PUBLIC

Code Assessment

of the Angle Protocol Smart Contracts

October 08, 2021

Produced for

Angle

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Angle Team,

First and foremost we would like to thank you for giving us the opportunity to assess the current state of your Angle Protocol. This document outlines the findings, limitations, and methodology of our assessment.

The documentation and the code reviewed are of a high standard. Nevertheless, the protocol logic as well as the implementation are quite complex. While most issues have been resolved, some remain open as the risk has been accepted and acknowledged.

For a complete list of issues please refer to the Findings overview.

The communication with your team during the audit was very good and helped to resolve arising questions quickly.

We hope that this assessment provides more insight into the current implementation and provides valuable findings. We are happy to receive questions and feedback to improve our service and are highly committed to further support your project.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

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

Critical-Severity Findings	0
High-Severity Findings	4
• Code Corrected	4
Medium-Severity Findings	18
• Code Corrected	13
• Specification Changed	2
• Code Partially Corrected	2
• Risk Accepted	1
Low-Severity Findings	37
• Code Corrected	29
• Specification Changed	3

• Risk Accepted	4
• Acknowledged	1

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 Angle Protocol repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	July 26 2021	2819ccd8b62b09ce948da5088acba7b0b575d3f8	Initial Version
2	August 29 2021	079d3b44d803dfe4e490d7b0cfdb79b213ec5c53	Second Version
3	October 01 2021	46e6d32837cbe97a4af0adb693e63afa0d01fc3e	Third Version

For the solidity smart contracts, the compiler version `0.8.2` was chosen. For the updated versions, compiler version `0.8.7` has been selected.

The following smart contracts directories were in scope:

- agToken
- core
- poolManager
- feeManager
- oracle
- perpetualManager
- sanToken
- stableMaster
- interfaces
- The `RewardsDistributor` contract in staking
- strategies
- utils

The imported OpenZeppelin contracts were not reviewed.

2.1.1 Excluded from scope

The contracts in following directories are out of scope:

- dao
- external
- genericLender
- staking (except the `RewardsDistributor` contract)
- mock
- collateralSettler
- bondingCurve

The economic model behind the protocol and the chosen economic paramters of the system are out of scope. Under certain economic conditions the system will fail.

3 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview. At the end of this report section we have added subsections for each of the changes accordingly to the versions. Furthermore, in the findings section we have added a version icon to each of the findings to increase the readability of the report.

The Angle Protocol offers investment opportunities for different kinds of actors. The protocol brings together:

- Stable Seekers: Actors looking to exchange a certain collateral token into a stable asset and back at the current market rate
- Hedging Agents: Actors looking for a perpetual future in order to increase the leverage on their collateral
- Standard Liquidity Providers: Actors looking to increase the interest earned by their collateral

The Angle Protocol issues three kinds of tokens:

- Stablecoins such as e.g. AgEur
- SanTokens for Standard Liquidity Providers representing their contribution
- Perpetuals which are technically NFTs

For each Stablecoin (e.g. AgEUR) a `StableMaster` is deployed. For each `StableMaster` the supported collaterals can be added individually. For each collateral of a `StableMaster`, a SanToken and a PerpetualManager are deployed.

Such a market issuing stablecoins must be collateralized at all times. Hedging Agents cover the collateral brought by stable seekers against price decrease. As no perfect match will exist at any time given, standard liquidity providers add additional liquidity in form of collateral while being able to earn interest accrued by the whole amount of this collateral held by this stablemarket. Variable fees play a vital role in the system. Overall the demand between the participants should be balanced for the system to work properly. To achieve this, the fees depend on the current state of the system. Actions balancing the system are cheaper compared to actions bringing the system into an even more unbalanced state where the fees increase accordingly. Many parameters exist for the governance to fine tune the fees system. For the proper working of the system the correct choice of these incentives is vital.

The system is governed by a DAO. Most contracts are upgradable through a proxy pattern.

3.1 Contracts

The system consists of following interacting contracts:

3.1.1 Core

The Core contract acts as a registry for all stablecoins deployed by the protocol. Thus, it keeps track of all `StableMaster` contracts. Moreover, Core is the entry point for governance changes as it allows to modify the guardian and add/remove governors. These changes are propagated to each `StableMaster` contract and is echoed through to the `AgToken`, `PoolManager`, `FeeManager`, `PerpetualManager`, `Strategy` and `Lender` contracts. The Core contract not only keeps track of deployed stablecoins but ensures that the `StableMaster` contracts are correctly deployed and interaction with them is enabled.

3.1.2 StableMaster

The Core is an administrative contract over all stablecoins. Similarly, the `StableMaster` is the administrative contract per stablecoin ensuring that governance changes coming from Core are propagated to all contracts deployed for the stablecoin. The `StableMaster` keeps track of which pool managers are whitelisted and, thus, keeps track of which collateral is supported for a stablecoin. Through `StableMaster`, governance can white- and blacklist collateral. It makes sure that the pool manager is correctly initialized once it becomes whitelisted. Guardians also can set the oracle, the fee manager, fees for SLPs and stable seekers, and pause/unpause some `StableMaster` functionality and governance can change the Core contract that is referenced by the `StableMaster`. If a collateral is revoked, the funds of that collateral are transferred to a settlement contract which will ultimately distribute the funds to the respective parties.

`StableMaster` is the entry-point for stable seekers/holders and allows exchanging whitelisted collateral for the stablecoin (and vice-versa) at the current exchange rate read from some oracle. However, the collateral bought back with the stablecoin does not have to be the collateral that was used to buy the stablecoin. Thus, this contract may also be used as some sort of exchange. The stablecoin of the `StableMaster` is represented by the AgToken contract which is an ERC-20 token with 18 decimals implementing the additional `permit()` functionality specified in EIP-2612. There is a one-to-one mapping from `StableMaster` to AgToken. The corresponding `StableMaster` is the only address allowed to mint AgTokens. However, governance can set a cap on the total amount of stablecoins.

`StableMaster` also allows standard liquidity providers to deposit collateral to a supported pool in exchange for pool-specific interest-accumulating SanTokens. These can be redeemed in exchange for the collateral by the standard liquidity providers. SanTokens are similar to AgTokens. However, SanTokens are pool-specific and, hence, there is one-to-many mapping from `StableMaster` to SanTokens. Also SanTokens do not have any parameters that can be set by the governance or guardians and have the same amount of decimals as their underlying token.

To conclude, one `StableMaster` is deployed per stablecoin. It is the entry-point for stable holders/seekers and for standard liquidity providers. Moreover, it allows adding or removing a collateral to this stablemarket and setting parameters that are individual per stablecoin but must be the same over all underlying contracts.

3.1.3 PoolManager

A separate pool manager contract is used to manage each individual collateral of a stablemarket. It holds the respective collateral tokens. When a new collateral is added to a stablemarket, the pool manager contract is registered and can then be used in this stablemarket. For one stablemarket, a collateral token must only have one pool manager. It receives the collateral used to buy stablecoins, deposited by SLPs and the HAs backing capital. To manage the funds the strategy role is allowed to allocate funds to strategies. This is limited by an overall and per strategy limit to ensure some liquidity always remains available e.g. for users burning stablecoins for this collateral or standard liquidity provider redeeming their SanTokens. As a central point for asset management per collateral, the pool manager forwards the reports of the strategies on gains and losses to the global accounting contract `StableMaster`.

3.1.4 PerpetualManager

For the third external party, the hedging agents, the perpetual manager is the entry point for opening and closing perpetual positions. For each pool manager, there is exactly one perpetual manager.

Hedging agents can through this contract:

- create a perpetual

  A hedging agent brings in a gross amount of collateral to the protocol from which fees are deducted and a net amount is computed. The hedging agent also specifies a committed amount which corresponds to the amount covered by his position. The committed amount in comparison to the net brought amount, however, cannot exceed the maximum leverage.

Further, a position can only be created if the covered amount after the transaction does not exceed the maximum amount that is allowed to be covered.

- cashout a perpetual

  A hedging agent can cashout his position with this function. In case of collateral price increase, the amount that can be cashed out is the capital brought plus the capital gains. In the case of collateral price decrease, the amount that can be cashed out is the minimum of the capital brough minus the capital losses and zero. Ultimately, the perpetual owner receives that amount minus withdraw fees (may be zero). The perpetual can only be cashed out after some number of blocks (set by governance) has passed.

- add to a perpetual

  This functionality allows a perpetual owner to modify his position by changing the amount he brought to the protocol. This means the perpetual owner can bring in more capital that is going to be added to his perpetual. As the committed amount remains unchanged, this decreases the leverage of the perpetual.

  As the price between when the perpetual was created and now has changed, the previous brought amount stored in cashOut amount is no longer current. The previously brought amount needs to be recalculated into its current value before the new addition is added on top. This operation is treated as creating a new position since the initial rate is set to the current one and the creation time is set to the current block time. Stocks users in `StableMaster` is notified about the change. However, the NFT token for the perpetual stays the same.

  Adding is only possible if the current cashout amount is larger than zero, otherwise the position will be liquidated automatically.

- remove from a perpetual

  Hedging agents can also remove some of the capital they brought in. It works similarly as the adding functionality. However, it can only be executed if the secure number of blocks has passed (as in cashout) and the maximum leverage is not exceeded after the removal. The amount that is received is the amount specified minus the fees.

The protocol and the perpetual manager heavily rely on keepers. They can perform the following actions:

- force cashout perpetuals

  Forcing a cashout for a perpetual is similar to the cashout that the owner can perform. However, the force cashout can only occur if the cashout amount is zero, the cashout amount is below the maintenance margin, the covered amount by perpetuals is larger than the maximum coverable amount or if the leverage has become too high. The amount that is returned to the owner is computed as the amount that the regular cashout would bring. However, the net amount will be less since the keeper will receive some fee for calling force cashout.

- liquidate perpetuals

  Liquidation works similar to the force cashout. However, the amount received by the user will be always zero. However, it can only be called if the maintenance margin has been reached or the cashout amount is zero. It also transfers a fee to the keeper.

Liquidating or cashing out too late may harm the protocol as it decreases the stocks users amount badly. Since the pool manager will have funds in strategies, it could be possible that there is not enough collateral to cashout. In such cases, new SanTokens are minted at the current exchange rate and given to the owner instead.

The perpetual are ERC-721 (NFT) and implement their functionality. Moreover, hedging agents will receive rewards for backing the protocol with collateral. The perpetual mangers will, therefore, be registered in the RewardsDistributor contract as staking contracts.

3.1.5 FeeManager

One FeeManager contract is deployed per PoolManager. It allows the Guardian role to set all fee related parameters. Using the update functions, this change can be propagated to the `StableMaster` or the PerpetualManager by anyone. No incentives are paid for this task. While the new fee data for the hedging agents are propagated as they are, the new values for the `StableMaster` are calculated on update based on the current system state. It is assumed that the update for the fees is triggered sufficiently often.

3.1.6 CollateralSettler

Should a collateral have to be revoked for any reason, the governance can do so by executing `revokeCollateral()` on the respective `StableMaster` contract. This will transfer all available collateral of the PoolManager to the passed settlement contract address.

During a claim period, all eligible parties (Ag token holders, San Token Holders, Perpetual Holders) can register their claims. After the claim period terminated, the available collateral is distributed over the claims. Users are now able to retrieve their collateral claimed.

3.1.7 Oracles

Although users do not directly interact with the oracles of the system, they play a vital role for the system. Any asset for which an oracle exist can be used as underlying stable asset and a stable market can be created for it. In the Angle Protocol, two types of oracles are used:

- OracleMulti

  This oracle contract is used for assets for which a Chainlink V3 and a Uniswap V3 oracle exists. The oracle queries both rates and the Angle Protocol uses the rate more favorable to the system depending on the respective action.

  It is known and accepted that the Uniswap rate may be manipulated.

- OracleChainlinkSingle

  In order to support stablemarkets for which no Uniswapv3 oracle exists, this oracle exists. It features the same interface as OracleMulti, however the data returned is based on the Chainlink oracle only.

3.2 Trust Model & Roles

Users of the System:

Users of the system are generally untrusted and expected to behave unpredictably. The user roles are:

StableSeekers: Anyone exchanging collateral into Ag stabletokens or vice versa. Untrusted Role.

HedgingAgents: Anyone may become a hedging agent and insure the system against the price decrease on a certain amount of a certain collateral. Untrusted Role.

Standard Liquidity Provider: Provides additional liquidity in return for interests. Untrusted Role.

Keepers: Executes actions in the system. Some actions (liquidating or closing perpetuals) offer a reward while others (updating fees) do not. Untrusted Role, assumed to be incentivized by the financial reward and acting accordingly to maximize his profits.

Furthermore there are following roles:

Governance: Expected to be a DAO. It is expected to behave honestly and correctly, but some safeguards should be in place in the Angle Protocol to prevent errors.

Guardian: Expected to behave honestly and correctly at all times. Fully trusted role completing the Governance for time critical actions where the governance (a DAO) may be too slow such as pausing the system should a problem arise or similar actions.

Oracles: The oracles of the system rely on underlying oracles as source for the price data:

- Chainlink. Fully Trusted.
- UniswapV3. Expected to be manipulatable. Hence the internal oracle of the system only uses this source together with Chainlink and takes the rate more favorable for the system.

3.3 Version 2

In this section we summarize the most significant changes in Version 2. Smaller changes only introduced as part of a fix are not mentioned.

- The PerpetualManager has been reworked:
  - Values `entryRate` and `entryTimestamp` no longer change when a perpetual is modified
  - `maxALock` has been replaced by `targetHACoverage` and `limitHACoverage`
  - Fees for a perpetual are now based on the committed rather than the amount brought
- The `stocksUser` variable of a collateral no longer tracks the collateral, but the amount of stablecoins minted/burned for this collateral

4 Limitations and use of report

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

The focus of our assessment was limited to the code parts associated with the items defined in the engagement letter on whether it is used in accordance with its specifications by the user meeting the criteria predefined in the business specification. 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 the facts that over the life cycle of any software product changes to the product itself or to its environment, in which it is operated, can have an impact leading to operational behaviours other than initially determined in the business specification.

5 Terminology

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

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

We categorize the findings into four distinct categories, depending on their severities. 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.

6 Findings

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

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

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

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	3

- Cached Rate May Be Wrong Risk Accepted
- Missing Slippage Protection for Users Code Partially Corrected Acknowledged
- Problematic Revocation of a Collateral Code Partially Corrected

Low-Severity Findings	5

- Inputs for Triggering Settlement Risk Accepted
- Removing From Perpetual Potentially Impossible Acknowledged
- Sandwich Attacks Against harvest Invocations Risk Accepted
- HA Fees May Exceed 100% Risk Accepted
- Small Perpetual Position, Too Low Keeper Incentive Risk Accepted

6.1 Cached Rate May Be Wrong

Correctness Medium Version 1 Risk Accepted

OracleMulti allows to read rates from Chainlink and Uniswap circuits with `_readAll()`. First the Uniswap rate is computed. However, the Uniswap circuit may not be final, meaning that the last pair (e.g. WETH to USD) requires a Chainlink rate. Next the Chainlink rate is read from the circuit. The rate of the last Chainlink circuit pair is cached, to be used for further computations on the Uniswap rate.

However, the constructor allows for the last Chainlink and Uniswap pairs to be different. Thus, the following scenario is possible:

1. OracleMulti is initialized with a Chainlink circuit (UNI-WBTC, WBTC-USD) and a Uniswap circuit (UNI-WETH). That means that the Uniswap is not final and a Chainlink rate has to be read for the rate WETH-USD.
2. The Chainlink rate is calculated and the WBTC-USD rate is cached.
3. The Uniswap UNI-WETH rate is computed. Inside the branch `if (uniFinalCurrency > 0)` the calculation of the rate is finalized using the cached WBTC-USD rate which leads to an incorrect result, as the rate for WETH-USD should have been used instead.

Risk accepted:

Angle will make sure that the Uniswap and Chainlink circuits are compatible. A comment has been made in the code.

6.2 Missing Slippage Protection for Users

Security | Medium | Version 1 | Code Partially Corrected | Acknowledged

There is no slippage protection for users interacting with the `mint()`, `burn()`, `deposit()` or `withdraw()` functions of a StableMaster contract nor for actors interacting with certain functions of a PerpetualManager contract which calculate the cash out amount of a perpetual.

A user can get caught unlucky, especially as fees depend on the current state of the system or as the cash out amount of a perpetual depends on the current rate returned by the Oracle. There is a risk of sandwich attacks on user's transaction: A user's transaction may be sandwiched between two of the attacker's transaction. The first transaction of the attacker may change the state of a system resulting in an unfavorable outcome of the user's transaction while the attacker profits with his second transaction just after the user's transaction.

Code partially corrected:

A slippage protection has been added for stable seekers minting and burning. Slippage protection has been introduced for hedging agents.

Acknowledged:

However, it was concluded that not further slippage protection for standard liquidity providers is necessary.

6.3 Problematic Revocation of a Collateral

Design | Medium | Version 1 | Code Partially Corrected

StableMaster allows for `revokeCollateral()` to be called by the governance. That transfers all the funds of the pool manager to a settlement contract. Afterwards users can make claims for withdrawing from the settlement contract. However, some user may lose.

This issue attempts to highlight two key points:

(I).

While this function is part of the emergency shutdown process of a stablecoin, this function can also be called on a single collateral only. In both situation following scenario (simplified that no HAs exist) which is mentioned in the documentation may occur:

1. `revokeCollateral()` gets called on a pool with 1000 WETH. 1 WETH is worth 1000 USD. 100 WETH-SanTokens are minted and have value of 200 WETH. The pool manager transfers his balance to the settlement contract.
2. `CollateralSettler.triggerSettlement()` is executed. The amount to redistribute is the balance of the settlement contract. All the rates are frozen.
3. SLPs claim their collateral. `totalLpClaims` increases.
4. A day before the claiming period ends, the price of WETH doubles. 1 WETH is worth 2000 USD now in the current markets.
5. Many users see the bargain and start claiming WETH for their AgUSD.

6. The claiming period ends. The claim of stable holders is 1000 WETH. The claim by SLPs is 200 WETH.
7. The WETH will be distributed only to stable holders. SLPs do not receive anything.

The documentation specifies this behaviour. However, it highly concerning for SLPs and HAs. Anybody with enough capital could take their investments into the protocol.

(II).

Furthermore, SLPs and HAs could lose funds if an attacker decides to frontrun the `revokeCollateral()` call. The pools balance could be moved to another pool using a flashloan and an external exchange. Then, the `revokeCollateral()` call will be executed but close to nothing would be transferred to the settlement contract and SLPs and HAs will not be able to receive their funds.

Code partially corrected:

Different changes have been made:

- As `stocksUser` now tracks the amount of created stable coins it can also be used to limit the claims.
- The oracle value is queried at the end of the claim period to reduce issues due to price fluctuation.

Lastly, the front-running issue will be partially mitigated through pausing, but as in comparable systems cannot be entirely avoided.

6.4 Inputs for Triggering Settlement

Correctness Low Version 2 Risk Accepted

When the `StableMaster` contract triggers a settlement on the `CollateralSettler` contract it passes the current `sanRate` along. During this process any queued `lockedInterests` that were supposed to be added to the `sanRate` later are ignored. Hence, the `sanRate` is not entirely correct.

Risk Accepted:

Angle replied:

```
We decided to leave it as is. lockedInterests supposed to be added to the sanRate remain ignored.
It could be a vector of attack to include these interests to SLPs. If trigger settlement was to be activated,
then this means that governance failed to maintain the pool in a healthy way, and in this situation,
interests should not be distributed to SLPs (we expect that there will also be fees aside)
```

6.5 Removing From Perpetual Potentially Impossible

Design Low Version 2 Acknowledged

When a perpetual position develops well and its margin has significantly increased, users might decide to remove some of the margin through the `removeFromCollateral` function. However, if the amount of collateral to be removed exceeds the initially set margin, this removal is not allowed, even though it might not violate any of the system restrictions such as maximum leverage:

```
require(
        ...
        (amount < perpetual.margin) &&
```

Acknowledged:

Angle replied:

```
In fact the margin of a perpetual never increases if the perpetual develops well:
the margin is the initial amount of collateral in the perpetual, and this does not evolve with price.
If we allowed HAs to remove more than their margin in case of price increase,
we would be back to the situation we had before your audit where we also update the oracle value,
and what we called the cashOutAmount at each perpetual update. For the HA to get more collateral than the margin, position should be cashed out
```

6.6 Sandwich Attacks Against `harvest` Invocations

Security | Low | Version 2 | Risk Accepted

The `harvest` functions of certain strategies, e.g., `GenericAave`, are susceptible to sandwich attacks. As these `harvest` functions can be called by anyone and perform a token sale, the following attack is possible:

- An attacker contract manipulates the relevant Uniswap pools
- That attacker contract calls the `harvest` function of one of the strategies, which triggers a Uniswap trade
- The attacker contract arbitrages the Uniswap pools to benefit from the previous trade

In the currently present strategies, such attacks are limited to the reward tokens.

Risk accepted:

Angle replied:

```
We forked these strategies from Yearn, we have hence decided to keep it as is,
and we are aware of this risk. It is important to note that to mitigate such attacks,
however costly it is, the harvest function needs to be called pretty regularly.
```

6.7 HA Fees May Exceed 100%

Design | Low | Version 1 | Risk Accepted

There are no limitations for the values of `haBonusMalusDeposit` or `haBonusMalusWithdraw`. These can be set arbitrarily by the Guardian / Governor in the FeeManager and are later propagated to the PerpetualManager.

Inside the PerpetualManager, the fee for HAWithdraw is calculated as follows:

```
haFeesWithdraw = (haFeesWithdraw * haBonusMalusWithdraw) / BASE;
return (amount * (BASE - haFeesWithdraw)) / BASE;
```

In case `haFeesWithdraw` exceeds BASE the transaction will revert and the withdrawal is blocked. The same applies for HADeposit accordingly.

Risk accepted:

Angle replied:

```
No specific change has been made for that, if this situation happens, then the transaction
will fail anyway and there is no need to add a require for that.

We thought of adding a require in the setters of the fees to make sure that fees will never
be able to be bigger than 100% (especially for users minting/burning fees), but we decided
not to do it. The reason is that our fees are of the form f(x)g(y), with 0 <= f(x) <= 1.

Therefore it may happen that for some value of the y parameters, you have g(y) > 1, and for
some couples (x,y), you have f(x)g(y)>1. We do not want to enforce that the product is always <1.

In our case, the evolution of the bonusMalus (depending on the collateral ratio for users)
and the evolution of the fees computed using the coverage curve are different. It is
possible that the product in the max element in the array yBonusMalusMint and in the array
yFeeMint are superior to BASE but that this situation is never observed in practice
because the evolution of the collateral ratio is not correlated to the evolution of the coverage curve.

Governance will still have to be wary and to make sure when setting these parameters that
even though a situation where  f(x) g(y)>1 can happen in theory, it will never happen in practice.
```

6.8 Small Perpetual Position, Too Low Keeper Incentive

Security Low Version 1 Risk Accepted

The system enforces no minimum amount for a perpetual position and corresponding minimum fee paid. The reward / incentive for a keeper to liquidate a perpetual that meets the condition to be liquidated respectively to be forcefully cashed out is a part of fees paid by this perpetual.

The incentive for keepers must at least cover their transaction costs. As no minimum amount for a perpetual (hence keeper fee) is enforced, perpetuals bringing a small amount of collateral to the system and hence paying a small fee may not be liquidated as the reward exceeds the keeper's transaction fees.

Risk accepted:

Angle replied:

```
Although we slightly changed the keeper incentives (as a portion of the cashOutAmount at
the time the perpetual is cashed out), we decided not to have a minimum position or a
minimum incentive for keeper. If the incentives are too low, we will do it ourselves, even
if it implies loosing money on it. Another thing we think about implementing is an
off-chain reward mechanism based on on-chain verifiable data. This way we/our community
could reward keepers which performed actions for which they did not make a profit but that
were still helpful for the protocol. We could also upgrade our smart contracts to arrive
to the solution you propose (minimum incentive for keeper coupled with a minimum position -
you cannot do one without the other otherwise you may be subject to attacks).
```

7 Resolved Findings

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

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

Critical-Severity Findings	0
High-Severity Findings	4

- Burning AgTokens in BondingCurve Does Not Update stocksUsers Code Corrected
- Collecting Keeper Fees, Closing Perpetuals Code Corrected
- Incorrect Maximum Collateral Amount Code Corrected
- Untracked Bad Debt / System Health Code Corrected

Medium-Severity Findings	15

- Everybody Can Pause Pools Code Corrected
- Incorrect Check During removeFromPerpetual Code Corrected
- Incorrect Handling of profitFactor Code Corrected
- Potentially Incorrect Strategy Report Code Corrected
- Reentrancy When Creating Perpetual Position Code Corrected
- Adding to Unhealthy Perpetual Liquidates and Consumes New Collateral Code Corrected
- Broken/Partial ERC165 Support Code Corrected
- Conversion of Locked Interest to SanRate in the Same Block* Code Corrected
- Governance Not Fully Propagated Specification Changed
- Guardian Cannot Be Managed by Guardian Code Corrected
- Incorrect Cash Out Amount Code Corrected
- Non 18 Decimals Protocol Tokens Specification Changed
- Unaccounted Collateral, Unrestricted updateStocksUsersGov() Code Corrected
- Unit Errors for Tokens With Decimals Different Than 18 Code Corrected
- safeApprove Not Used, USDT Not Supported Code Corrected

Low-Severity Findings	32

- BondingCurve Specification Mismatches Code Corrected
- Gas-inefficient Strategies Code Corrected
- Inefficiency in Binary Search Code Corrected
- No Slippage Protection in BondingCurve Code Corrected
- Reference Coin Changes May Affect the Bonding Curve Code Corrected
- Specification Mismatch in Strategy Specification Changed
- StableMaster Might Be Unnecessarily Paused Code Corrected
- Cache Value Instead of Reading From Storage Code Corrected
- Consistency Checks for Oracles Missing Code Corrected

- Different Calculation of Cashout Fees Code Corrected
- Double Getters Code Corrected
- Enhance Check During Deployment of Collateral Code Corrected
- Events Missing Code Corrected
- Gas Inefficiencies When Removing From List Code Corrected
- Gas Inefficiencies When Searching Lists Code Corrected
- Governance Changes Code Corrected
- Inconsistencies Between Staking Contract Code Corrected
- Inconsistent Parameters in RewardsDistributor Possible Code Corrected
- Incorrect Comment Specification Changed
- Inefficient Structs Code Corrected
- No Check if onERC721Received Is Implemented Code Corrected
- No Checks Performed in Constructor Code Corrected
- Outdated Compiler Version Code Corrected
- Overhead Due to Loading Struct Into Memory Code Corrected
- Possibly Failing Assert Code Corrected
- Potential Confusing readLower(uint256 lower) Code Corrected
- Reward Token Issues Code Corrected
- Specification Mismatch in OracleMath Specification Changed
- Unnecessary Double Checks Code Corrected
- Updated SanRate When Converting to SLP Code Corrected
- Wrong Incentive for Which Perpetuals to Forcefully Cash Out Code Corrected Acknowledged
- capOnStablecoin May Be Violated by Guardians Code Corrected

7.1 Burning AgTokens in BondingCurve Does Not Update `stocksUsers`

Correctness High Version 2 Code Corrected

The BondingCurve contract allows users to buy tokens (most likely Angle tokens). To receive tokens from the BondingCurve, AgTokens get burned according to the bonding curve price. This may significantly reduce the amount of AgTokens minted and improve the protocol health. However, no `stocksUsers` variable is updated. Several issues may arise regarding such accounting issues. For example, the coverage ratio of the protocol may be much higher than the one indicated and the system create bad debt through that mismatch.

Note that regular `burn` operations also do not update the `stocksUsers` variable.

Code corrected:

When buying tokens the AgTokens are transferred to the bonding curve contract and not burned. Governance can make use of functions `BondingCurve.recoverERC20` and

`AgToken.burnNoRedeem` which calls `StableMaster.updateStocksUsers` to transfer AgTokens to itself and burn them while updating the stocks users for a specified pool manager.

7.2 Collecting Keeper Fees, Closing Perpetuals

Security High Version 1 Code Corrected

Should the maximum covered amount of collateral be exceeded, anyone can forcefully cash out any perpetual in order to bring the amount of covered collateral back below the limit. However this can be abused as one can manipulate the amount of collateral to be covered.

Assume a working StableMaster issuing `AgUSD` with several collateral pools like `USDC`, `DAI` and `WETH`. There is a decent amount of liquidity provided by standard liquidity providers and the coverage ratio of the pools is around 80% with a limit at 90%. Many perpetuals of different sizes exist. An arbitrary attacker can now do the following:

1. Either the Attacker has funds available or borrows them using a flashloan
2. These funds are exchanged into `AgUSD` on a third party exchange
3. These AgUSD are now burned for the collateral under attack.
4. Burning the AgUSD tokens increases the collateralization ratio for this collateral as collateral is withdrawn. The attacker does this at least until the coverage limit is exceeded.
5. The attacker is now able to forcefully cash out perpetuals until the amount covered is below the limit. While forcefully cashing out perpetuals the attacker collects the fees.
6. Pay back the flashloan using the collateral.

This attack is profitable when the transaction, flashloan and burn fees are below the keeper reward collected for closed perpetuals. As keeper fees for each perpetual have to cover for the transaction base fees (as they may have to be closed individually by keepers due to reaching the cashout leverage) the collected rewards likely exceed the fees when the attacker manages to forcefully cash out multiple perpetuals during this action.

Code corrected:

The new fee structure rewards keepers reaching the targeted coverage ratio. Moreover, the keeper reward is capped such that the profit of the keeper is lower than the estimated cost of the flash loan needed for such an attack. For more information see the description of System Accounting.

7.3 Incorrect Maximum Collateral Amount

Correctness High Version 1 Code Corrected

The function `_testMaxCAmount` computes the "Maximum amount of collateral that can be insured". This is computed as follows:

1. The `stocksUsers` variable is queried from the `StableMaster` contract.
2. The amount of minted stable coins is queried from the `StableMaster` contract and converted into a collateral amount using the current rate.
3. The smaller of the two values above is multiplied with `maxALock` (the maximum percentage to be insured) and then returned.

Both of these values are sometimes incorrect and hence shouldn't be used for the calculation:

1. `stocksUsers` is defined as:

```
// Amount of collateral in the reserves that comes from users
// + capital losses from HAs - capital gains of HAs
```

Due to the capital losses and capital gains it might be bigger or smaller than needed for the present calculation. Consider the following example:

After a late liquidation `stocksUsers` = 15 ETH, with a rate of 500, but previously 10,000 stable coins have been minted. The system needs to insure 20 ETH, but would return 15 ETH * `maxALock`.

2. The amount of minted stable coins can only be used for this calculation if only a single collateral is used for this stable coin. However, multiple collaterals might be available to mint this stable coin and hence the system would calculate an incorrect amount of insurable collateral.

Code corrected:

Now, the `stocksUsers` variable represents the amount of stablecoins minted per collateral and the system separates the stable coins minted against different collaterals. For more information see the description of System Accounting.

7.4 Untracked Bad Debt / System Health

Design | High | Version 1 | Code Corrected

A stablemarket attempts to keep the equilibrium between the positions of stable seekers and hedging agents. At times the attempt to keep the equilibrium may not be successful:

- Should they price of a collateral decrease too fast and keepers can't or don't forcefully cash out bad perpetuals in time, the situation arises where the collateral put up by the perpetual evaluated at the current rate can no longer cover its committed amount at the rate the perpetual has been created. In this case the perpetual is liquidated leaving bad debt to the system.

Should the collateral of the stablecoin not be fully covered at all times (This means the collateral brought by stable seekers equals the amount covered by the hedging agents), 2 kinds of bad debt can occur:

- Coverage of the collateral is less than 100% and the price of the collateral decreases from x to y:

For the uncovered collateral stablecoins have been minted a the higher collateral price x. Now the value of the collateral dropped to y. The minted stablecoins are now only partially covered by the value of the collateral. Note that should a new Hedging Agent now enter the system and covers some more of the collateral, this is done at the current exchange rate, not the rate used to mint the stablecoin. At this point the virtual loss of the system is converted into actual bad debt of the system.

Vice versa, should the price of uncovered collateral increase the system makes a profit.

- Coverage of the collateral is more than 100% and the price of a collateral increases.

(Note that this cannot happen if `maxALock` is set to less than 100%)

Here profits made by Hedging Agents would exceed the increase in value of the collateral held by the system to back the minted stablecoins. This loss is taken by the system.

Overall bad debt is neither tracked nor handled otherwise. If possible it could be accounted for and compared with what is currently called "system surplus" which includes the fees collected and other gains made by the system.

No functionality to query the health of the system exist. Such information however is vital for all users investing funds into the system.

Code corrected:

The new `stocksUsers` enables to keep better track of the current system health and the bad debt. However, these computations need to be performed off-chain, e.g., in the front-end. For more information see the description of System Accounting.

7.5 Everybody Can Pause Pools

Design Medium Version 2 Code Corrected

In the second version of the code, a pool will be paused, if during a user's burn, the amount of AgTokens to be burned is higher than the stocks users of the collateral. However, there is no check whether the user actually owns the necessary amount of AgTokens. Thus, any user can specify a high amount to burn to pause the contract. The pool will remain paused until the governance unpauses this change. Malicious parties could act as follows:

- Stableholders: In case of expected collateral price drop can pause to make HAs and SLPs lose.
- SLPs: A SLP providing much liquidity in a state with much HA capital could pause the contract to keep other SLPs from entering the protocol so that his profit is maximized.
- HAs: HAs can front-run liquidations and force-cashouts by pausing the contract. Ultimately, that could lead to a highly unbalanced state.

In conclusion, anybody can pause the protocol at any time. Such actions could be profitable for the parties and could throw the system into an unhealthy state if they are executed repeatedly.

Code corrected:

When the amount of AgTokens burned exceeds the `stocksUsers`, the transaction reverts instead of pausing the contracts.

7.6 Incorrect Check During `removeFromPerpetual`

Design Medium Version 2 Code Corrected

The function `removeFromPerpetual` contains the following check:

```
// Withdrawing collateral should not make the leverage of the perpetual too important
perpetual.committedAmount * BASE_PARAMS <= (perpetual.margin - amount) * maxLeverage,
```

The `maxLeverage` check is performed based on `perpetual.margin - amount`, however, in case the `cashOutAmount < perpetual.margin` then this check underestimates the actual leverage. Hence, a perpetual above the leverage limit might go undetected.

Code corrected:

Now, an additional check was added to ensure that the new cashout amount is not exceeded.

7.7 Incorrect Handling of `profitFactor`

Correctness | Medium | Version 2 | Code Corrected

The profit factor serves to reduce the profit keepers can make from calling `harvest`. Thus, following condition occurs:

```
profitFactor * rewardAmount < want.balanceOf(address(this)) + profit
```

This condition needs to be fulfilled for a reward payment to be made and is hence quite important.

Incorrectly, the condition is unaware of the decimals of the tokens since `profitFactor` is initialized to be 100 for any pair. Moreover, it is unaware of the prices of the tokens.

The decimal unawareness may cause the following behaviour:

- Assume the reward token is USDC (6 decimals) and the want token is DAI (18 decimals). The condition will almost always pass since profit factor does not account for the base differences between the tokens.
- Assume the reward token is DAI (18 decimals) and the want token is USDC (6 decimals). Then, this condition will almost *never* pass to since the reward amount will already be much larger than the right-hand-side.

The price unawareness may cause the following behaviour:

- Assume the reward token is AgEUR and one strategy's want token is DAI while for the second one the want token is WETH. If now both strategies have similar balances and profits (when converted to USD), they will still be treated very differently.

To conclude, inconsistencies in the keeper reward payouts between strategies could occur since the above condition is unaware of the decimal representation and prices of the tokens.

Code corrected:

`profitFactor` has been removed. Now, a minimum amount `minimumAmountMoved` denotes how much needs to be at least in the contract plus the profits. Also, this amount and the reward amount are set jointly now to prevent errors.

7.8 Potentially Incorrect Strategy Report

Correctness | Medium | Version 2 | Code Corrected

Calling the `harvest` function on a strategy contract may result in a bad report to the pool manager. If the `reward` token the keepers receive is equal to the `want` token of the strategy, then the transfer to the keeper can be successful even though no specific allocation of funds to the strategy for the rewards was made. Incorrectly, the keeper's fee will still be part of the reported profit as the profit is computed beforehand and not adjusted.

Code corrected:

In the constructor of the strategy it checked that `want` and `reward` token are not the same.

7.9 Reentrancy When Creating Perpetual Position

Security | Medium | Version 2 | Code Corrected

When creating a new perpetual position there is a possibility for a reentrancy attack. During the `mint()` operation of the token a callback is triggered that can be used for a reentrancy attack.

Among other things, possible consequences of such an attack could be:

- that the coverage exceeds the expected values
- that a non-liquidatable perpetual exists
- that a mismatch between NFTs and positions exists

Code corrected:

The call to `_mint()` is done after all state changes.

7.10 Adding to Unhealthy Perpetual Liquidates and Consumes New Collateral

Design | Medium | Version 1 | Code Corrected

`addToPerpetual()` allows a hedging agent to increase the cash out amount of the perpetual. Should a hedging agent attempt to add collateral to an unhealthy perpetual, the perpetual is liquidated while the collateral amount intended to increase the position is transferred to the pool manager contract without being accounted for.

Unaware users are at risk, especially as a hedging agent may attempt to increase the collateral of a position which is just short of being liquidated. Any oracle update now may change the situation and the perpetual can be liquidated while the hedging agent loses his added collateral to the pool manager.

Code corrected:

Attempting to add collateral to an unhealthy perpetual results in the perpetual being liquidated, which is intended. In this case, the new collateral amount however is no longer transferred to the pool manager in the updated code.

7.11 Broken/Partial ERC165 Support

Design | Medium | Version 1 | Code Corrected

Through inheritance, mostly when inheriting `AccessControl` or `AccessControlUpgradable` multiple contracts inherit `ERC165`.

This contract implements the ERC165 standard which defines a standard method to publish and detect what interfaces a smart contract implements.

```
function supportsInterface(bytes4 interfaceID) external view returns (bool);
```

The more derived contracts of Angle with the exception of the PerpetualManager contract, that does it partially, do not expand or overwrite this function. Hence, their functionality is not included and `supportedInterface()` will not return `true` for the public/external functions they implement.

Either the more derived contracts should implement this function in order to make complete the ERC165 functionality or if this is not required, overwrite `supportsInterface()` with an empty function which would even slightly reduce the contract's code size.

Code corrected:

Angle forked the code of `AccessControl` and `AccessControlUpgradaeable` and removed the ERC165 support. The only contract which implements the ERC165 interface is the `PerpetualManager` as it emits the perpetual futures as ERC721-NFTs. The `supportsInterface` function will return `true` for all interfaces it implements.

7.12 Conversion of Locked Interest to SanRate in the Same Block*

Security Medium Version 1 Code Corrected

**While the review was ongoing Angle informed us about this issue independently in parallel.*

`_updateSanRate()` consists of two parts: In the first half the locked interest accrued in previous blocks are added to the sanRate while in the second part the new amount of tokens to be distributed are added to the locked interests. These should be distributed in the next block this function is executed.

`lastBlockUpdated` however is only updated if some of the locked interest where added to the sanRate in this block.

This leads to following corner case when `_updateSanRate()` is executed: In case there are no locked interest to distribute, `lastBlockUpdated` is not updated to the current `block.timestamp` but rewards to be distributed are added to `lockedInterests`. However a second call to `_updateSanRate()` will now convert these lockedInterests and add them to the sanRate before updating `lastBlockUpdated` blocking a future update in this same block.

The sanRate is updated upon collection of profit from strategies or when a part of the fees for stable seeker is distributed to standard liquidity provider.

The profit from strategies can be attacked as follows:

At a time where `lockedInterests` is equal to zero and a strategy has a significant amount of rewards to collect, the attacker executes the following steps:

- Deposits a large amount of collateral (e.g. acquired through a flashloan) for San Tokens. While `updateSanRate()` is called, this currently has no effect as `lockedInterests` is equal to zero and the amount to distribute is 0, so `lockedInterests` remains zero.
- Call `harvest()` on the Strategy. This collects the profit and executes `_updateSanRate()`. As currently no locked interests are to be distributed, the first part is skipped and *lastBlockUpdated*` remains unchanged. In the second part the

`lockedInterests` to be distributed in the future are updated.

- Withdraw the collateral by burning the san tokens. `_updateSanRate()` is executed once again, this times with `lockedInterests` being nonzero the sanRate is now actually updated and `lastBlockUpdated` is set to the current block. Hence the user can withdraw more collateral than deposited.

This may be abused to drain the profit of the strategy.

Note that this is a rough description only and the actual execution of this attack is a bit more complicated: In order to extract most of the protocols interest more calls will be needed than described above. As the initial `deposit()` by the attacker will have significantly increased the amount of total assets available to

the PoolManager, the PoolManager will push a lot of funds into the strategy during the call to report (in order to keep the planned `debtRatio`). Hence the `withdraw()` cannot really withdraw sufficient amounts. Multiple calls with carefully crafted arguments to `withdraw()` and `harvest()` are necessary to complete the attack successfully and repay the flashloan.

Code corrected:

`lastBlockUpdated` is now updated each time `updateSanRate()` is executed, this prevents the issue described above.

7.13 Governance Not Fully Propagated

Correctness Medium Version 1 Specification Changed

The documentation specifies the following:

```
The Core contract has the ability to add a new governor or remove a governor from the
system and propagate this change across all underlying contracts of the protocol.
```

Similarly, the Core contract should propagate guardian changes. However, that is not the case for some contracts. For example, the changes are not propagated to OracleMulti or RewardsDistributor. That mismatches the specification. Fortunately, the governance can use functions `grantRole()` and `revokeRole()` to perform the changes jointly with the functions from Core.

Specification changed:

The documentation has been updated and now describes how the governance change propagates from the Stablemaster. Additionally the code of the core contract now contains following comment:

> Keeps track of all the *StableMaster* contracts and facilitates governance by allowing the propagation of changes across most contracts of the protocol (does not include oracle contract, *RewardsDistributor*, and some

7.14 Guardian Cannot Be Managed by Guardian

Correctness Medium Version 1 Code Corrected

The documentation specifies the following:

```
The guardian is indeed able to transfer its power to another address or to revoke itself.
```

However, that is not possible. Core functions `setGuardian` and `revokeGuardian` call the inherited `grantRole()` and `revokeRole()`. The administrator of the guardian role is the governor role. Thus, the calls `grantRole()` and `revokeRole()` would fail since the guardian is not allowed to access these and the guardian cannot set or revoke guardians.

Code corrected:

Access control has been reimplemented. In the new implementation the guardian can transfer its power to another address or revoke itself.

7.15 Incorrect Cash Out Amount

Correctness | Medium | Version 1 | Code Corrected

Existing perpetual positions can be updated. Any perpetual position that is modified through `addToPerpetual` or `removeFromPerpetual` results in the wrong `cashOutAmount`. Please consider the following example in which all transactions happen shortly after each other. Hence, we assume that the oracle prices do not change and are 1000 and 1125 respectively. Please note that changing oracle prices can make the problem worse. We will also ignore fees in this example.

1. A new position is created and its `cashOutAmount` = 10 ETH, while `committedAmount` = 20 ETH. The `initialRate` = 1125.
2. The position is updated through `addToPerpetual` and 1 ETH is added. The new committed amount is calculated as 20 ETH * 1125 / 1000 = 22.5 ETH. Hence the new `cashOutAmount` is calculated as 20 ETH + 10 ETH - 22.5 ETH + 1 ETH = 8.5 ETH. The `initialRate` remains 1125.
3. The user performs a cash out using `cashOutPerpetual`. The newly committed amount is calculated as 20 ETH * 1125 / 1000 = 22.5 ETH. Hence the new `cashOutAmount` is calculated as 20 ETH + 8.5 ETH - 22.5 ETH = 6 ETH. Therefore, the user receives 6 ETH, despite depositing 11 ETH.

In short, whenever the oracle rates significantly deviate from each other, users can lose significant value. This issue can grow in severity with fees, repetitive operations and price fluctuations.

Code corrected:

This issue has been addressed by only storing the initial rate. Hence, errors can no longer accumulate with the number of actions.

7.16 Non 18 Decimals Protocol Tokens

Correctness | Medium | Version 1 | Specification Changed

The documentation states:

```
Decimals

To be consistent with the BASE chosen when computing numbers, it has been decided that all
the ERC20 tokens created by the Angle protocol would involve 18 decimals.

Although this is not specified anywhere in the code, this means that the base for agTokens
and sanTokens is 18.
```

The decimal of the sanToken however is set equal to the decimal of the underlying collateral. For collaterals with decimals different than 18, the sanTokens decimal will not be equal to 18.

```
function initialize(
    string memory name_,
    string memory symbol_,
    address poolManager
) public initializer {
    __ERC20Permit_init(name_);
    __ERC20_init(name_, symbol_);
    stableMaster = IPoolManager(poolManager).stableMaster();
    decimal = IERC20MetadataUpgradeable(IPoolManager(poolManager).token()).decimals();
}
```

Specification changed:

The developer documentation will be changed to accurately reflect this.

7.17 Unaccounted Collateral, Unrestricted `updateStocksUsersGov()`

Correctness | Medium | Version 1 | Code Corrected

Function `updateStocksUsersGov` of a stablemaster contract allows the Guardian / the Governance to update `col.stocksUsers` arbitrarily. There are no checks at all, e.g. whether the update respects the actual amount of free collateral available.

This variable is described as:

```
// Amount of collateral in the reserves that comes from users
// + capital losses from HAs - capital gains of HAs
```

While `col.stocksUsers` is updated as described during the actions of stable seekers and hedging agents an additional function `updateStocksUsersGov` exists allowing the Guardian / the Governance to change this variable arbitrarily.

Amongst others, this function is annotated with

```
Updates the `stocksUsers` for a given collateral to allow or prevent HAs from coming in
This function can typically be used if there is some surplus that can be put in `stocksUsers`.
```

The system surplus arises due to the fees collected by the system. When minting or burning stablecoin only a part of the fee collected is incorporated into the sanrate. The rest remains as surplus collateral in the poolmaster contract. When creating or cashing out a perpetual a fee is taken. This fee collected resides in form of unaccounted for collateral at the poolmaster. A part of the fee must be set aside while the perpetual is active as it may be needed to pay the keeper slashing this perpetual.

Note that not all balance of the collateral token held by the poolmaster is available to use freely. Some of this balance may belong to standard liquidity providers.

Overall there is no automatic accounting of the fees collected, the system rather relies on a manual update where the caller can freely specify the parameter. The description of the function hints that the function may be used to steer whether to allow/prevent more HAs from coming in. Note this can also steer whether perpetual can be cashed out forcefully. Allowing the update of this value without any checks may let the system reach an incorrect state.

Code corrected:

The function name was changed to `rebalanceStocksUsers`. It reduces the `stocksUsers` of one collateral and adds it to another one. However, the cap for the maximum stocks users value cannot be exceeded with this operation. Hence, the number of stablecoins minted in total stays the same. For more information see the description of System Accounting.

7.18 Unit Errors for Tokens With Decimals Different Than 18

Correctness Medium Version 1 Code Corrected

Perpetuals earn a reward in form of governance tokens. Additionally the staking contract may allow AgToken and SanToken to be staked in order to earn governance tokens. For both, the calculation of the reward does not work correctly for collaterals with decimals different than 18.

Using the example of the PerpetualManager, the reward per committed collateral token of the perpetual is calculated as follows:

```
function _rewardPerToken() internal view returns (uint256) {
    if (totalCAmount == 0) {
        return rewardPerTokenStored;
    }
    return
        rewardPerTokenStored + ((_lastTimeRewardApplicable() - lastUpdateTime) * rewardRate * BASE) / totalCAmount;
}
```

and

```
function _earned(uint256 perpetualID) internal view returns (uint256) {
    return
        (perpetualData[perpetualID].committedAmount *
            (_rewardPerToken() - perpetualRewardPerTokenPaid[perpetualID])) /
        BASE +
        rewards[perpetualID];
}
```

The governance token, the angle token has 18 decimals. Hence `rewardPerTokenStored` and the returned value of `_rewardPerToken()` should be in 18 decimals as well for the calculation in `_earn()` to work correctly.

In both calculations however, `BASE` is used as unit instead of the actual base of the collateral. As a consequence, the calculation breaks for tokens not having 18 decimals.

Code corrected:

Now, the calculations are done correctly.

7.19 `safeApprove` Not Used, USDT Not Supported

Design Medium Version 1 Code Corrected

Since not all ERC-20 tokens adhere to the standard, it is recommended to use `safeApprove` such that interactions with a broader range of tokens are possible. Especially, this is important since interactions with some tokens, e.g. USDT, require `safeApprove`. However, in function `_changeTokenApprovalAmount` in PoolManagerInternal.sol a simple approve call is made with regards to the underlying pool tokens. As this method is used during the deployment of some collateral to give infinite approval to the perpetual manager and the stable master, this means that USDT pools cannot be deployed.

Code corrected:

The new version of the code now uses either `safeApprove` or `safeIncreaseAllowance`.

7.20 BondingCurve Specification Mismatches

Correctness Low Version 2 Code Corrected

There are several mismatches between the implementation of BondingCurve and its documentation. Examples are:

- The `power` parameter is fixed in the implementation. However, it is not stored anywhere but instead the formulas have been implemented assuming `power` to be two. In contrast, the specification states only that `power` should be strictly greater than one.
- The documentation specifies that the guardian should have the same powers as the governors with the exception of recovering tokens.
- Moreover, the code is divided in different sections. `changeOracle()` is in the guardian role section. Both documentation and code structuring imply that this function should be callable by the guardian. However, only governors can call this function.

The inconsistencies may confuse users.

Code corrected:

The specification has changed for the power parameter while the code has been corrected to restrict the guardian's power.

7.21 Gas-inefficient Strategies

Design Low Version 2 Code Corrected

The gas consumption of the strategies could reduced by reducing the storage reads. Some examples of inefficiencies in the strategies are:

- `_estimateAdjustPosition`: the element with the highest and the element with the lowest APR are searched. The code iterates through the `lenders` array twice and always reads from storage. Storage reads could be reduce by a factor of two.
- `_removeLender`: `lenders[i]` is read first in the `if` condition and then in the first line of the `if` body.
- `_withdrawSome`: in the `while` loop the `for` loop reads always from storage, hence wasting gas.

Code corrected:

Gas consumption has been reduced for the functions.

7.22 Inefficiency in Binary Search

Design Low Version 2 Code Corrected

The binary search within the `_piecewiseLinear` function works as follows:

```
uint256 lower;
uint256 upper = xArray.length - 1;
uint256 mid = (upper - lower) / 2;
while (upper - lower > 1) {
    if (xArray[mid] <= x) {
        lower = mid;
    } else {
        upper = mid;
    }
    mid = lower + (upper - lower) / 2;
}
```

Here the following improvements can be made:

1. The initial value of `mid` is computed using the wrong formula as the computation should read `upper + lower` rather than `upper - lower`. However, it doesn't matter in the current code version as `lower` is always initialized to 0. Hence, it is unclear why `lower` is part of this computation.
2. The value of `mid` is needlessly computed once at the end of the loop. This could be refactored to save a computation of `mid`.

The gas savings of these improvements are negligible, however, they might contribute to more maintainable code.

Code corrected:

The code is now more gas-efficient.

7.23 No Slippage Protection in BondingCurve

Security | Low | Version 2 | Code Corrected

The BondingCurve contract allows the purchase of governance tokens (most likely Angle tokens) in exchange for other tokens. However, the function `buySoldToken` does not protect users from growing prices. The user could experience an unexpectedly trade result if they have given a high or infinite approval to the BondingCurve contract.

Code corrected:

Users can now specify the maximum amount of AgTokens they are willing to pay for the specified amount of ANGLE tokens.

7.24 Reference Coin Changes May Affect the Bonding Curve

Correctness | Low | Version 2 | Code Corrected

The `BondingCurve` contract contains a `referenceCoin` variable. This `referenceCoin` can be set to the zero-address by being revoked. This may affect the BondingCurve in several ways:

- `getCurrentPrice()` will return the price in the reference coin. However, there is no reference coin.

- `buySoldToken()` will take the oracle value based on the previous stablecoin. However, having the 0-address suggests that the reference price is currently to be determined.

Similar issues may occur if the `referenceCoin` is set then to another token. Now, if the oracles are not updated, the price will differentiate highly from what governance would have expected. Also, the `startPrice` variable is in the currency of the reference token. Thus, it could be possible that the bonding curve changes if the start price stays the same when the reference currency changes.

Code corrected:

The contract will be paused to give the governance time to change the parameters.

7.25 Specification Mismatch in Strategy

Correctness Low Version 2 Specification Changed

The documentation of `rewardAmount` specifies in Strategy.sol:

```
/// @dev If this is null rewards will never be distributed
```

In contrast, the documentation of `setRewardAmount` in Strategy.sol specifies:

```
/// @dev A null reward amount corresponds to reward distribution being activated
```

However, if the reward amount is null, then the rewards can be eventually distributed if the reward amount is changed. Moreover, the reward amount being null means that the reward distribution is deactivated.

Specification changed:

The specification has been changed to correctly specify the reward amount.

7.26 StableMaster Might Be Unnecessarily Paused

Correctness Low Version 2 Code Corrected

When the `signalLoss` function registers a loss exceeding `sanRate * sanMint` it will pause the `StableMaster` contract. However, during this calculation it does not consider any `lockedInterests` that were queued to increase the `sanRate`. Hence, when factoring in the correct `sanRate` pausing might not be necessary.

Code corrected:

The code has been corrected.

7.27 Cache Value Instead of Reading From Storage

Design Low Version 1 Code Corrected

In function `update` of the `PerpetualManager`, `perpetual.fees` is first updated and later read from storage in order be emitted in the event.

Caching the value would result in lower gas used.

Code corrected:

Function `_update` has been removed from the `PerpetualManagerInternal` contract.

7.28 Consistency Checks for Oracles Missing

Design Low Version 1 Code Corrected

ModuleChainlinkMulti checks whether `circuitChainlink` and `circuitChainlinkIsMultiplied` have the same length. In contrast, ModuleUniswapMulti does not check this property for Uniswap circuits.

Code corrected:

The check was added.

7.29 Different Calculation of Cashout Fees

Correctness Low Version 1 Code Corrected

Should the coverage of a collateral exceed the limit, the fees for withdrawal differ depending on whether the perpetual is cashed out using `forceCashOutPerpetual()` or `cashOutPerpetual()`:

- In `forceCashOutPerpetual()` the withdrawal fee is computed with a margin of 0 (representing the status before the cashout of this perpetual).
- In `cashOutPerpetual()` the withdrawal fee is computed based on the new margin calculated after the perpetual has been cashed out.

Code corrected:

The structure of the two functions has been changed. Both initially cashout the perpetual and then compute the withdrawal fee.

7.30 Double Getters

Design Low Version 1 Code Corrected

In Core.sol, `governorList()` can be accessed through the automatically generated `governorList` getter and through the manually implemented `getGovernorList()`. Having only one getter will reduce code size, gas consumption on deployment, and confusion.

In OracleAbstract.sol, `inBase` has two getters: the automatically generated one and `getInBase()`.

Code corrected:

`governorList` was made `internal` and `getInBase()` was removed.

7.31 Enhance Check During Deployment of Collateral

Design | Low | Version 1 | Code Corrected

`deployCollateral()` of the StableMaster contract checks that the passed arguments are non-zero address before the struct for the collateral is initialized.

Some of these checks may be made more thorough with low effort:

It may be checked whether the collateral token of the perpetual manager matches the collateral.

Furthermore, it is possible to create a shared SanToken for multiple pool managers which may lead to unwanted behaviour. Also the number of decimals of the SanToken is not checked when a new collateral is deployed.

Also the oracle is not checked for compatibility with the pool manager.

Code corrected:

Checks were added.

7.32 Events Missing

Design | Low | Version 1 | Code Corrected

Even though many events are emitted by the protocol, not all important state changes emit events. For example, guardians are allowed to set a new fee manager with function `setFeeManager()` in StableMaster. As this sets a new address as a fee manager, emitting an event could help users notice this change. Another example is that not all ERC-721 events are emitted. For example, no events are emitted for approvals.

Code corrected:

ERC-721 event are now emitted. For `setFeeManager()`, an event was added to the StableMaster contract.

7.33 Gas Inefficiencies When Removing From List

Design | Low | Version 1 | Code Corrected

Core keeps track of governors and stable masters with `governorList` and `stablecoinList`. StableMaster keeps track of all pool managers with `managerList`. PoolManager records all active strategies in `strategyList`. Each strategy registers lenders in `lenders` and RewardsDistributor keeps all staking contracts in `stakingContractList`.

In all cases, elements from the arrays can be removed. Removing a list item is always done using the following scheme (code from removing a governor in Core).

```
for (uint256 i = 0; i < governorList.length - 1; i++) {
    if (governorList[i] == _governor) {
        indexMet = 1;
    }
    if (indexMet == 1) {
        governorList[i] = governorList[i + 1];
    }
}
require(indexMet == 1 || governorList[governorList.length - 1] == _governor, "governor not in the list");
governorList.pop();
```

Assume the element to be removed is the first in the array. That shifts all elements by one position creating many storage reads and writes. Since the order of the array is not system relevant, the item to be removed could be, if found, overwritten with the value of the last element in the array, and the last entry could then be popped. That would reduce the number of storage reads and writes significantly for large arrays and, hence, reduce gas consumption.

Code corrected:

Instead of moving all element, the value of the last entry is written to the position of the element to be removed. Then, the last entry is popped.

7.34 Gas Inefficiencies When Searching Lists

Design | Low | Version 1 | Code Corrected

When pushing a new StableMaster to `stablecoinList` in `deployStableMaster()` of Core.sol, it is checked if the item to push is already in the array. However, once found the loop continues and does not early quit. The overhead in storage reads could be avoided in a similar way as in function `addGovernor()`.

Also, the search in `_piecewiseLinear()` could be optimized. Since `xArray` is sorted, a binary search may reduce the total number of operations if the array is large enough.

Code corrected:

Checking whether an element is pushable to an array is now implemented using a mapping to booleans, faciliating the search. For the piecewise linear interpolation, a binary search was implemented.

7.35 Governance Changes

Design | Low | Version 1 | Code Corrected

The documentation specifies the following:

```
The way a governance change occurs is that it is notified by governance (or by the guardian) to the Core
which then propagates this change to all the StableMaster contracts of the protocol. Each StableMaster then
notifies the AgToken contract it relates to as well as all the PoolManager.
```

Changes in governance should be propagated from Core to all other contracts.

However, usage of access control functions inherited from OpenZeppelin's contracts may lead to inconsistencies.

For example following scenarios could occur:

- In function `removeGovernor` in Core.sol, the call reverts if there is only one governor. However, it is still possible to remove the last governor by calling one of the inherited methods `renounceRole` or `revokeRole`. Ultimately, the check in `removeGovernor()` can easily be circumvented.
- A governor may change the Core role using `grantRole()` and `revokeRole()` for a StableMaster. This may lead to inconsistencies with the `core` state variable in StableMasterStorage.sol. Moreover, removing or adding a guardian or governor would always revert in such a scenario (if the change is not manually undone). Also, multiple core contracts could be allowed in StableMaster.
- A governor could grant or revoke a governor or guardian role to someone in Core. These changes are not propagated and may lead to inconsistencies in governance between the different contracts.

Code corrected:

The core does not inherit any access control functionality anymore. The access control for Core was customly implemented. Thus, the issues cannot occur anymore.

7.36 Inconsistencies Between Staking Contract

Design | Low | Version 1 | Code Corrected

There are two types of staking contracts. The StakingRewards and the perpetual managers. They have similar functionality and their staking mechanism should be similar. However, inconsistencies in their implementations can be found.

Some of the inconsistencies are:

- Difference in `setNewRewardsDistributor()`: The perpetual manager checks whether a new reward distributor contract has the same reward token as itself. StakingRewards does not do that.
- Difference in emitting events: In the above function the two contracts emit different events.
- `Recovered` event is not emitted in `recoverERC20()` in PerpetualManager but in StakingRewards.

Code corrected:

The same events are now emitted and the RewardsDistributor, the only contract allowed to call the staking contracts' `setNewRewardsDistributor()`, checks whether the new rewards distributor has the same reward token as itself.

7.37 Inconsistent Parameters in RewardsDistributor Possible

Design | Low | Version 1 | Code Corrected

In contract RewardsDistributor consistency checks are missing and some parameters could contradict each other. Examples are:

- No check that `StakingParameters.updateFrequency` is smaller than `StakingParameters.duration` in function `setUpdateFrequency`.
- No check that `StakingParameters.updateFrequency` is smaller than `StakingParameters.duration` in function `setDuration`.

Code corrected:

Checks were added in the mentioned functions.

7.38 Incorrect Comment

Correctness Low Version 1 Specification Changed

In `PerpetualManagerFront.forceCashOutPerpetual()` there is a check whether a perpetual can forcefully be cashed out due to the maximal collateralization amount has been exceeded:

```
// Now checking if too much collateral is not covered by HAs
(uint256 currentCAmount, uint256 maxCAmount) = _testMaxCAmount(0, rateUp);
// If too much collateral is covered then the perpetual can be cashed out
canBeCashedOut = currentCAmount > maxCAmount ? 1 : 0;
```

The "not" in the first comment is incorrect. The code is checking if too much collateral is currently covered by HAs.

Specification changed:

The comment has been corrected to suit the modified force cashout functionality.

7.39 Inefficient Structs

Design Low Version 1 Code Corrected

There are multiple structs with multiple `uint256` fields. Each of these fields uses a full storage slot of 32 bytes. Storing data in Ethereum is expensive. A significant amount of gas is used when reading from or writing to storage. For example struct `SLPData` stores eight `uint256`. Given the nature of the data stored in `SLPData` all of these variables would not need to be of type `uint256`. Using smaller datatypes would allow to group multiple of the variables into one storage slots. If done appropriately, this would reduce the total amount of storage reads/writes resulting in lower gas costs. The same applies for other structs which could be optimized similarly.

Code corrected:

Smaller datatypes have been chosen for some parameters and, hence, gas consumption has been reduced.

7.40 No Check if `onERC721Received` Is Implemented

Design | Low | Version 1 | Code Corrected

Perpetuals are treated as NFTs and are implemented as ERC-721 tokens. Much code is adapted from OpenZeppelin's ERC-721 implementation. In `createPerpetual()` of PerpetualManagerFront.sol, `_mint()` is is used to mint tokens. The documentation of `_mint()` specifies that using this method is unsafe and `_safeMint()` should be used. However, the `_safeMint()` method was removed when code from OpenZeppeling was adapted. The intention behind this function is to check if the address receiving the NFT, if it is a contract, implements `onERC721Received()`. Thus, there is no check whether the receiving address supports ERC-721 tokens and perpetuals could be not transferrable in some cases.

Code corrected:

`mint()` checks if a receiving contract implements `onERC721Received()`.

7.41 No Checks Performed in Constructor

Design | Low | Version 1 | Code Corrected

Contrary to the constructor of ModuleChainlinkMulti, the constructor of ModuleUniswapMulti does not perform sanity checks on the length of `_circuitUniswap` and `_circuitUniIsMultiplied`.

Code corrected:

The check was added to the constructor of ModuleChainlinkMulti.

7.42 Outdated Compiler Version

Design | Low | Version 1 | Code Corrected

The project uses an outdated version of the Solidity compiler.

```
pragma solidity 0.8.2;
```

Known bugs in version 0.8.2 are:

https://github.com/ethereum/solidity/blob/develop/docs/bugs_by_version.json#L1530

More information about these bugs can be found here: https://docs.soliditylang.org/en/latest/bugs.html

At the time of writing the most recent Solidity release is version `0.8.6` which contains some bugfixes but no breaking changes.

Code corrected:

In the meantime, the compiler version has been updated to `0.8.7` which was also set in the hardhat configuration file.

7.43 Overhead Due to Loading Struct Into Memory

Design Low Version 1 Code Corrected

Inside the `else` branch in function `removeFromPerpetual` the whole perpetual struct of this perpetual is loaded from storage into memory. Given that only `perpetual.creationBlock` and `perpetual.committedAmount` are read later on, loading the whole struct into memory is an unnecessary overhead.

Code corrected:

The code of `removeFromPerpetual` has changed significantly, loading from storage into memory is now more efficient.

7.44 Possibly Failing Assert

Design Low Version 1 Code Corrected

In UniswapUtils.sol in function `_readUniswapPool()` an `assert` statement checks if the cast of `twapPeriod` from `uint32` to `int32` overflowed. However, this may fail and, thus, consume all remaining gas. The usage of `require`, in contrast, would refund the remaining gas to the user.

Actually, the value is checked in the constructor and could be checked in function `changeTwapPeriod`, removing the need for checking it in every execution of `_readUniswapPool()`.

Code corrected:

The check is now in the constructor and the setter. Moreover, the `assert` has been replaced with a `require` statement.

7.45 Potential Confusing `readLower(uint256 lower)`

Design Low Version 1 Code Corrected

Function `readLower` defined in OracleAbstract returns the lower rate if parameter `lower` is equal to 1 or else the higher rate returned by `_readAll()`.

The function name increases the risk that the function is used incorrectly in the future. It could be considered to split this functionality in two functions with distinct names.

Code corrected:

`readLower()` always returns the lower rate. `rateUpper()` was introduced to get the upper rate and avoid confusion.

7.46 Reward Token Issues

Design Low Version 1 Code Corrected

The documentation specifies that governance could decide to choose a reward token different from ANGLE. Thus, RewardDistributor must be generic in terms of tokens. However, some inconsistencies can be found.

- event `ANGLEWithdrawn` is emitted in function `governorWithdrawRewardToken`. The event name is inconsistent with the possible use cases.
- In `_incentivize()`, the error message in the `require` statement specifies that an ANGLE transfer failed. That does not have to be the case since it could be the COMP token.
- Function `_incentivize` uses `transfer()`. Non ERC-20 compliant tokens may fail, e.g. USDT is unsupported as a reward token. Generally when interacting with unknown ERC-20 tokens the `safeXYZ` functions may be used. These wrappers allow a safe interaction with non-compliant ERC-20 tokens.

Furthermore, since `rewardToken` cannot be modified, it can be made immutable.

Code corrected:

The code, comments and the naming was generalized to fit the general purpose of this class as specified in the documentation. Furthermore, `safeXYZ` functions are used to support a broader ranger of reward tokens. Also, `rewardToken` is now immutable.

7.47 Specification Mismatch in OracleMath

Correctness Low Version 1 Specification Changed

The OracleMath contract implements functionality for retrieving UniSwap rates. The documentation specifies the following:

```
/// @return rate uint256 representing the ratio of the two assets `(token1/token0) * decimals(token1)`
```

However, this is not correct. The specification should specify that the rate is multiplied with base `10**decimals` instead of the number of decimals.

Specification changed:

The specification was changed to document the correct base.

7.48 Unnecessary Double Checks

Design Low Version 1 Code Corrected

The Angle Protocol uses the access control library of OpenZeppelin to restrict access to some functions. However, some functions have double checks whether a caller can execute a certain function. This occurs when rights are granted or revoked. For example, in StableMaster.sol in `deploy()`, the core sets the governors and guardian. First, the `onlyRole` modifier of `deploy()` checks whether the caller has the appropriate role or not. Then, for each governor `grantRole()` is called from the access control library. `grantRole()` calls the `onlyRole` modifier. Thus, many redundant checks are executed due to the modifier of `deploy()` and the repetitive calls to `onlyRole` modifier in the `grantRole()` function. Similar inefficiencies occur in other contracts and function with `grantRole()` and `revokeRole()`.

Code corrected:

OpenZeppelin's access control libraries have been forked and modified such that `_grantRole()` and `_revokeRole()` are now `internal`. Now, these, instead of the public methods, are used to avoid double checks.

7.49 Updated SanRate When Converting to SLP

Correctness Low Version 1 Code Corrected

Function `convertToSLP` of the StableMaster contains following commented out code:

```
// we could potentially add
// _updateSanRate(0, col);
```

This function is used during the cash out of perpetuals should there be an insufficient amount of collateral available. It converts an amount of collateral into santokens, hence should be treated equally as depositing this amount of collateral. The san rate should be updated indeed, this distributes accrued interests which have been collected before the collateral of this HA is converted into san tokens.

Note that contrary to the `deposit()` function, there is no check whether the stablemaster is paused.

Code corrected:

The line was uncommented. Now, the sanrate is updated and it is checked if the contract is paused. Thus, the behaviour is consistent with `deposit()`.

7.50 Wrong Incentive for Which Perpetuals to Forcefully Cash Out

Design Low Version 1 Code Corrected Acknowledged

Whenever the covered amount of collateral exceeds the maximum allowed amount, keepers can freely choose which perpetuals to liquidate.

The incentive for how the perpetuals to be liquidated are chosen may not be ideal. Although the maximum fee for the keeper can be capped (depending on the parameter set by the governance), in general the reward for the keeper depends on the fee the perpetual has paid.

- Although fees are variable, generally larger perpetuals have paid more in fees and hence may be more attractive for keepers

to forcefully cash out.

- Using functions `addToPerpetual()` or `removeFromPerpetual()` increases the fees paid by the perpetual and hence increases the

risk of the perpetual to be selected by keepers.

For the system however, it would be more beneficial if keepers choose to liquidate perpetuals which bring the covered amount just short of the limit for the maximum amount to be covered. E.g. the coverage limit may be 90%, however currently 91% is covered. Multiple perpetuals exist, one of them may cover 2% while another covers 20%. In case a keeper cashes out the perpetual that covered 20%, the system now only has roughly 70% of its collateral covered, significantly below the targeted 90%. If the keeper however had chosen to cashout the smaller perpetual, the resulting new covered amount would have been just short of the target.

Code corrected:

The system introduced changes on how fees are computed. Keepers are now earning more fees if they cashout/liquidate perpetuals so that the covered amount is close to the target amount.

Acknowledged:

Since now multiple perpetuals can be liquidated at the same time, cashing out multiple perpetuals will cost more gas than one big one. Angle acknowledged that a commented:

```
In a future protocol upgrade, we could weight the amount of fees going to keepers by using
another piecewise linear function that depends on the number of perpetuals cashed out:
this would kill the incentive to only cash out in priority the biggest perpetuals.
```

7.51 `capOnStablecoin` May Be Violated by Guardians

Design | Low | Version 1 | Code Corrected

`capOnStableCoin` is documented as follows:

```
/// @notice Maximum amount of stablecoin in circulation
```

Assume that currently 1000 AgUSD are minted. Then a guardian may call `setCapOnStablecoin` and set `capOnStablecoin` to 500. Even though no new stablecoins can minted the invariant `AgUSD.totalSupply() <= capOnStablecoin` could be violated in such a scenario.

Code corrected:

`capOnStablecoin` was removed. However, there is now a cap on the issueable stablecoins per collateral for which it is checked that it is always higher than or equal to `stocksUser`.

8 Notes

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

8.1 Frontrunning Keepers

Note Version 1

Keepers are responsible to keep the system in balance. As an incentive they collect part of the fees. However, another party could see the transactions coming from the keepers and front run them without much computational effort. The incentive for keepers may be lost.

Note that Angle is aware of this and it's documented in the code above the respective keeper functions:

```
As keepers may directly profit from this function, there may be front-running problems with miners bots,
/// we may have to put an access control logic for this function to only allow white-listed addresses to act
/// as keepers for the protocol
```

8.2 New BASEFEE Opcode

Note Version 1

The recent hardfork introduced EIP-1559 and EIP-1398. While the first EIP introduces a new fee system for transaction, the later introduces a new opcode BASEFEE allowing to query the new basefee parameter of the transaction.

Roughly speaking the previous gasprice now consist of the basefee + a tip. Overall transaction prices are now much more predictable.

Appropriate Keeper rewards should cover their transaction costs and an additional incentive. While previously the GASPRICE opcode could not really be used as this opened possibilities for abuse for miners, the new BASEFEE opcode is now much more suitable. It may be considered to use it as base for the keeper reward calculation.

8.3 Oracle-related Issues

Note Version 1

The system uses two different oracles: Chainlink and Uniswap.

In principle these oracles provide reliable data sources, however, it is not impossible to manipulate them. For Uniswap the cost of manipulation depends on the liquidity and the activity within the affected pools.

As volumes increase within the system the following oracle-based attacks become possible. We split them between attacks that require manipulation of one oracle and attacks that require manipulation of both oracles.

8.3.1 Single-Oracle Manipulation

- Whenever there is a big mint or burn operation of agTokens, existing system participants have an incentive to attack this mint or burn with a manipulated oracle. As a result the system will accumulate a surplus. The value of the attack is limited by the value of the mint or burn.

- When there an innocent user cashes out a perpetual, existing system participants have an incentive to manipulate an oracle in order to increase the system surplus. The value of the attack is limited by the value of the perpetual position.
- A malicious user could perform an oracle attack to liquidate a large percentage of the perpetual positions to collect the keeper fees. The value of the attack is limited by the combined keepers fees of the perpetual positions.

8.3.2 Multi-Oracle Manipulation

- A malicious user can extract large amounts of collateral by manipulating both oracles before performing an agToken burn operation. The value of the attack is limited by the total collateral deposited of this type.
- A malicious user can extract large amounts of collateral by manipulating both oracles before performing a cashout of a perpetual position. The value of the attack is limited by the total collateral deposited of this type.

As seen from the list, some of these attacks increase in impact as the system accumulates more liquidity. Hence, the risk of such attacks grows with the rise of the system and hence needs to be monitored.

Please note that some of the mentioned attacks against other users can be evaded through slippage protections as mentioned in the separate issue above.

8.4 Perpetual NFTs May Not Be Composable With Other Protocols

Note Version 1

Perpetual positions are represented as NFTs. That allows them to be transferrable. Thus, users may want to sell the NFTs on secondary marketplaces. However, the design of the NFTs is not composable with other protocols.

1. A perpetual is opened and an NFT for it is issued.
2. The user wants to sell the NFT on a marketplace. The NFT is deposited on a marketplace contract.
3. The perpetual is force-closed by a keeper. The NFT is burned.

The marketplace, as the current owner, receives the underlying funds while the NFT gets burned. However, the marketplace is unaware of the NFT being burned and the funds being received. Thus, the underlying funds could be lost.

8.5 SLPs Timing Their Entry

Note Version 1

Although the san rate is updated before a new standard liquidity provider enters the system, a new SLP may still profit from interest accrued previously:

- Due to the maximum update of the san rate in one block there may be some locked interest

which are to be distributed in the next / over the next block.

- An outsider may observe the performance of the strategies and may forsee that a call to

`Strategy.harvest()` will be profitable.

In both scenarios an SLP entering at the right time may benefit from interests accrued before his participations at the cost of other participants.

To mitigate both, `Strategy.harvest()` should be called frequently in order to distribute the rewards accrued smoothly.

8.6 Setting Variables May Lead to Inconsistent State

Note **Version 1**

When setting a new core from the old core, it is ensured that the governors and the guardian of both cores are the same. Similarly, the `stablecoinList` is checked. However, `deployedStableMasterMap` is not checked. That should be ensured in the constructor of the new core contract. Otherwise, stable masters could be redeployed.

Moreover, a guardian can set a new fee manager through the stable master contract. However, it is never checked whether this fee manager has the same governance structure. The governance structure must be setup in the constructor.

8.7 Special ERC-20 Token Behavior May Be Problematic

Note **Version 1**

Tokens with fees or rebasing tokens may be errorneous if added as collateral. Some ERC-20 tokens have transfer fees. Supporting such tokens as collateral for a stablecoin may lead to accounting errors. When a user mints some AgToken in exchange for collateral, he specifies how many collateral tokens should be transferred from him to the pool manager. This amount is used to update `stocksUsers`. However, the amount received by the pool manager may differ from the amount specified by the user due to transfer fees. Not only would accounting issues occur but also too many AgTokens would be minted. The amount of AgTokens the user would receive depends on the amount he sent but not on the amount the pool manager received. To conclude, the current system will not work as intended if tokens have fees.

Similarly this applies for rebalancing tokens where the balance of token holders changes.

8.8 System Accounting

Note **Version 2**

Certain system states are not tracked by the smart contracts and need to be tracked by interested users and the operators separately in order to properly react.

1. Bad debt and system surplus are not obviously visible in the system but can be computed by retracing all the relevant system actions or by through a computation based on system variables and token balances.
2. The amount of stablecoins minted against a particular collateral (which is newly saved `stocksUsers`) can, for different reasons, become out-of-sync with the actual backing collateral. The operators need to step in by adjusting parameters accordingly.
3. Certain stablecoin-collateral imbalances can be rebalanced using the system function, however, this only works if there is a roughly matching positive and negative imbalance.

4. Certain payments, such as the fees paid by hedging agents are not being accounted but generally support the system's health. Users hence need to query token balances to evaluate collateral value.

8.9 The Devil Takes the Hindmost

Note Version 1

Contrary to other similar systems, the loss of the system is not evenly distributed across all participants. For example, such loss may stem from uncovered collateral and a decreasing price, or keepers not liquidating perpetuals timely. In such situations, the system continues to operate normally as long as there are sufficient funds available. The first actors redeeming / withdrawing their assets get everything at market prices while slow users are left behind as they can no longer burn their stable tokens, redeem their san tokens or cash out their perpetuals due to insufficient funds.

8.10 Transferable Perpetuals and Reward

Note Version 1

Perpetual positions are NFT tokens adhering to the ERC-721 standard, hence perpetuals are transferable. Note that Perpetuals are eligible to earn a reward. While such a reward is associated with the NFT, holders of the perpetual should be aware that it's to their advantage to claim their reward before transferring the perpetual.

8.11 Unbounded Decrease of `sanRate`

Note Version 2

The `sanRate` which is the conversion rate of `SanTokens` cannot increase arbitrarily to limit economic attacks. However, it can decrease arbitrarily when a strategy reports losses. Such a change is harder to exploit by an economic attacker, however, it is still possible if there exists a platform when `SanTokens` can be borrowed. Through such a borrow operation the `sanRate` drop can be exploited.