUNDRED_PERCENT`. In all places in the current code base, `auctionConfig.minEntitled` is checked to be strictly less than `HUNDRED_PERCENT`. Therefore, the minimum will never be `_generalDiscount`. Computing min of two values is redundant.

```
floorDiscount = min(_generalDiscount, _generalDiscount * auctionConfig.minEntitled / HUNDRED_PERCENT)
```

Code corrected:

The redundant code was removed.

6.47 Unnecessary Parameters in Calldata and Function Parameters

Design | Low | Version 1 | Code Corrected

CS-RIPE-025

Multiple functions and structs contain parameters that can be queried cheaply. These params are `block.timestamp`, `block.number`. Examples where these params are used:

- `struct MainAddys`, `currentBlock` and `currentTime` fields.
- `RoloDex.addReferencePoolSnapshot()`

- `TreasuryEpochs.refreshBondEpoch()`
- `LedgerManager.touch()`

Passing these parameters to other functions is unnecessary and can be replaced with a call to `block.timestamp` and `block.number`. This will save gas and improve the code's readability.

Code corrected:

Hightop Financial revised the code and now uses the cheap params when possible, according to them.

6.48 Unused Functions

Design | Low | Version 1 | Code Corrected

CS-RIPE-026

- The two functions `setUserGovPoints` and `setAssetGovPoints` are present in `RipeGovStratData` and only accessible by the `RipeGovStratPoints` contract but never called.
- `LootBoxData.setGlobalPoints`, `setAssetPoints`, `setUserPoints`, `setUserBorrowPoints`, `setGlobalBorrowPoints` should be called by the LootBoxPoints contract but, never is.
- `AuctionHouseData.disableFungibleAuction` can only be called by `AuctionHouse(self.depts.auctionHouse()).auctionHouseFungible()` but is not called in the contract.
- Similar `AuctionHouseData.disableNftAuction` is not used in `auctionHouseNFT`.
- Similar `AuctionHouseData.recordAuctionPurchase` is not used in `self.depts.auctionHouse()`.

Code corrected:

All functions were removed.

6.49 Unused Storage Variables

Design | Low | Version 1 | Specification Changed

CS-RIPE-027

When a user gets liquidated, the `AuctionHouseData` registers and saves the user in `initFungibleAuction` calling `_registerFungibleLiqUser`. This will affect three storage variables `fungLiqUsers`, `indexOfFungLiqUser` and `numFungLiqUsers`. The need for the variables seems unclear. These variables are public and could be queried, but no other contract in the code base relies on them.

Specification changed:

Hightop Financial specifies that they need to have these variables as they are used for front-end, web-app or other off-chain monitoring purposes.

6.50 `CreditData.totalUserDebtByAsset` Is the Same as `UserDebt.principal`

Design | Low | Version 1 | Code Corrected

CS-RIPE-045

The `CreditData` contract stores the `UserDebt` struct per user. This struct has a `principal` field that stores the uint256 debt of the user. The same contract also has the `totalUserDebtByAsst` mapping which stores the `uint256` debt per user. Every time the user debt is updated, both the `UserDebt.principal` and the `totalUserDebtByAsset` are updated. This is a redundant design, that complicates the reasoning about the code and increases the risk of errors.

Please also note that the name `totalUserDebtByAsset` is misleading, as it is not the total debt per asset, but the total debt per user.

Code corrected:

The mapping `totalUserDebtByAsset` was removed.

6.51 Gas Optimizations

Informational | Version 4 | Code Corrected

CS-RIPE-067

Here is a **non-exhaustive** list of gas (and bytecode) optimizations:

1. `didUpdateBoardRoomOrDepts` is loading `boardRoom` in memory for most of the Governors, but some are missing
2. Some GenericParamData elements are not used for some Governors:
    1. `AuctionHouseGovA.GenericParamData.flag`
    2. `CreditGov.GenericParamData.addr`
3. `Csuite._isMemberOfAnyCommittee` doesn't need `isMember` and can return directly.
4. `ElectionHall._isCandidateEligible` doesn't need the check `isMemberOfAnyCommittee` since there is a stronger `isMember` check after that.
5. `CSuite._canBecomeDesignated` queries full `committeeData` just to check `numMembers`, `CSuiteData.getMembershipCount` can be used instead. Please note that checks need to take into account `-1` that `getMembershipCount()` performs by default.
6. `ElectionHall._updateVotingPower` does consecutive calls to `ElectionData` to read and update user power. Such can be tried to be avoided. Also, `voteData` queried in a first call is changed in a second call, and technically stale struct is used later for deletion checks. But since the `alloc` field cannot be changed in `updateVotingPower` call, this is fine.
7. `adjustVetoWeight` has check: `p.results.didVeto or block.number >= p.times.executeStart:`. However, vetoed proposals are immediately deleted. `didVeto` check is not needed.
8. `PropHouse.createProposal` does two consecutive calls to `RuleBook` to check two things. These calls can be unified.
9. `transferTokens` uses `min` when, because of the previous `if` it is superfluous.

10. `updateVotingPowerRipeDistro` and `updateVotingPowerEndaoDistro` are guaranteed that `userData.allocs.total != 0` thanks to the early check, so this inequality can be assumed to be true in the rest of the function.

Code corrected:

Items 1-10 were all corrected.

6.52 Lacking Sanitization for Input Data of Proposals

Informational | Version 4 | Code Corrected

CS-RIPE-068

Parameters of proposals should be sanitized as much as possible to prevent unexecutable proposals, when possible the test should be the same as the one used in the proposal execution function.

(Also, sometimes when it's not possible, sanitization can still be done, for example forbidding zero addresses or zero values for `uint256`.)

Randomly picked examples where sanitization is missing:

- See `VariousPriceSourcesGov.proposeAddCurvePrice` for an example where an `asset` can be address(0), but the proposal execution function will revert in this case.
- See `JuiceBoostNFTGov.proposeMintBoostNft` for an example where a `_juiceBoost` can be checked to be between 1 and 100 percent.

Code corrected: Parameters validation was improved.

6.53 Proposal With Failed Execution Does Not Emit Event

Informational | Version 3 | Code Corrected

CS-RIPE-069

Only successfully emit `ProposalExecuted` event. Failed proposal execution does not emit any event and thus it might be hard to track the failed proposal execution.

Code corrected:

A specific event is emitted when a proposal fails to execute.

6.54 Check of Balance After Transfer From TellerDeposits

Informational | Version 1 | Code Corrected

CS-RIPE-028

The `StabilityPool.depositTokens()` function has following check:

```
assert ERC20(_asset).balanceOf(self) >= _amount
```

However, the `depositTokens()` is only callable by the `TellerDeposits` contract, which is trusted. The only time this check can be useful is if the TellerDeposits contract is upgraded to a bad version or tokens are rebasing or have fees.

Code corrected

The check was removed from the code.

6.55 CreditBorrow.borrowForUser Check `isActivated`

Informational | Version 1 | Code Corrected

CS-RIPE-029

The `self.isActivated` checked in `CreditBorrow.borrowForUser()` as well as in the `_borrow()`. This wastes gas and can be checked in the `_borrow()` only.

Code corrected

The check was removed from the `CreditBorrow.borrowForUser()` and is only present in `_borrow()` now.

6.56 Dev Comment Bigger Vs. >=

Informational | Version 1 | Code Corrected

CS-RIPE-030

`TreasuryConfig.addRipeBondProgram()` has following assert statement:

```
assert _juiceScoreDecay >= _vestingLength # dev: juice score decay must be bigger than vesting length
```

The comment is misleading. The check is actually for `_juiceScoreDecay >= _vestingLength` and not `_juiceScoreDecay > _vestingLength`.

Code corrected:

Comment was fixed.

6.57 Dust in Ripe Bond Programs

Informational | Version 1 | Code Corrected

CS-RIPE-031

Due to how the TreasuryRipeBonds is computing the `ripePayout`, a small dust amount can be left unsellable in the contract.

```
ripePayout: uint256 = assetAmount * userRipePerUnit / (10 ** convert(ERC20Detailed(_asset).decimals(), uint256))
assert ripePayout != 0 # dev: bad deal, user is not getting any Ripe
```

If the `assetAmount * userRipePerUnit` is low, the `ripePayout` will be 0 and no one will be able to buy that last wei. The `_isEpochDepleted` check to reset the epoch early in TreasuryEpochs will not be triggered with dust amounts.

Code corrected:

If 10**18 of assets is left to be sold, the Ripe and jUSD bonds will be considered depleted. It does not need to be strictly 0.

6.58 Incorrect Comments

Informational **Version 1** **Code Corrected**

CS-RIPE-032

- In `CreditRepay._finalizeAmount()`: Technically not only jUSD handled by this function

```
assert senderAmount != 0 # dev: no jusd balance
```

- In `_updateDebtData` there is the comment: The function `updateUserDebt` does not exist in the code base.

```
# turn off any auctions (if applicable) -- we also do this during `cleanHouse` -> `updateUserDebt()`
```

- `AuctionHouseFungible._handlePayment()`. The value can be insufficient or too much. The comment mentions only the insufficient case.

```
assert self._valueIsCloseEnough(amountToPay, amountReceived) # dev: payment was insufficient
```

- `TellerWithdraw._transformAsset()` has natspec for `@return` type that is wrong. The function returns single value, not a tuple.

```
@return Tuple of the transformed asset address, amount, and a flag indicating if alternate pricing is needed.
```

Code corrected:

All comments were fixed.

6.59 Partner Could Block Disabling a Stabilizer Pool

Informational **Version 1** **Code Corrected**

CS-RIPE-047

In `EndaoStabilizer.disableStabilizerPool` the lp tokens are transferred to the partner. In the case of a token that implements callback hooks to the receiver, the partner could let the call fail. This issue is just informational as we assume the partner always needs to agree with disabling a pool due to the fact that a partner is allowed to disable pools, they partner with at will anyway.

Code corrected: This part of the code was removed.

6.60 Potential Vulnerable Voting

Informational **Version 1** **Code Corrected**

CS-RIPE-034

This issue is a potential issue to raise awareness of the fact that to fix the DOS vector in the voting loop, a simple batching will lead to the following issue:

If votes are counted in batches because of gas issues and split over multiple transactions, attackers might be able to transfer voting power. This needs to be considered when fixing the gas intense loop over all potential voters.

Code corrected:

The voting was rewritten and does not require loops anymore.

6.61 Redemption Fee Can Be Less Than Designed

Informational **Version 1** **Code Corrected**

CS-RIPE-048

In `CreditRedeem._redeemOnDebtPosition()` function the `actualRedeemVal` amount is not always guaranteed to be less than or equal to `min(maxRedeemableOnDebt, remainingRedemptionVal)`. The `jusdAmount` that redeemer needs to provide is computed as:

```
jusdAmount: uint256 = min(maxJusdPayAmount, totalRedeemedVal * HUNDRED_PERCENT / (HUNDRED_PERCENT - redemptionFee))
```

If it happens that `actualRedeemVal` is greater than `min(maxRedeemableOnDebt, remainingRedemptionVal)`, `totalRedeemedVal` plus the fee will be more than `maxJusdPayAmount`. Thus, redeemer will not pay full fee value for such redemption. In existing strategies the `actualRedeemVal` can be greater only in theoretical case due to rounding errors.

Code corrected:

Hightop Financial response:

The redemption code was slightly reworked so that as it iterates thru strats/assets, the usdValue is subtracted from remainingValue So the risk is mitigated. This is really only possible on the last item in the loop before remainingValue reaches zero. Should only be dust if this occurs.

6.62 Transfer to Self in StratData Contracts

Informational | Version 1 | Code Corrected

CS-RIPE-061

The function `transferBalance()` in the `StratData` contracts can return `isDepleted = True`, even if the from and to are the same address. At this moment `LedgerManager._transferAssets()` does not allow transfers between the same addresses and thus the `transferBalance()` function should not be called with the same params. Have you considered enforcing no transfers between the same addresses in the `transferBalance()` function? `LedgerManager` can be upgraded and allow transfers the between same addresses, but then the `isDepleted` flag would be misleading.

Same question applies to the function `transferNft()` in the `FloorErc721Data` contract.

Code corrected:

A check was added to prevent transfers between the same addresses.

6.63 Unused Function `transferBalance` in Stability Pool

Informational | Version 1 | Code Corrected

CS-RIPE-056

The function `StabilityPoolData.transferBalance` is not used anywhere in the code base. even though, it is external, it could only be called by the `StabilityPool` but it does not call it.

Code corrected

The function was removed.

6.64 Unused Return Value isSafeLtv

Informational | Version 1 | Code Corrected

CS-RIPE-049

In `CreditRedeem._redeemOnDebtPosition` a memory variable `isSafeLtv` is initialized to `False` and overwritten by the return value of the call to `CreditRepay.repayDebtDuringBulk`. The variable `isSafeLtv` has no functional use in the function and is not returned or emitted.

Code corrected:

The `isSafeLtv` was removed.

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 `RipeGovStratPoints` Uses Same Decimals Offset for Different Assets

Informational **Version 3**

CS-RIPE-070

User balances are scaled by the same DECIMAL_OFFSET for potentially different assets.

```
shares: uint256 = userBalData.amount / DECIMAL_OFFSET
```

At this point `RipeGovStrat` does not support any asset that does not have 18 decimals. However, if this changes in the future, the DECIMAL_OFFSET should be adjusted accordingly.

7.2 Number of Elements in Lists Is off by One

Informational **Version 1**

CS-RIPE-033

Ripe system tracks many arrays with the help of mapping(index=>data) and length field. As a design decision, the length of many such arrays starts with one. External systems and users that plan to use the Ripe system need to be aware of this feature. Examples of such arrays:

- Strategies and the ledger are tracking the number of participants. The variable `numParticipants` starts to count from `1`.
- `TreasuryConfig.numRipeBondPrograms`
- `AuctionHouseData` contract: `numNftLiqUsers` and `numNftLiqUsers`
- `LootboxConfig.numVoteDepositAssets`

7.3 USD Price Fluctuations

Informational **Version 1**

CS-RIPE-035

By evaluating the prices of tokens in USD, the token amount might stay constant but the token price might fluctuate extremely. This would affect the valuation of tokens and strategies. E.g., profit or losses might be reported due to changing USD evaluation and not a real increase of underlying funds. If `EndaoManager` mints jUSD because of a spiked USD evaluation the system might overvalue the profits and mint too much jUSD.

7.4 User Actions Might Lead to Unexpected Results

Informational **Version 1**

CS-RIPE-036

In many cases, the Ripe system opts out of enforcing limits silently. For example, if users try to borrow more than they can, the system will simply let them borrow the maximum amount they can. However, users need to be aware of such a design choice. Similar soft limits are also used for bonds, liquidations, and other operations. Users might need to use wallets that revert if the outcome of the operation is not desirable.

7.5 User Indexing State Variables

Informational **Version 1**

CS-RIPE-057

There are often indices used. E.g., in the strategies for the assets and participants in `StabilityPoolData` (`indexOfAsset`, `stabAssetHoldingsId` and `indexOfParticipant`). In most cases it seems like the indexing is not needed or could be implemented more efficiently.

7.6 Loops in Vyper

Informational **Version 1** **Code Partially Corrected**

CS-RIPE-046

First, Ripe uses range iterations in many places. https://docs.vyperlang.org/en/latest/control-structures.html#range-iteration However, vyper has `for i in range(start, end, bound=N):` syntax, that can be used to skip the first iterations. This would be more efficient than the `if i == 0: continue` syntax that Ripe uses. Terminating based on the `end` value is also more efficient and readable than `if i == numUserAssets: break` syntax.

Second, the optimisations, where variables used inside the loop are declared outside the loop, are not necessary in Vyper. This will technically not allow vyper to deallocate the memory used by the variable after the loop. In vyper, unlike solidity, everything declared inside loop bodies is defined in a separate scope and deallocated after the loop ends.

Code partially corrected:

Hightop Financial responded:

> This is only partially fixed. We still need to add START param (to skip 0 index), which requires updating vyper, which had some breaking changes, deprecations (enums → flags, struct instantiation, etc). Once we update all dependencies, and do the required changes to support 0.3.11, we will add the START param to the for loops where we skip 0 index.

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 Arbitrage on `swapAssist()`

Note Version 1

The `EndaoAllocate.swapAssist()` function uses the PriseDesk to determine the price of the swap. This price can be different from the price of the swap on some onchain DEX. Arbitrageurs might front run the transaction to get the delta. In this case the discrepancy cannot be detected by the system as it checked the price and traded at the price from the oracle. But the price at the DAX might have been even better.

8.2 Frontrunning Liquidations

Note Version 1

Liquidations and auctions can be front-run by repaying or updating a user's debt position to a healthy LTV or by another liquidation. In this case, the user liquidation will be skipped. However, the liquidators will still need to `cleanHouse` and use some gas. Liquidators need to be aware of this.

8.3 Theoretically No Best Endao Strategy Can Be Returned

Note Version 1

In `_getBestStratForDeposit` the `EndaoAllocate` contract tries to figure out the "best" strategies that should get the new funds. However, if `stratData.enforceAllocMax` is `true` for all strategies, and the `usdValueNeeded` per strategy is smaller than the total `_depositUsdValue`, the funds will not be distributed as no best strategy would be returned.

8.4 Transaction Ordering Dependency in jUSD Reward Distribution

Note Version 1

If jUSD is burned from the Endao wallet or new jUSD is minted depends on the jUSD balance of the wallet when `endaoManager.moneyPrinterGoBrr` is called because in `_deductJusdRevenue` queries `balanceOf` jUSD. The jUSD balance of the wallet fluctuates depending e.g., on swaps, redemptions and other operations. E.g., swaps can be triggered by users, who now can influence if jUSD is minted or burned. Depending on what operations are done before or after `moneyPrinterGoBrr` the result might differ.