Analysis

□ Analysis - Ethereum Credit Guild

Overview

Ethereum Credit Guild (previous Volt Protocol) protocol is focused on trust-minimized pooled lending. Pooled lending is prevalent on Ethereum, involving numerous lenders depositing assets into a pool governed by tokenholders and relying internal credit multiplier accounting. The Credit Guild aims to scale pooled lending benefits while reducing traditional trust assumptions. It adopts an optimistic governance model, allowing a minority of GUILD holders to propose and veto lending terms, promoting a decentralized decision-making process. The protocol employs mechanisms to handle liquidations and bad debt more efficiently, ensuring fair treatment of lenders and maintaining market functionality during adverse events.

The key contracts of Ethereum Credit Guild protocol for this audit are:

- **LendingTerm**: Contract serving as an entry point for borrowers, this contract is used to borrow, repay (either full or partial), provide more collateral, call loans, and in case loan is totally insolvent - forgive it, which makes the whole Credit token market to suffer the consequences by reducing its price
- **LendingTermOnboarding**: contract inheriting **GuildGovernor** and extending its functionality by providing a way to create new lending terms, acts as a lending term factory. In addition there is **OZClones** used to greatly reduce the gas costs by cloning the contract instead of deploying a new one. This contract also calls Governor contract to propose **only terms which are not onboarded yet.**
- **LendingTermOffboarding**: **Non-governed, non-timelock** constrained contract, used for off-boarding lending terms which are not considered healthy and clean the system from it, when there are no pending loans. Everyone can propose terms that are already used in the system and guild token holders can vote whether to remove the gauge and auction all the loans opened from it.
- **ProfitManager**: Profit distributor contract tied to CreditToken, each **CREDIT** token has its own instance. This contract takes care of lending terms by increasing/decreasing the first-loss capital provided by the stakers (termSurplusBuffer and surplusBuffer) and either distributing interest paid by borrowers or initiating bad debt, caused by the need system to handle the unpaid loan interest, which slashes the GuildToken stakers in this market. In case there is a loss that empties all the first-loss capital (termBuffer + market buffer) this contract also decreases the ratio of CreditToken:pegToken (CreditMultiplier)
- **AuctionHouse**: Contract used to create new Dutch auction for loans that are not in healthy state - partial repay not on time or misbehaviour which lead to market off-boarding. It also calculates how much collateral will receive the bidder, the borrower and how much debt will have the bidder to repay in order to claim the collateral.
- **SimplePSM**: Contract used to maintain the peg between credit and reference asset, there borrowers can redeem their credit tokens received and get the equivalent (USDC:gUSDC) based on the credit multiplier. It also has internal peg token balance accounting and removes donation attack possibilities.
- **SurplusGuildMinter**: Contract which aggregates users who have credit tokens and want to enlarge the borrowing capabilities of a term.

  Stakers give credit tokens to term's buffer and this contract receives guild tokens which are used to support the term, this way they increase the debt ceiling, resulting in increased borrow capabilities of the market. It also distributes rewards in form of Credit and Guild tokens to stakers. Unstake is the opposite - decreasing the debt ceiling and burning the guild tokens. There is slashing in case of bad debt which leaves all the user credit and guild tokens of stakers for the system.
- **RateLimitedMinter**: This contract limits the minting of Credit and Guild tokens by calculating time-constrained buffer based on the mint/burn ratio of the tokens. There will be one instance for the Guild token, and one per every distinct Credit token in the system.
- **CreditToken**: **ERC20 tokens** with rebasing mechanism representing the debt in certain market. Holders of this tokens are **suppliers and borrowers**. Used as votes token in the governance as well.
- **GuildToken**: **ERC20 token** used for governance and weight allocation. Non-transferable at the beginning, old Volt protocol users will receive them at the beginning and stakers from SurplusGuildMinters will receive additional tokens for increasing the weight of a given term.

System Overview

Scope

- `Core` - core contract that manages the access control of the system.
- `CoreRef` - abstract core contract which is inherited by contracts, provide pausing, unpausing and emergency action functionality.
- `CoreRoles` - library that provides all the roles hashed.
- `GuildGovernor` - governor contract based on OZ, responsible for the system voting.
- `GuildVetoGovernor` - veto Governor where DAO votes to cancel an action queued in a timelock can be created
- `GuildTimelockController` - timelock contract based on OZ, responsible for delaying proposal executions for the voting period.
- `LendingTermOffboarding` - permissionless, non-timelock contract, responsible for proposing and execution of active terms.
- `LendingTermOnboarding` - permissionless, timelock constrained contract, responsible for proposing and onboarding new lending terms.
- `ProfitManager` - accounting contract, manages reward indexes for lenders and borrowers and also distributes **CREDIT** token rewards.
- `ERC20Gauges` - abstract contract, inherited by Guild token, exposes functionality for guild token holders to vote for terms and increase their weight and reward ratio.
- `ERC20MultiVotes` - abstract contract, inherited by both Credit and Guild token, allows token holders to vote for various proposals in the system.
- `ERC20RebaseDistributor` - abstract contract, inherited by Credit token, exposes rebasing mechanism used to distribute rewards regularly for 30 days period, instead of all-at-once.
- `CreditToken` - ERC20 token contract, pegged to the reference asset in the `SimplePSM` contract (USDC - reference asset, gUSDC - credit token), with the ability to vote in the system and earn passive yield.
- `GuildToken` - ERC20 token contract, used as a main governance asset, with the ability to vote in the system, increase the weight of a given term and earn rewards from the interest payed from borrowers of the term.
- `AuctionHouse` - contract representing **Dutch Auction**, loans who are insolvent are being auctioned from it, also manages the collateral received and debt repaid based on the time passed since loan is called.
- `LendingTerm` - contracts that are used from borrowers to create new loans and repay them, they are also accountable for calculating the interest and notifying the profit manager.
- `SimplePSM` - contracts used to maintain peg between credit token and reference asset.
- `SurplusGuildMinter` - contract used from stakers to stake credit tokens and vote for lending term with guild tokens minted to this contract.
- `RateLimitedMinter` - contract that implements limitation on Guild and Credit tokens by constraining them with buffer.

Privileged Roles

1. Core (roles, expected to be transferred to automatic governance processes in a later stage):

- `GOVERNOR` - main role, which also is admin of all other roles, used to set various system critical parameters.
  - in `CoreRef`
    - set new Core contract
    - perform emergency actions (similar to MakerDAO)
  - in `GuildGovernor`
    - set voting delay (between two proposals)
    - set voting period (of single proposal)
    - set proposal threshold
    - set quorum
  - in `GuildVetoGovernor`
    - update timelock
    - set quorum
  - in `LendingTermOnboarding`
    - set quorum
  - in `LendingTermOnboarding`
    - whitelist lending term implementations
  - in `ProfitManager`
    - initialize
    - set minimum borrow amount
    - set gauge weight tolerance
    - set profit sharing configs
  - in `LendingTerm`
    - forgive a loan
    - change auctionHouse contract
    - set hard cap of a lending term
  - in `SimplePSM`
    - pause Credit token redemptions when lending term get off-boarded
  - in `SurplusGuildMinter`
    - set mint ratio
    - set reward ratio
  - in `GuildToken`
    - enable transfers
    - change ProfitManager used
  - in `RateLimitedMinter`
    - set rate limit per second
    - set buffer cap
- `GUARDIAN` - guardian role, used to pause certain contracts and cancel potentially malicious proposals.
  - in `CoreRef`
    - pause/unpause
  - in `GuildGovernor`
    - cancel proposal in progress

2. Token Supply :

- `CREDIT_MINTER` - role, given to `SimplePSM` contracts, will allow Credit token minting.
- `RATE_LIMITED_CREDIT_MINTER` - role, given to `LendingTerm` , will allow borrowers to mint Credit tokens through `RateLimitedMinter` .
- `GUILD_MINTER` - role, given to the `RateLimitedMinter` contract, will allow Guild token minting when staking from `SurplusGuildMinter` .
- `RATE_LIMITED_GUILD_MINTER`

3. GUILD Token Management:

- `GAUGE_ADD` - role, given to `LendingTermOnboarding`, will be responsible for adding new terms/gauges into the system.
- `GAUGE_REMOVE` - role, given to `LendingTermOffboarding`, will be responsible for removing terms/gauges from the system.
- `GAUGE_PARAMETERS` - role, given to the **deployer and DAO Timelock**, will be responsible for changing max allowed gauges per user.
- `GAUGE_PNL_NOTIFIER` - role, given to the `LendingTerms`, will be responsible for notifying the `ProfitManager` when users repay their loans or there is a bad debt which has to be handled.
- `GUILD_GOVERNANCE_PARAMETERS` - role, given to the **deployer and DAO Timelock** of the system, will allow him to extend the number of addresses any account can delegate voting power to.
- `GUILD_SURPLUS_BUFFER_WITHDRAW` - role, given to `SurplusGuildMinters` contracts, will be accountable for withdrawing from the surplus buffer when staker unstakes.

4. CREDIT Token Management:

- `CREDIT_GOVERNANCE_PARAMETERS` - role, given to the **deployer and DAO Timelock** of the system, will allow him to extend the number of addresses any account can delegate voting power to.
- `CREDIT_REBASE_PARAMETERS` - role, given to the `SimplePSM`, will be responsible for managing the rebasing status of **credit token holder** who wants to enter and exit the rebasing mechanism.

5. Timelock Management:

- `TIMELOCK_PROPOSER` - role, given to `LendingTermOnboarding` contract, will be responsible for proposing new lending terms created from **GUILD** token holders.
- `TIMELOCK_EXECUTOR` - role, given to `address(0)` and everyone can execute passed proposals, will be responsible for executing proposal that meets the quorum.
- `TIMELOCK_CANCELLER` - role, given to the `GUARDIAN`, will be responsible for cancelling potentially malicious proposals.

Approach Taken-in Evaluating The Ethereum Credit Guild

Stage	Action	Details	Information
1	Compile and Run Test	Installation	Easy setup and commands provided for testing and deploying
2	Documentation review	GitBook	Provides a basic governance, staking and lending/borrowing overview without emphasizing the technical side
3	Contest details	Audit Details	Thorough details for the contracts and the idea of the protocol were provided. Known issues and possible attack scenarios are helpful.
4	Diagramming	Excalidraw	Drawing diagrams through the entire process of codebase evaluation.
5	Test Suits	Tests	In this section, the scope and content of the tests of the project are analyzed.
6	Manual Code Review	Scope	Reviewed all the contracts in scope
7	Special focus on Areas of Concern	Areas of concern	Observing the known issues and bot report

Codebase Explanation & Examples Scenarios of Intended Protocol Flow

All possible Actions and Flows in Ethereum Credit Guild:

One of the most important part of grasping the protocol is the `ProfitManager::notifyPnL`

`NotifyPnL` is used by `LendingTerms` to notify the `ProfitManagers` about interest repaid or bad debt occurred, the diagram provided explains how the function will behave and what will be the impact in the 2 scenarios (negative and positive amount passed as an argument).

Positive PnL (Interest repaid):

Rewards are split between 4 recipients, based on the percentages set by the governor:

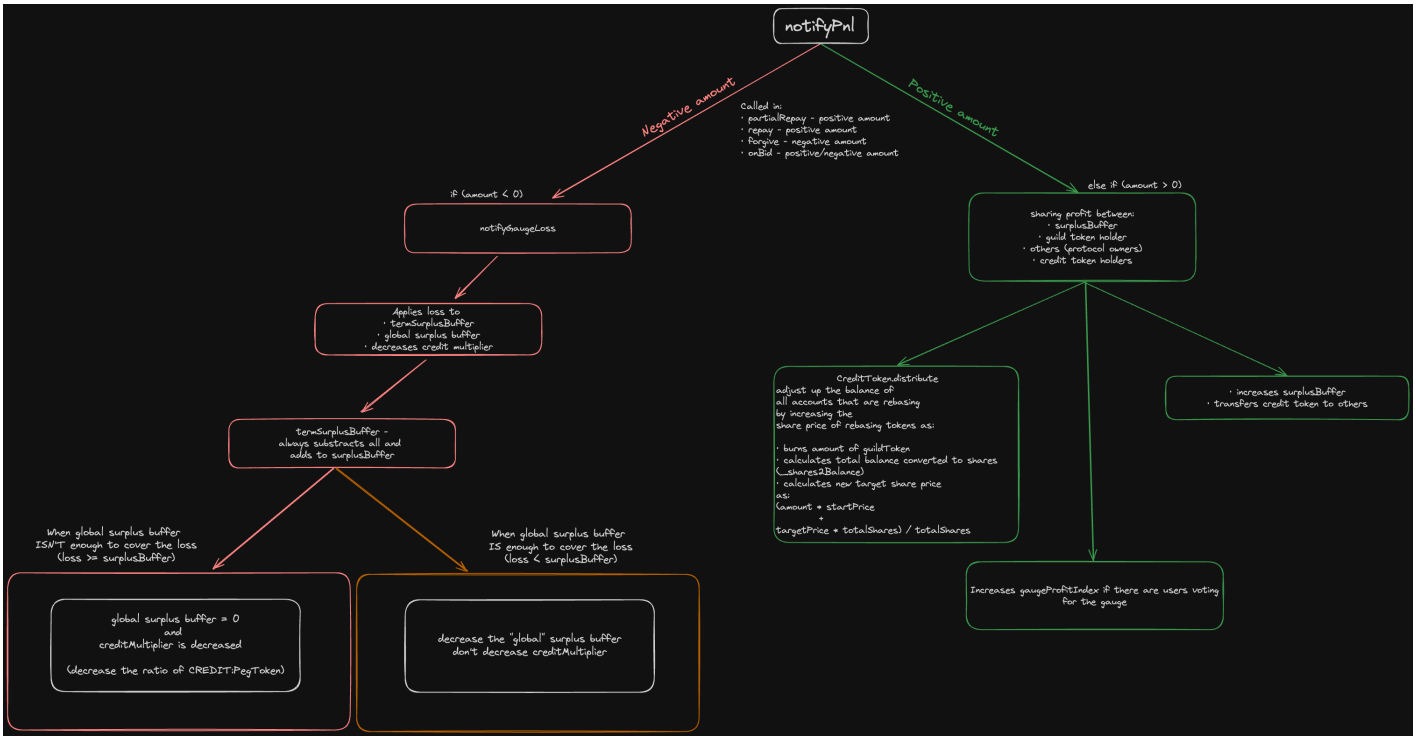
1. `surplusBuffer`
2. `otherRecipient` - can be set to any address.
3. `CREDIT` token holders, who are rebasing, in a drip-vested amount.
4. `GUILD` stakers for the specific term.

Negative PnL (Bad Debt):

In scenarios involving bad debt, the following steps are taken:

1. Notify the `GuildToken` contract to reflect the loss in the gauge.
2. Decrease the entire `termSurplusBuffer` if applicable.
3. Based on the magnitude of the loss:

- If the loss is smaller than the `surplusBuffer`:
  - Subtract the `loss` from the `surplusBuffer` and burn the corresponding amount.
- If the loss exceeds the `surplusBuffer`:
  - Deplete the `surplusBuffer` entirely and reduce the `creditMultiplier`.



1. Initial term creation (through `LendingTermOnboarding`)

Users who has at least 1M `GUILD` tokens create new lending terms by initiating proposals with different configurations:

```
address collateralToken
uint256 maxDebtPerCollateralToken
uint256 interestRate
uint256 maxDelayBetweenPartialRepay
uint256 minPartialRepayPercent
uint256 openingFee
uint256 hardCap
```

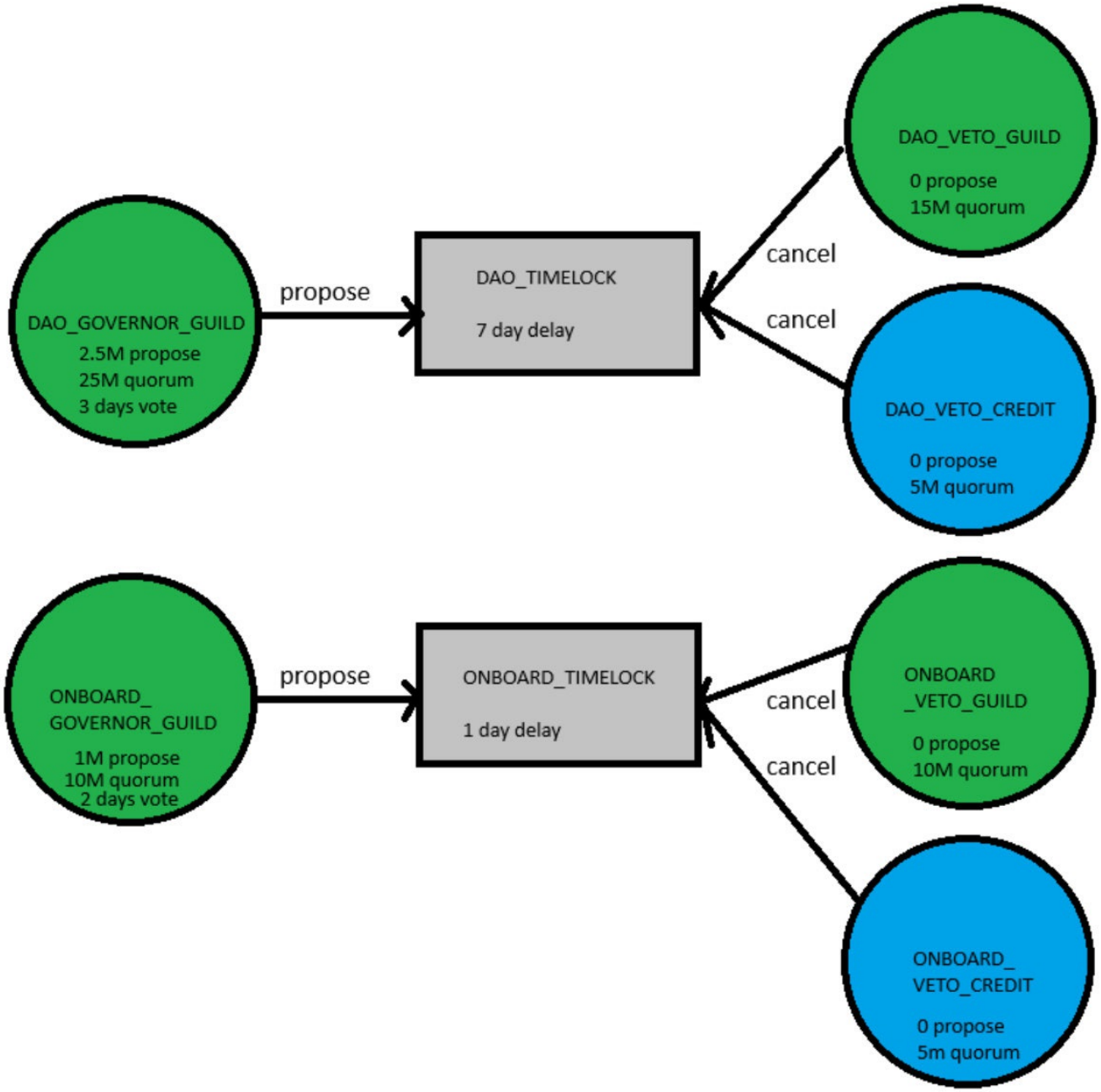
There are hardcoded contract addresses passed to every newly proposed lending term, which is wrong as there will be different market types, only `guildToken` and `auctionHouse` will be the same for all of them:

```
address profitManager
address guildToken
address auctionHouse
address creditMinter
address creditToken
```

OpenZeppelin clones library is utilized to provide gas-efficient way of deploying the contract. Still not onboarded the lending term is unusable until proposal has passed as it add the term to the appropriate market and gives the following roles to it:

- `RATE_LIMITED_CREDIT_MINTER` to mint CREDIT tokens through `RateLimitedMinter`
- `GAUGE_PNL_NOTIFIER` to notify the `ProfitManager` about loan repayment or loss that has to be realized.

In order for proposal to pass, quorum of 10M tokens should be fulfilled within two days. Another one day where the proposal is a subject of eventual veto by `GUILD` token holders. After that proposal is executed, roles are given and new borrows can occur.



2. Borrowing

Once term is onboarded users can start to initiate new loans if there is enough debt ceiling to satisfy the needs of borrowers. Loan is executed by providing collateral token amount and receive `CREDIT` token amount back based on the `creditMultiplier` for the current term (initially 1:1 ratio, but it can be lowered when there is a bad debt and no first-loss capital to handle the loss, for example multiplier of 50% means for 1 USDC provided user will receive 2 gUSDC tokens).

3. Repaying

Borrowers have the ability to either repay the full loan or part of it, there are also non-mandatory partial repayments (depending on the term's configuration), when loan is being repaid the total debt is calculated based on `interestRate` + `openingFee` + `borrowAmount`. After successful repayment principal is **burned and buffer is replenished by the same amount**. `ProfitManager` is notified for the repayment and **interest** is split amongst 4 configurable recipients:

- Surplus buffer
- Guild token holders
- Credit token holders
- Other users, such as the DAO treasury or upgraded `ProfitManager` contract.

4. Lending

Lenders in `Ethereum Credit Guild` are people who provide reference asset (ex. USDC, WETH) to the `SimplePSM` and get `CREDIT` token (ex. gUSDC, gWETH) and **eventually** enter the savings rate (**enterRebase** in `ERC20RebaseDistributor`), that way their balance is converted to shares with monotonically increasing `sharePrice` for 30 days. Rewards from lending are distributed when **notifyPnL** is called with positive amount as a result from borrowers paying their loan interests: (ProfitManager.sol#L379).

5. Staking

Users who want to receive `CREDIT` and `GUILD` tokens as a reward while increasing the terms surplus buffer (first-loss capital) and weight allocation can stake from

`SurplusGuildMinter` contract. To do so they have to provide `CREDIT` tokens and in exchange `GUILD` tokens, calculated by their ratio, will be minted to the contract. Stakers rewards are calculated with the help of indexes which are increasing when **notifyPnL** is called with positive amount as a result from borrowers paying their loan interests: (ProfitManager.sol#L383-L402).

User who staked through `SurplusGuildMinter` contract can claim their rewards from getRewards function.

There is also risk of **slashing** when loss is reported from **notifyPnL called with negative amount.** Every staker loses his `GUILD` tokens accrued since last claim, but can redeem the `CREDIT` tokens.

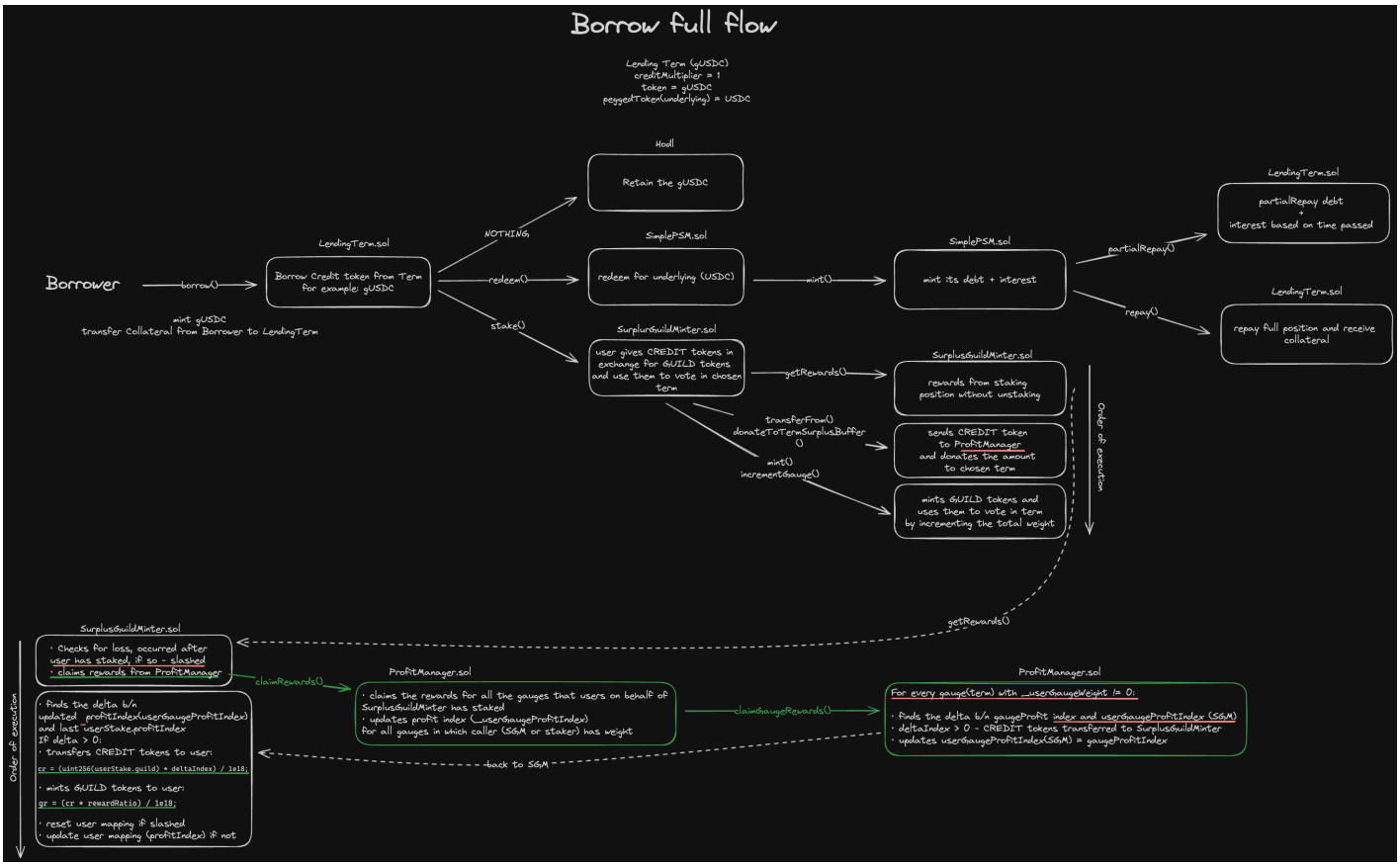


Diagram showing the architecture of the points above: 2, 3, 4, 5

6. Auctioning loans

Ethereum Credit Guild utilizes **Dutch Auction** system named `AuctionHouse`, used to call loans and allow users to bid for them.

Loan is eligible to be called in case there is missed repayment period or lending term is getting off boarded. In the second option all the loans in the system will be called and the goal will be to lead the issuance(total borrowed `CREDIT` tokens without counting the interest) to zero.

When loan is called auction is created initially offering buyer to repay 100% of the loan debt and get 0% collateral in return, gradually increasing the amount collateral until **midpoint** of the auction. Then 100% collateral is offered to buyer for 100% loan debt repaid. After **midpoint**, until the end of an auction 100% payment for the buyer is proposed, for decreasing percent of debt, opening a door for the arbitrageurs. Word auction is somehow misleading as the first person who bids wins the auction (mitigates the block stuffing attack).

Winner has to transfer debt in form of `CREDIT` token, replenish the `RateLimitedMinter` buffer with the principal repaid and burn the same amount of tokens. Lastly, `ProfitManager` is notified to split the interest repaid by the four recipients or take the loss by subtracting the buffer and even lowering the `creditMultiplier` in case there is no enough first-loss capital.

7. Lending term off-boarding

Off-boarding is the process of removing a potentially dangerous term. Everyone can propose term through the proposeOffBoard function in `LendingTermOffboarding` contract and then `GUILD` token holders can vote in supportOffBoard. This process is not time locked and once a certain quorum is met, offboard is called which add the gauge to `deprecatedGauges` in the `GuildToken` contract and pauses the `CREDIT` token redemptions for the reference asset. All the loans for off-boarded term are callable which will lead to cleanup. Off-boarded loan still can be activated by adding it again to the gauges by governor calling addGauge. After `issuance`, which are the total borrowed `CREDIT` tokens, gets to 0 as a result from auctioning all the loans, cleanup is called and redemptions are unpaused in the `SimplePSM`.

Architecture Recommendations

In overall architecture is pretty decent, given the fact the complexity of the code. There are a lot of moving parts as this increases the attack surface but this cannot be changed due to the end goal of the developers - to have many independent markets with core components in common (ex. `GuildToken` and `AuctionHouse`).

ERC20 contracts used - Gauges, MultiVotes and RebaseDistributor are already audited and the team only applied minor changes, which makes the auditing easier as it lowers the possible attack surface.

Governance part is also good, using many of the OZGovernor contracts and removing the unnecessary overhead that could occur if contracts were developed from scratch.

Important notes regarding the codebase:

Ensure there are no things left from the refactoring because now system will be able to handle different types of tokens, regarding decimals, fees, blocklists, reentrancy and many more. To mitigate it consider adding constants such as `MIN_STAKE` in `SimplePSM` to be assigned through the constructor.

Another serious flaw is the hardcoded `ProfitManager` in `GuildToken` which brings serious vulnerability in case there are two or more different markets, bad debt can be realized for the one which has the same `ProfitManager` contract as this in the `GuildToken`.

Last thing to mention is to verify that there is no way call stake in `SurplusGuildMinter` for term which is not from the market.

Codebase Quality Analysis

We consider the quality of the `Ethereum Credit Guild` complex but versatile with room for improvement.

Governance processes are well defined, using battle-tested OZ implementations.

New market term creation has its dangers and is gas costly, because of the amount of contracts needed to be deployed, but gives the term creators the freedom to have as many as possible configurations.

Approach taken for the lending and borrowing part of the system is good, it handles the bad debt realization different from the other protocols. Oracle-less approach has many benefits as it removes the unnecessary external dependencies. Staking in the system is highly incentivizes as there rewards not only in their native token `GUILD`, but also in the `CREDIT` token which can be redeemed for its reference asset.

Tests are too superficial and only cover the most obvious cases. Most of the problems remain undiscovered because there are no tests with 2 active markets and many terms in them. That way to properly test the behaviour of the contracts, responsible for holding data from many lending terms, such as `GuildToken`.

Codebase Quality Categories	Comments	Score (1-10)
Code Maintainability and Reliability	The codebase demonstrates moderate maintainability with well-structured functions and shared modularity across various contracts constructing all the calls in the system. Well-designed system for the initial purpose (having only one market), but lacks extendability for handling multiple markets mainly because GUILD token is not developed to work with many PROFIT_MANAGER contracts, as there will be many CREDIT tokens after refactoring. Another small flaw is variable names, some of them are confusing such as newBuffer in RateLimitedV2 contract.	5
Code Comments	All the contracts in scope have comments explaining the purpose and functionality of their functions, making it easier for developers and auditors to understand and maintain the code. Comments describe methods, variables, and the overall structure of the contract. Abstract contracts have tremendous amounts of comments that make the understanding an easy process.	9
Documentation	Mostly non-technical documentation is provided in ECG's GitBook and a decent overview on the contest page explaining the idea, invariants, attack scenarios, and the main actions that various users will be performing in the system. Later on, diagrams regarding the Dutch Auction and Governor mechanisms were provided as well as Excel tables explaining the math by showing exact numbers.	7
Code Structure and Formatting	The codebase, except for some abstract contracts is well-structured and formatted, utilizing OpenZeppelin's governance and Fei Tribe DAO's ERC20Gauges, which are battle-tested. That results in a lower attack surface and easier auditability and development mostly by people familiar with them.	8

Test analysis

The audit scope of the contracts to be audited is ~99% most of them only covering easy scenarios, without many function calls.

A good amount of testing should be applied to the full capabilities of the code, because now tests are emphasising on quantity rather than quality. The fact that there are integration and proposal tests is great, same amount of effort should be given to the **unit** tests as well.

Systemic & Centralization Risks

Actually, the `Ethereum Credit Guild` contains many roles, each with quite a few abilities. This is necessary for the Protocol's logic and purpose.

But this won't be problem by itself because team is planning to revoke the `GUARDIAN` and `GOVERNOR` roles after beta has passed as we can see from Code4rena contest page:

> After the beta period, governance powers will be burnt and no further arbitrary code changes possible. Instead, the system is build around explicitly defined processes such as the onboarding and offboarding of lending terms, or adjusting system parameters such as the surplus buffer fee.

Other roles explained in Privileged Roles does not expose any security risk, because they will be given to contracts and no certain people will be assigned to them.

Possible issue will be large governance token holders as they will have more voting power and can guide the protocol by their desire for example vote for their proposals and parameter changes.

Team Findings

Severity	Findings
High	4
Medium	7
Low/NC	8

Most of the findings were identified in the `SurplusGuildMinter`, `ProfitManager` and `GuildToken`.

The ones with a significant impact are:

- Stakers can avoid being slashed by frontrunning the transaction.
- User can decrease `GUILD` token stakers rewards with flash loan and claim 99% of the rewards for a single transaction.
- Wrong hardcoded `ProfitManager` in `GuildToken` will lead to impossibility of calling `notifyPnL` with negative amount for other types of gauges.
- Stepwise jumps in the `rewardRatio` will disturb the stakers rewards.
- Relying on stakers to update their `mintRatio` for themself isn't the optimal approach.

New insights and learning from this audit

Learned about:

- Governance
- Rebasing mechanism used in `ERC20RebaseDistributor`
- DutchAuction

Conclusion

In general, the Ethereum Credit Guild project exhibits an standard and well-developed architecture we believe the team has done a good job regarding the code, but the identified risks need to be addressed, and measures should be implemented to protect the protocol from potential malicious use cases. Additionally, it is recommended to improve the tests as there is a need of proper testing covering complex test scenarios. It is also highly recommended that the team continues to invest in security measures such as mitigation reviews, audits, and bug bounty programs to maintain the security and reliability of the project.