Angle Labs

Angle Protocol

Smart Contract Security Review

Version: 2.0

September, 2021

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Angle smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contracts. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Angle smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Angle smart contracts.

Overview

Angle Protocol is a Decentralised Finance (DeFi) protocol that issues over-collateralised stablecoins. Angle Protocol provides a risk market between stablecoin seekers, hedging agents, and standard liquidity providers. Angle Protocol also runs strategies to improve the efficiency of the collateralised assets by lending them to other platforms such as Compound to get a higher yield which will be distributed to users, which in this case, are treated as stakers.

Angle Protocol works identical to popular futures product called perpetual contract but on DeFi settings. Oracles play an important role to keep track of the collateralised assets, while keepers execute actions in timely manner.

Angle Protocol is managed by Governance contract. There is also a guardian role which can act swiftly in emergency situations.

Security Assessment Summary

This review targeted commit 2819ccd hosted on the `angle-core` repository.

Specifically, the following smart contracts/folders were part of the scope of this security assessment:

- `agToken/Agtoken.sol`
- `bondingCurve/bondingCurve.sol`
- `core/Core.sol`
- `dao/ANGLE.sol`
- `poolManager/*`
- `oracle/*`
- `feeManager/*`
- `perpetualManager/*`
- `sanToken/*`
- `stableMaster/*`
- `staking/*`
- `utils/*`
- `collateralSettler/*`

Note: the OpenZeppelin libraries and dependencies were excluded from the scope of this assessment. Any other smart contract not included in the list above was also excluded from this review.

Retesting activities targeted commit 271e6bb and identified additional issues (from AGL-35 to AGL-39) related to a series of updates introduced by the development team.

The manual code review section of the report focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts. Specifically, their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focused on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [1, 2].

To support this review, the testing team used the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 39 issues during this assessment. Categorized by their severity:

- Critical: 1 issue.
- High: 2 issues.
- Medium: 4 issues.

- Low: 7 issues.
- Informational: 25 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Angle smart contracts in-scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
AGL-01	Amount Zeroed Before Use in `recoverUnderlying()`	Critical	Resolved
AGL-02	Potential Exploits from Guardian Role	High	Resolved
AGL-03	Potential Flashloan Attacks on `sanRate`	High	Resolved
AGL-04	Lost Funds When Calling `addToPerpetual()` with a Negative Asset Value	Medium	Resolved
AGL-05	`setProportionalRatioGov()` Potentially Not Called Before `triggerSettlement()`	Medium	Resolved
AGL-06	Claim Collateral on Behalf of HAs	Medium	Resolved
AGL-07	Incorrectly Handled Edge Case in `_computeDripAmount()`	Medium	Resolved
AGL-08	Revoked `StableMaster` May be Deployed A Second Time	Low	Resolved
AGL-09	Potential Overflows in `BondingCurve`	Low	Resolved
AGL-10	`setGuardian()` May Overwrite Itself	Low	Resolved
AGL-11	Circumvention of Maintenance Margin	Low	Resolved
AGL-12	`_computeDripAmount()` is Non Linear in its Distribution	Low	Resolved
AGL-13	Potential Accumulation of Interests After Calling `signalLoss()`	Low	Resolved
AGL-14	Integer Overflow in `revokeStableMaster()`	Informational	Resolved
AGL-15	`Governor` Must Be a Contract	Informational	Closed
AGL-16	Suboptimal Search Iteration	Informational	Resolved
AGL-17	Suboptimal Delete Iteration	Informational	Resolved
AGL-18	Duplicate Oracle Allowed on `setOracle()`	Informational	Resolved
AGL-19	Identical `newFeeManager` & `oldFeeManager` Allowed in `setFeeManager()`	Informational	Resolved
AGL-20	Integer Overflow on Empty List in `onlyCompatibleInputArrays`	Informational	Resolved
AGL-21	Integer Overflow in `removeStakingContract()`	Informational	Resolved
AGL-22	Calling `updateHA()` inside `setHAFees()`	Informational	Closed

AGL-23	Revert When No `Strategy`	Informational	Closed
AGL-24	Emit Then Update on `addStrategy()`	Informational	Resolved
AGL-25	`Core` Does not Need to be `Initializable`	Informational	Resolved
AGL-26	Initialisation of Proxy Implementations	Informational	Resolved
AGL-27	Purchase of Zero Tokens from `BondingCurve`	Informational	Resolved
AGL-28	TWAP Period May be Set to Zero	Informational	Resolved
AGL-29	Additional Constructor Checks	Informational	Resolved
AGL-30	Event `ReferenceCoinChanged` is Unused	Informational	Resolved
AGL-31	Front-Runnable Functions	Informational	Closed
AGL-32	Draft OpenZeppelin Dependencies	Informational	Closed
AGL-33	Reduce `SLOAD` Instructions when Reading Storage	Informational	Resolved
AGL-34	Miscellaneous `AngleProtocol` General Comments	Informational	Resolved
AGL-35	Reentrancy When Closing Perpetuals	Low	Open
AGL-36	Unnecessary Update of `lastUpdateTime`	Informational	Open
AGL-37	Reverts for Nonexistent Perpetuals in `liquidatePerpetuals()`	Informational	Open
AGL-38	Gas Optimisation - Remove `onlyOwnerOrApproved` from `addToPerpetual()`	Informational	Open
AGL-39	Miscellaneous `AngleProtocol` General Comments 2	Informational	Open

AGL-01	Amount Zeroed Before Use in `recoverUnderlying()`		
Asset	`CollateralSettlerERC20.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The `recoverUnderlying()` function can be called by a governor to recover `amountToRedistribute`, that is the amount of underlying tokens not allocated to any claims (by users, SLP or HA). The claimed tokens will be transferred to a preferred address specified in `to` as a function input variable.

This function can only be called when the base amount has been computed through calling function `setAmountToRedistributeEach()`. However, there is an error on line [370-371] as seen below:

```
amountToRedistribute = 0;
underlyingToken.safeTransfer(to, amountToRedistribute);
```

In the code above, `amountToRedistribute` is zeroed before the tokens are transferred through `safeTransfer()`. The impact is that the `to` address receives 0 tokens and the remaining tokens are effectively locked in the contract and cannot be withdrawn.

Recommendations

Consider using a temporary variable to store the amount before being zero (thereby preventing reentrancy) as seen in the following example.

```
localAmountToRedistribute = amountToRedistribute;
amountToRedistribute = 0;
underlyingToken.safeTransfer(to, localAmountToRedistribute);
```

Resolution

PR #80 mitgates the issue by removing the function `recoverUnderlying()` to be replaced by a new function `recoverERC20()`. The new function allows a `governor` to transfer either the underlying token or any ERC20 tokens owned by the `CollateralSettlerERC20` contract to a destination address. For the case where `tokenAddress` is the `underlyingToken`, the amount transferred is no longer read from state after being modified, as seen in the following lines.

```
require(amountToRedistribute >= amountToRecover, "too big amount");
amountToRedistribute -= amountToRecover;
underlyingToken.safeTransfer(to, amountToRecover);
```

AGL-02	Potential Exploits from Guardian Role		
Asset	contracts/		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

Angle's threat model includes two main types of permissioned users, **Guardians** and **Governors**. The Governor role will be controlled by a DAO and will have the highest level of permissions, allowing direct withdrawal of funds from the protocol. The Guardian role is intended to be a multisig account owned by core developers and potentially trusted third parties which can be used to act quickly in case of an attack.

The Guardian role should not be allowed to directly withdraw funds from the protocol. However, there are four potential ways for the Guardian role to exploit its privileges to withdraw funds from the protocol:

- The first attack vector available to a Guardian is through the `PoolManager.addStrategy()` function. The `PoolManager` will transfer a large proportion of the funds owned by the protocol to a `Strategy`. The Guardian is allowed to add any arbitrary address as a new `Strategy` which will receive the protocols collateral tokens as an investment. Hence, they may add a malicious contract as the new strategy which receives tokens from the protocol then transfers these tokens to an attacker owned address.
- The second and third attack vectors are from manipulating the price by setting malicious oracles in the functions `StableMaster.setOracle()` and `BondingCurve.changeOracle()`:
  - Manipulating the oracle price in `StableMaster` could be exploited from a malicious user by taking out a large position in the `PerpetualManager`. Then using a malicious oracle to increase the price exponentially. This would increase the attackers `cashOutAmount` in the perpetual swap enough that they could withdraw all the collateral tokens in the protocol.
  - Similarly by setting an advantageous price in `BondingCurve` they could buy tokens for significantly less than what they are worth.
- Finally, a Guardian may extract all of the funds from the `RewardsDistributor` through the function `setStakingContract()`. This allows the Guardian to specify a contract to receive reward tokens and the amount of tokens that will be sent. By setting the staking contract to a malicious address, the Guardian could withdraw all reward tokens from the `RewardsDistributor`.

Recommendations

Consider updating these functions to only be allowed to be called by accounts with the Governor role, thereby preventing misuse or malicious use by Guardians.

Resolution

Commits 48ae7f8 and a14d3e4 resolve the issue by setting the access control to be Governor only (rather than Guardian) for the following functions:

- `PoolManager.addStrategy()`
- `StableMaster.setOracle()`
- `Bondingcurve.changeOracle()`
- `RewardsDistributor.setStakingContract()`

AGL-03	Potential Flashloan Attacks on `sanRate`		
Asset	`StableMasterInternal.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The function `_updateSanRate()` is used to distribute any fees or interest payments to the current holders of `SanTokens`. This is done by increasing the `sanRate` which will be multiplied against a user's token balance to determine the amount of collateral tokens they will receive when withdrawing.

There are two potential issues with the updating process which could allow an attacker to use a flashloan to take a large portion of the fees or interest that is to be paid to the `SanToken` holders.

The first issue is if no previous `lockedInterests` are present then `lastBlockUpdated` will not be updated. Thus, a user (e.g. a malicious flashloaner) who deposits when there is no `lockedInterests` is still eligible for any future interest payments or fees received during the block.

The second issue is related to the protection against distributing too high a quantity of fees at one time. If the fees are above a certain percentage of the total supply of `SanTokens` then only a portion of the income will be distributed. The following lines show the check if the proportion of fees is too high to be distributed in one block.

```
if (col.slpData.lockedInterests > (sanMint * col.slpData.maxSanRateUpdate) / BASE) {
    // Limit distribution
    col.sanRate += col.slpData.maxSanRateUpdate;
    // Substracting before dividing for rounding
    _lockedInterests = (_lockedInterests * BASE - sanMint * col.slpData.maxSanRateUpdate)
        / BASE;
} else {
    // Distribute all lockedInterests
    col.sanRate += (_lockedInterests * BASE) / sanMint;
    _lockedInterests = 0;
}
```

Here `sanMint` represents the total supply of `SanTokens` and `maxSanRateUpdate` is a fixed percentage, making the check above equivalent to $lockedInterests > totalSupply * maxRate$.

The exploit here is inherrent to a flashloan attack, that is because a flashloan attack desposits a significant sum of funds into `SanTokens`, making the total supply of `SanTokens` increase dramatically. Therefore, `_updateSanRate()` will happily distribute a much higher quantity of `lockedInterests` compared to the amount that would be distributed before the flashloan had been deposited.

Recommendations

There are two issues to be mitigated:

1. Allowing an attacker to receive income in the same block they deposit funds. This can be mitigated

by having `col.slpData.lastBlockUpdated` be updated to the current timestamp during every call to `_updateSanRate()`, even if no interest is distributed.

2. Increasing the total supply of `SanTokens` to allow for more income to be distributed at one time. This can be prevented by having a cap as an amount in tokens rather than a percentage. However, this value would need to be updated regularly as the total supply grows and falls over time. The mitigation of the first issue prevents flashloans from claiming the interest and thus this attack would only be able to be executed by large scale investors who may invest for two or more blocks.

Resolution

Commit a14d3e4 corrects the issue by adding the following code on line [86] of `StableMasterInternal.sol`.

```
col.slpData.lastBlockUpdated = block.timestamp;
```

The code above ensures that `lastBlockUpdated` is updated every time `_updateSanRate()` is called, thus preventing a user from minting `SanTokens` and claiming interest in the same block.

Additionally, `maxSanRateUpdate` is replaced by a new parameter called `maxInterestsDistributed`, which sets the maximum amount to be distributed as a fixed quantity rather than a percentage of total supply of `SanTokens`. This update prevents a user from being able to distribute more interest in one block by increasing the total supply of `SanTokens`.

AGL-04	Lost Funds When Calling `addToPerpetual()` with a Negative Asset Value		
Asset	`PerpetualManagerFront.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

When a perpetual value falls below the amount used to secure the swap, a perpetual is said to have a *negative position*. That is the perpetual `cashOutAmount` is less than zero. When there is a negative position for a perpetual, it should be liquidated and the amount used to cover the position should be transferrred to the protocol.

The function `addToPerpetual()` will check if the perpetual has a non-positive (≤ 0) position and if the condition is met the perpetual will be liquidated.

However, before the perpetual is liquidated the funds are transferred from the user to the protocol. These funds are not accounted for in the perpetual's position when performing the liquidation check as seen in the code below.

```
_token.safeTransferFrom(msg.sender, address(poolManager), amount);

// Getting the oracle price
(uint256 rateDown, uint256 rateUp) = _getOraclePrice();

// The committed amount does not change, there is no need to update staking variables here
(uint256 cashOutAmount, ) = _getCashOutAmount(perpetualID, rateDown);

if (cashOutAmount == 0) {
    // Liquidating the perpetual if it is unhealthy
    _liquidatePerpetual(perpetualID);
} else {
    // Add to perpetual
    ...
}
```

Since the perpetual is liquidated without the funds being accounted for they are essentially lost from the user's perspective.

Recommendations

We recommend moving `safeTransferFrom()` to within the `else` statement, thereby not transferring the funds when the perpetual is to be liquidated.

Resolution

This has been resolved in commit a02f0b0. The new implementation transfers the collateral tokens only if the perpetual passes the liquidation check (through internal function `_checkLiquidation()`), as identified in the following snippets from line [`183-189`].

```
(, uint256 liquidated) = _checkLiquidation(perpetualID, perpetual, rateDown);
if (liquidated == 0) {
 // Overflow check
 _token.safeTransferFrom(msg.sender, address(poolManager), amount);
 perpetualData[perpetualID].margin += amount;
 emit PerpetualUpdated(perpetualID, perpetual.margin + amount);
}
```

AGL-05	`setProportionalRatioGov()` Potentially Not Called Before `triggerSettlement()`		
Asset	`CollateralSettlerERC20.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

Contract `CollateralSettlerERC20` allows users to claim collateral tokens with their their stable tokens (and governance tokens). The claim process can only be started once `triggerSettlement()` is called by a `StableMaster` contract. While claiming, the contract requires some mandatory information, including `proportionalRatioGovUser` and `proportionalRatioGovLP`, both of which are set through the function `setProportionalRatioGov()` by a Governor.

There is a chance that function `setProportionalRatioGov()` is never called before `triggerSettlement()` is executed. If this happens, then there would be `Division or modulo by zero` error during the claim. This is because `proportionalRatioGovUser` and `proportionalRatioGovLP` are divisors/denominators in function `_treatClaim()` if `(amountGovToken > 0)` and `lp > 0`.

The cause is the following code snippet from line [416-424].

```
if (amountGovToken > 0) {
  // From the ‘amountGovToken‘, computing the portion of the initial claim that is going to be
  // treated as a preferable claim
  uint256 amountInCGov;
  if (lp > 0) {
   amountInCGov = (amountGovToken * BASE) / proportionalRatioGovLP;
  } else {
   amountInCGov = (amountGovToken * BASE) / proportionalRatioGovUser;
  }
```

This occurrence will make claiming not possible if the user wants to use governance token, and there is no remedy for this event. Function `setProportionalRatioGov()` cannot be called after `triggerSettlement()`, as defined in line [384]

```
require(startTimestamp == 0, "ratios cannot be modified after start");
```

Recommendations

Make sure `proportionalRatioGovUser` and `proportionalRatioGovLP` are not zero before calling `triggerSettlement()`.

Resolution

This has been resolved in commit e7c8e8b. The function now conducts an extra check to ensure all prerequisites are satisfied.

```
require(proportionalRatioGovLP != 0 && proportionalRatioGovUser != 0, "invalid proportion");
```

AGL-06	Claim Collateral on Behalf of HAs		
Asset	`CollateralSettlerERC20.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

When a collateral pool is being revoked users are able to make claims over the collateral. These claims are paid out based on preference. Users are able to attach governance tokens to their claims to give the claims a higher preference.

In `CollateralSettlerERC20`, the function `claimHA()` may be called by any user and will make a claim for the perpetual owner. As part of this function, the user is able to specify how much governance tokens are to be attached. Since there are no requirements for who the message sender is when claiming a perpetual, a malicious user may claim other users perpetuals with **zero** governance tokens attached.

The benefit of this attack is that the malicious user would have a higher priority when it comes to redeeming the tokens if they have attached governance tokens when claiming their perpetual.

Recommendations

Consider only allowing users who are either approved or the owner of a perpetual to be allowed to claim a perpetual. This can be achieved through a public getter of the function `PerpetualManagerInternal._isApprovedOrOwner()`.

Resolution

This has been resolved in commit 9065e85. The function can only be called by the perpetual owner or an approved account. The check is implemented in the following code from line [301].

```
require(perpetualManager.isApprovedOrOwner(msg.sender, perpetualID), "not approved");
```

AGL-07	Incorrectly Handled Edge Case in `_computeDripAmount()`		
Asset	`RewardsDistributor.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The function `_computeDripAmount()` calculates the amount of tokens that should be sent to the `StakingRewards` contract, dependant on the amount of time that has elapsed since the last reward distribution.

The following condition can be found on line [286-290] of `RewardsDistributor.sol` .

```
uint256 timeElapsed = _timeSinceStart(stakingParams);
uint256 timeLeft = stakingParams.duration - timeElapsed;
if (stakingParams.distributedRewards >= stakingParams.amountToDistribute || timeLeft == 0) {
    return 0;
}
```

The condition where `timeLeft == 0` is reached when at least `duration` amount of time has passed since `timeStarted` . In this case the rewards to be distributed is set to zero. This is not desirable as there may be rewards that should be distributed.

Consider the extreme case where we begin with `setStakingContract()` , we let `duration` seconds pass then call `drip()` . Now `timeLeft == 0` , hence `_computeDripAmount()` will return zero. However, since the entirety of `duration` has passed and no distributions have been made it would be preferable to return `amountToDistribute` .

The undesirable edge case will arise when we have the following conditions:

- `timeStarted + lastDistributeTime + updateFrequency < duration` AND
- `drip()` is not called until after `timeStarted + duration` .

The result is a proportion of tokens will not be distributed and will remain unaccounted for in the contract.

Recommendations

This issue may be mitigated by distributing the `rewardsLeftToDistribute` if `timeLeft == 0` which can be seen as follows.

```
if (stakingParams.distributedRewards >= stakingParams.amountToDistribute) {
    return 0;
}

uint256 timeElapsed = _timeSinceStart(stakingParams);
uint256 timeLeft = stakingParams.duration - timeElapsed;
uint256 rewardsLeftToDistribute = stakingParams.amountToDistribute
    - stakingParams.distributedRewards;
if (timeLeft == 0) {
    return rewardsLeftToDistribute;
}
```

Resolution

This has been resolved in commit 4cd59ce. The function `_computeDripAmount()` now returns `rewardsLeftToDistribute` if the mentioned edge case occurs.

AGL-08	Revoked `StableMaster` May be Deployed A Second Time		
Asset	`Core.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `Core` smart contract is able to deploy `StableMaster` contracts which represent a new stable coin. The `Core` also has the ability to `revokeStableMaster()` which essentially disowns a stablecoin.

When a `StableMaster` contract is revoked it is possible to add the contract back again by calling `deployStableMaster()`. This will call `StableMaster.deploy()` adding the current guardian and governor roles, without removing the previous ones.

Recommendations

Consider adding a state variable in `Core` which stores a `mapping(address=>boolean)` as to whether an address has been deployed as a `StableMaster` or not yet.

Resolution

In commit 4cd59ce a state variable was added to `Core`, that is, `deployedStableMasterMap`. The variable enables to check whether a `StableMaster` has been deployed previously or not. The check is conducted on line [114] in function `deployStableMaster()`.

```
require(!deployedStableMasterMap[stableMaster], "stableMaster previously deployed");
```

AGL-09	Potential Overflows in `BondingCurve`		
Asset	`BondingCurve.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The operation `totalTokensToSell**power` appears in the functions `_computePriceFromQuantity()` and `getCurrentPrice()`.

The expected decimal places of `totalTokensToSell` is 18 and the initial total supply is $1,000,000,000$ (1 billion). As `totalTokensToSell` is a `uint256` this gives us $2^{256} - 1$ as the max value which is about 78 decimal digits. Since the maximum in initial value of `totalTokensToSell` is 10^{27}, if $power \geq 3$ we could have a value of 10^{81} which would overflow a `uint256`.

Note in `getCurrentPrice()` we have `startPrice * (totalTokensToSell**power)` where start price is also in 18 decimal digits further increasing the likelihood of an overflow.

Recommendations

Since `power` must be greater than one (1) due to other requirements and less than three (3) to prevent overflow consider using a constant or hard coding the value to two (2).

If the value is hard coded to two (2) it may also be beneficial to change the order of operations to do some division operations before all of the multiplications. However, this will have the trade off in a reduction in precision due to rounding errors.

Resolution

The issue has been resolved in commit f0b86ff. The variable `power` was replaced by a hardcoded value of two (2).

AGL-10	`setGuardian()` May Overwrite Itself		
Asset	`Core.sol` , `PoolManager.sol` & `StakingRewards.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The function `setGuardian()` in `Core` is used to recursively update the Guardian role for all contracts in the protocol.

The function takes a parameter `_guardian` to be the new Guardian. It then performs the operations `grantRole()` for the new Guardian and `revokeRole()` for the previous Guardian in that order.

This poses a potential issue if both the new Guardian and old Guardian are the same address. That is because it will grant the permissions then revoke them leaving no users with Guardian permissions.

The same issue is present in `PoolManager.setGuardian()` which does `_addGuardian(_guardian)` then `_revokeGuardian(guardian)` .

A similar issue also exists in `StakingRewards.setNewRewardsDistributor()` . Indeed, if the `newRewardsDistributor` is the same as `rewardsDistribution` then the `REWARD_DISTRIBUTOR_ROLE` will end up being revoked for this address and there will be no addresses with `REWARD_DISTRIBUTOR_ROLE` permission.

Recommendations

We recommend adding a check to ensure the new Guardian is different to the previous Guardian (or new rewards distributor is different to the previous rewards distributor) and perform the revoke operations before the grant operations.

Furthermore, consider updating the name `_guardian` to `_newGuardian` for better clarity.

Resolution

This issue has been addressed in PR #89. An additional check was added to ensure that the `newGuardian` is not the same as the current guardian. A similar change was also made in `RewardsDistributor.setNewRewardsDistributor()` , where the new `RewardsDistributor` contract should not be identical to the current one.

Additionally, the order of operations was reversed in `setCore()` and `setGuardian()` to perform the **revoke** operations before the **grant** operations.

AGL-11	Circumvention of Maintenance Margin		
Asset	`PerpetualManagerFront.sol`		
Status	**Resolved:** Closed Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The maintenance margin exists so users can be liquidated if their position falls below a certain percentage of the perpetual's inital bought amount.

The maintenance margin is measured as $maintenanceMargin = \frac{position}{cashOutAmount}$ in the function `_getCashOutAmount()`.

`cashOutAmount` is initially the amount transferred to the pool as collateral for the swap. However, `perpetual.cashOutAmount` is updated when users call `addToPerpetual()`.

If a user has made a loss and is nearing the maintenance margin, they are able to call `addToPerpetual(1)` where 1 represents the smallest unit of value for the token, which will likely have negligible real value. This will update `cashOutAmount` to the total value of the new position, as a result the fraction $\frac{position}{cashOutAmount} = 100\%$ since `cashOutAmount == position`. This will prevent liquidation which will occur if the maintenance margin falls below 0.3%.

The impact of this is that it is possible for a user to continue to stay above the maintenance margin so long as their position is greater than zero. It is worth noting that the leverage, $leverage = \frac{boughtAmount}{position}$ will continue to increase by repeating this process. Since the leverage is increasing if the price continues to decrease the position will inevitably become negative and may be liquidated.

Note in addition to the above it is possible to call `forceCashOutPerpetual()` when the leverage reaches `cashOutLeverage` which is initially 100.

Recommendations

Ensure the development team is aware of this scenario and understands how it may be mitigated.

Resolution

A high level redesign has modified the functions `addToPerpetual()` and `removeFromPerpetual()` such that this issue is no longer relevant.

AGL-12	`_computeDripAmount()` is Non Linear in its Distribution		
Asset	`RewardsDistributor.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The function `drip()` adheres to the following formula,

$$dripAmount_i = updateFrequency * \frac{amountToDistribute - distributedRewards_{i-1}}{duration - min(timestamp_i - timeStarted,\ duration)}$$

There are two issues with the formula. First, since it can be expected that,

$$updateFrequency < timestamp_i - lastDistributionTime$$

as `drip()` transactions will not be mined exactly every $updateFrequency$ seconds. Thus, the amount of time for this distribution ($timestamp_i - lastDistributionTime$) is likely more than what is accounted for ($updateFrequency$), effectively under valuing $dripAmount$.

The second issue is that `timeLeft` uses the current timestamp whereas we use the previous `distributedRewards`.

Consider the following example with the inital setup,

- $timeStarted = 0$
- $timestamp_0 = 0$
- $duration = 10$
- $updateFrequency = 1$
- $amountToDistribute = 30$
- $distributedRewards_0 = 0$

Say we are at $timestamp_1 = 1$,

- $dripAmount_1 = 1 * \frac{30-0}{10-min(1-0,\ 10)} = \frac{30}{9} = 3.33$

However, for a linear equation we would expect to have,

- $dripAmount_1 = \frac{amountToDistribute}{duration} * timeElapsed = \frac{30}{10} * 1 = 3$

Thus, the $dripAmount$ from `_computeDripAmount()` is slightly over valued compared to a linear distribution.

Recommendations

To obtain a linear rewards distribution consider updating the `_computeDripAmount()` function to use the formula,

- $dripAmount_i = amountToDistribute * \frac{timestamp_i - lastDistributionTime_{i-1}}{duration}$

Note here we would need to update `lastDistributeTime` after executing `_computeDripAmount()` and there should be an additional check to handle the case where the elapsed time is greater than duration as follows.

```
if (rewardsLeftToDistribute < dripAmount) {
    dripAmount = rewardsLeftToDistribute;
}
```

Resolution

Function `_computeDripAmount()` has been updated in commit 4cd59ce to use the formula specified in the recommendation section, thereby ensuring linearity of the distribution.

AGL-13	Potential Accumulation of Interests After Calling `signalLoss()`		
Asset	`PoolManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

When a `PoolManager` has made a loss, the `Strategy` will call the `report()` function to signal this loss.

The order of operations in `PoolManager.report()` is to first call `StableMaster.signalLoss()` before calling `StableMaster.accumulateInterest()`. Noting that it is not expected that a `Strategy` will report both a loss and a gain at the same time, thus this edge case is unlikely to arise.

After a loss is reported, no more gains should be accumulated to the `SanRate`. However, due to the order of operations in `PoolManager.report()` if both a loss and a gain are reported at the same time, the gain will be accumulated to the `SanRate` after the loss is reported, thereby distributing the interest to all of the `SanToken` holders.

Recommendations

A solution to this issue is to reverse the order of operations in `PoolManager.report()`, thereby first accumulating the gains before reporting the losses.

Resolution

The order of operations was switched in PR #56 to do `accumulateInterest()` before `signalLoss()`.

AGL-14	Integer Overflow in `revokeStableMaster()`
Asset	`Core.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `revokeStableMaster()` function is used to revoke a `StableMaster` contract from the `Core` contract, and therefore removes the related stablecoin from `stablecoinList`. This function assumes there is always at least one stablecoin in `stablecoinList`, as shown in line [89].

```
for (uint256 i = 0; i < stablecoinList.length - 1; i++) {
```

If a Governor tries to execute this function where there is no existing stablecoin, there will be a negative integer overflow, caused by `stablecoinList.length - 1` which equals to `0 - 1` in an `uint256` variable. Fortunately, starting from Solidity 0.8.0, integer overflows are mitigated. However, from a user's perspective, the error message is unclear.

Recommendations

Consider adding an extra check inside the `revokeStableMaster()` function to make sure there is at least one stablecoin in `stablecoinList`.

```
if(stableCoinList.length > 0)
  for (uint256 i = 0; i < stablecoinList.length - 1; i++) {
```

Resolution

This has been resolved in commit 4cd59ce. The new implementation introduces an additional check to ensure that there exists at least one item on the list before revoking. Related code from line [134-136]:

```
uint256 stablecoinListLength = stablecoinList.length;
// Checking if ‘stableMaster‘ is correct and removing the stablecoin from the ‘stablecoinList‘
require(stablecoinListLength >= 1, "incorrect stablecoin");
```

AGL-15	`Governor` Must Be a Contract
Asset	`Core.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `Core` contract that has a strong control over `StableMaster` contracts is managed by **Governors**, which are expected to be instances of the `Governor` contract. Specifically, `governors` are authorised to call the following functions:

- `deployStableMaster()`
- `revokeStableMaster()`
- `addGovernor()`
- `removeGovernor()`

While the above functions are supposed to be called from authorised instances of the `Governor` contract, governors could technically be Externally Owned Accounts (EOAs).

This would significantly increase the likelihood of a compromise as a related private key could be leaked or retrieved by malicious actors, endangering the security of the `Core` contract, and ultimately the overall Angle protocol.

Recommendations

Consider ensuring that assigned governors are actual smart contracts. For example, The OpenZeppelin Address contract provides a function that implements this check (`isContract()`).

Resolution

The development team acknowledges the risk and will not implement any mitigation strategy at the moment. During the deployment phase, the Governor role is given to an EOA.

AGL-16	Suboptimal Search Iteration
Asset	`Core.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Contract `Core` utilises two address lists, namely `stablecoinList` and `governorList` to store the list of stablecoins and governors, respectively. To track the existence of a stablecoin or a governor in the mentioned lists, the contract iterates through the lists to get the item index.

For example, line [65-69] in function `deployStableMaster()` indicates an iterative process of finding whether a `StableMaster` contract of an `AgToken` contract has been deployed. This iteration can be expensive in terms of gas usage if `stablecoinList` has a large membership.

Similar type of iteration in the contract can be found in the following functions:

- `deployStableMaster()`, line [65-69]
- `addGovernor()`, line [115-117]
- `setGuardian()`, line [167-169]

Consider the following snippet from line [64-70]:

```
uint256 indexMet = 0;
for (uint256 i = 0; i < stablecoinList.length; i++) {
  if (stablecoinList[i] == stableMaster) {
    indexMet = 1;
  }
}
require(indexMet == 0, "stableMaster already deployed");
```

The `for` iteration in the snippet above always conducts search from index 0 to `stablecoinList.length`, which is suboptimal.

Recommendations

The testing team recommends inserting a `break` after the search data is found.

```
uint256 indexMet = 0;
uint256 listLength = stablecoinList.length;
for (uint256 i = 0; i < listLength; i++) {
  if (stablecoinList[i] == stableMaster) {
    indexMet = 1;
    break;
  }
}
require(indexMet == 0, "stableMaster already deployed");
```

Resolution

This has been resolved in commit 4cd59ce. Some search functions within the project were optimised by utilising `mapping` to access the intended item instantly without iteration. `break` was also introduced to applicable codes to stop an iteration quickly after a condition is met.

AGL-17	Suboptimal Delete Iteration
Asset	contracts/
Status	**Resolved:** See Resolution
Rating	Informational

Description

The contracts have several list item deletion operations, which can be found in the following functions:

- `Core.revokeStableMaster()`, line [89-96]
- `Core.removeGovernor()`, line [138-145]
- `PoolManager.revokeStrategy()`, line [301-328]
- `StableMaster.revokeCollateral()`, line [328-369]
- `RewardsDistributor.removeStakingContract()`, line [154-173]

Consider the following snippet from `Core.sol` line [88-98].

```
uint256 indexMet = 0;
for (uint256 i = 0; i < stablecoinList.length - 1; i++) {
 if (stablecoinList[i] == stableMaster) {
  indexMet = 1;
 }
 if (indexMet == 1) {
  stablecoinList[i] = stablecoinList[i + 1];
 }
}
require(indexMet == 1 || stablecoinList[stablecoinList.length - 1] ==
 stableMaster, "incorrect stablecoin");
stablecoinList.pop();
```

The `for` iteration in the snippet above always conducts search from index 0 to `stablecoinList.length`; if an item is found, then the next items are shifted to the left in order not to leave a gap in the list. This operation uses an excessive amount of gas due to numerous `SLOAD` operations.

Recommendations

The testing team recommends swapping the deleted item with the last item if item order is not important.

Also consider a gas optimisation where the length of the list (`listLength`) is cached to reduce the number of `SLOAD` instructions, which can be applied to all array iterations, not just deletions.

```
uint256 indexMet = 0;
uint256 listLength = stablecoinList.length;
for (uint256 i = 0; i < listLength - 1; i++) {
    if (stablecoinList[i] == stableMaster) {
        stablecoinList[i] = stablecoinList[listLength - 1];
        stablecoinList.pop();
        indexMet = 1;
        break;
    }
}
require(indexMet == 1, "incorrect stablecoin");
```

Resolution

This has been resolved in commit 2848add. The deleted item is now swapped with the last item, because the development team considers item order as not important. The list length is now cached whenever possible to reduce `SLOAD` opcode usage.

AGL-18	Duplicate Oracle Allowed on `setOracle()`
Asset	`StableMaster.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The function `setOracle()` in `StableMaster` sets a new oracle address which is forwarded to the associated collateral's `PoolManager` and `PerpetualManager`.

The function does not check whether the input `_oracle` is the same as the current oracle in the collateral.

Recommendations

The testing team recommends adding a check similar to the following snippet:

```
require(col.oracle != _oracle, "identical oracle");
```

Resolution

The recommended check has been added in commit 2848add, thereby preventing accidental issue.

AGL-19	Identical `newFeeManager` & `oldFeeManager` Allowed in `setFeeManager()`
Asset	`StableMaster.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `setFeeManager()` function enables a caller with `GUARDIAN_ROLE` to modify the `FeeManager` address of a specific collateral from `oldFeeManager` to `newFeeManager`. The changes affect the `contractMap` variable and the related `PoolManager` contract through `poolManager.setFeeManager()`.

The function `setFeeManager()` in `StableMaster` contract does not check whether the `newFeeManager` is identical to the previous value, `oldFeeManager`.

Recommendations

The testing team recommends adding the following extra check:

```
require(newFeeManager != oldFeeManager, "identical fee manager");
```

Resolution

The recommended check has been added in commit 2848add, thereby preventing accidental issue.

AGL-20	Integer Overflow on Empty List in `onlyCompatibleInputArrays`
Asset	`FunctionUtils.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The modifier `onlyCompatibleInputArrays` verifies whether two lists of `uint256` satisfy certain conditions, namely:

- both lists should be of the same length,
- the numbers inside the list are ascending.

When both lists are empty, it passes the length check on line [57]. Then, an integer overflow error occurs. This is caused by the iteration on line [58].

```
for (uint256 i = 0; i <= yArray.length - 1; i++) {
```

When `yArray.length` is zero, then `yArray.length - 1` overflows the `uint256`.

Note: from Solidity 0.8.0, integer overflows result in `reverts`.

Recommendations

The testing team recommends rejecting empty lists as inputs. For example, by adding an extra check using the following snippet:

```
require(xArray.length > 0, "empty array");
```

Resolution

The recommended check was added in commit 2848add preventing negative integer overflow.

AGL-21	Integer Overflow in `removeStakingContract()`
Asset	`RewardsDistributor.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Function `removeStakingContract()` removes a staking contract from `RewardsDistributor` contract. If this function is called while no staking contract is set, then a negative integer overflow occurs. This is because there is no check on the length of `stakingContractsList` before the following iteration on line [156].

```
for (uint256 i = 0; i < stakingContractsList.length - 1; i++)
```

If there is no existing staking contract on `stakingContractsList`, then the function reverts.

Note: from Solidity 0.8.0, integer overflows result in `reverts`.

Recommendations

An extra check can be introduced within the function to make sure there is at least one staking contract in `stakingContractsList`.

```
if(stakingContractsList.length > 0)
  for (uint256 i = 0; i < stakingContractsList.length - 1; i++) {
```

Resolution

The recommended check was added in commit 2848add preventing negative integer overflow.

AGL-22	Calling `updateHA()` inside `setHAFees()`
Asset	`FeeManager.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The functions `setHAFees()` and `updateHA()` in `FeeManager` contract are used to manipulate the keeper fees. `setHAFees()` changes the values of the `haFeeDeposit` and `haFeeWithdraw` variables. On the other hand, `updateHA()` sends these values to `PerpetualManager` (through a call to the function `PerpetualManager.setFeeKeeper()`).

The current implementation requires two separate calls to `setHAFees()` (by guardian role) and `updateHA()` (by a keeper).

Recommendations

Consider calling `updateHA()` inside `setHAFees()` to ensure the fee rates are immediately updated.

Resolution

The development team have decided not to implement this recommendation as it will not be compatible with improvements planned for `updateHA()`.

AGL-23	Revert When No `Strategy`
Asset	`PoolManager.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The functions `addGovernor()`, `removeGovernor()`, `setGuardian()`, and `revokeGuardian()` respectively invoke the functions `_addGuardian()` and `_revokeGuardian()` from the `PoolManagerInternal` contract. The last two functions mentioned assume there exists at least one `Strategy` contract in `strategyList`. This condition is not always necessarily true. If no `Strategy` has been added, the functions to add and remove (or revoke) governors and guardians in the `PoolManager` contract will revert.

This is caused by the following loop block from line [40-42] (`PoolManagerInternal.sol`):

```
for (uint256 i = 0; i < strategyList.length; i++) {
  IStrategy(strategyList[i]).addGuardian(_guardian);
 }
```

Additionally, the loop block from L51-53 (`PoolManagerInternal.sol`):

```
for (uint256 i = 0; i < strategyList.length; i++) {
  IStrategy(strategyList[i]).revokeGuardian(guardian);
 }
```

Recommendations

Make sure this behaviour is intended. The testing team recommends adding a conditional check to ensure at least one strategy has been added before stepping inside a loop block. See the following snippets for examples:

- `addGuardian()`:

```
uint256 listLength = strategyList.length;
require(listLength > 0)
for (uint256 i = 0; i < strategyList.length; i++) {
 IStrategy(strategyList[i]).addGuardian(_guardian);
}
```

- `revokeGuardian()` :

```
uint256 listLength = strategyList.length;
require(listLength > 0)
for (uint256 i = 0; i < listLength; i++) {
 IStrategy(strategyList[i]).revokeGuardian(guardian);
}
```

Resolution

These lists may be iterated when the length is zero and thus this is not considered an issue by the development team.

AGL-24	Emit Then Update on `addStrategy()`
Asset	`PoolManager.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The function `addStrategy()` adds a new strategy to the `PoolManager` contract and also updates the global parameters `debtRatio`. In this function, the event `StrategyAdded` is emitted in line [289], while the `debtRatio` variable is updated in line [292]. As a result, the event emits an old value of `debtRatio` and not the newest one.

Recommendations

Consider updating the `addStrategy()` function such that it emits the `StrategyAdded` after the `debtRatio` variable update.

```
// Update global parameters
debtRatio += _debtRatio;

emit StrategyAdded(strategy, debtRatio);
```

Resolution

The resolution of this issue was to update `debtRatio` before emitting the event. This can be seen in PR #54.

AGL-25	`Core` Does not Need to be `Initializable`
Asset	`Core.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The contract `Core` inherits `Initializable`. This is an unnecessary inheritance since `Core` will receive all of its parameters in the constructor and is not upgradeable via a proxy.

Recommendations

The inheritance `Initializable` may be safely removed.

Resolution

The inheritance of `Initializable` was removed in commit f0b86ff.

AGL-26	Initialisation of Proxy Implementations
Asset	contracts/
Status	**Resolved:** See Resolution
Rating	Informational

Description

When using an upgradeable proxy over an implementation it is important to ensure the underyling implementation is initialised in addition to the proxied contract.

Consider the following example where we have a `TransparentUpgradeableProxy` as `contractA` and a `StableMaster` as `contractB`. As a parameter to the constructor of `contractA` the data is provided to make a ***delegate call*** to `contractB.initialize()` which updates the storage as required in `contractA`.

Thus, `contractB` has not been initialized, only `contractA` has had its state updated. As a result any user is able to call `initialize()` on `contractB`, giving themselves full permissions over the contract.

There are two reasons why it is not desirable for malicious users to have full control over a contract:

1. Any delegate calls to external contracts can call `selfdestruct` which would delete the implementation contract temporarily making the proxy unusable (until it can be updated with a new implementation);
2. Scammers and malicious users may use contracts that are verified on `etherscan.io` and other block explorers.

Recommendations

Ensure that the `initialize()` function is called on all underlying implementations during deployment.

Resolution

Updates have been made to the deployment scripts to mitigate this issue by initialising the underlying contracts.

AGL-27	Purchase of Zero Tokens from `BondingCurve`
Asset	`BondingCurve.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Tokens can be bought from the `BondingCurve` contract through the function `buySoldTokens()`, which exchanges a stablecoin token for another ERC20 token.

The contract allows users to pass zero (0) as the `targetSoldTokenQuantity` to the function. This will likely later fail the condition `require(amountToPayInAgToken > 0)` since the cost of zero tokens should also be zero.

Recommendations

Consider adding an additional check to prevent users from attempting to buy zero tokens.

Resolution

This issue has been resolved in commit 2848add by adding a check to ensure `targetSoldTokenQuantity` is greater than zero.

AGL-28	TWAP Period May be Set to Zero
Asset	`UniswapUtils.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The function `changeTwapPeriod(uint32 _twapPeriod)` sets the **T**ime **W**eighted **A**verage **P**rice (**TWAP**) period for an Uniswap oracle.

It is possible to set `_twapPeriod` to zero using this function. This would make the function `_readUniswapPool()` unuseable due to the `assert` statement on line [42].

Recommendations

We recommend adding a check to ensure that `int32(twapPeriod) > 0` in `changeTwapPeriod()` thereby preventing both potential integer overflows (when casting from a `uint32`) and the zero case described above.

Consider also updating the `assert` on line [42] to a `require` statement.

Resolution

This issue was mitigated in commit 2848add by ensuring `twapPeriod` is non-zero and will not overflow on casting in `changeTwapPeriod()` , and removing the check from `_readUniswapPool()` .

AGL-29	Additional Constructor Checks
Asset	contracts/
Status	**Resolved:** See Resolution
Rating	Informational

Description

Additional checks may be added to the constructors in various contracts to prevent accidental misconfiguration during deployment.

These checks include:

- `ModuleUniswapMulti.sol`
  - `guardians.length > 0`
  - `_circuitUniswap.length == _circuitUniIsMultiplied`
  - sanity checks for `observationLength`
- `ModuleChainlinkMulti.sol`
  - `circuitChainIsMultiplied` is either 0 or 1
- `OracleMulti.sol`
  - `_uniFinalCurrency` is either 0 or 1
- `StakingRewards.sol`
  - `_rewardsToken == IRewardsDistributor(_rewardsDistribution).rewardToken()`

Recommendations

Consider adding some or all of these checks to the relevant constructor.

Resolution

The development team has decided to add the following checks:

- `ModuleUniswapMulti.sol`
  - `guardians.length > 0`
  - `_circuitUniswap.length == _circuitUniIsMultiplied`

The checks `ModuleChainlinkMulti.circuitChainIsMultiplied` and `OracleMulti_uniFinalCurrency` is either 0 or 1 were not implemented as 1 is considered `true` and all other values `false`.

Verification of `StakingRewards._rewardsToken` is done in `RewardsDistributor.sol` rather than `StakingRewards.sol` to improve deployment and testing.

AGL-30	Event `ReferenceCoinChanged` is Unused
Asset	`BondingCurve.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The event `ReferenceCoinChanged` in `BondingCurveEvents` is never emitted and may be safely removed.

Recommendations

Consider removing the event `ReferenceCoinChanged` from the `BondingCurveEvents.sol` contract.

Resolution

The recommendation has been implemented by safely removing the unused event.

AGL-31	Front-Runnable Functions
Asset	contracts/
Status	**Closed:** See Resolution
Rating	Informational

Description

A number of functions have the potential to be front-run to the gain of malicious users. Front-running attacks [3, 4] involve users watching the Blockchain mempool for particular transactions and, upon observing such a transaction, submitting their own transactions with a greater gas price. This incentivises miners to prioritise the later transaction.

The function `StableMaster.signalLoss()` can be front-run by users calling `StableMasterFront.withdraw()` as an SLP. The benefit to the attacker is that they are not distributed the losses which are shared over all `SanToken` holders. Similarly, a user may want to call `StableMaster.deposit()` if there is a large amount of locked interests or fees that are due to be accumulated to the `SanTokens`.

Users can submit transactions before or after fee changes (and potentially call `update*` functions themselves) to gain the best fees. As such the functions `updateUsersSLP()` and `updateHA()` are front-runnable.

Price feeds are updated via transactions from the oracle, users are able to front-run these transactions to gain more favourable prices.

Each of the keeper functions, such as `PerpetualManager.liquidatePerpetual()` and `PerpetualManager.forceCashOutPerpetual()`, reward the first message sender in the form of fees can also be front-run.

Similarly, the function `RewardsDistributor.drip()` can cause competition with only one winner taking the incentive.

Recommendations

Ensure all possible front-running vectors are understood, accounted for in the protocol, and potentially documented.

Resolution

The development team have acknowledged potential risks of front running and are preparing documents for users. From the protocol's perspective, the keepers rewards are only distributed once and it is irrelevant if they are front-run by other users. With respect to `signalLoss()`, it should be very rare (if ever) that this function is called and thus it is unlikely to be a targeted by front-running.

AGL-32	Draft OpenZeppelin Dependencies
Asset	`AgTokenEvents.sol` & `SanToken.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

Both of `AgToken` and `SanToken` inherit `ERC20PermitUpgradeable`, an OpenZeppelin contract. This contract is still a draft and is not considered ready for mainnet use. OpenZeppelin contracts may be considered draft contracts if they have not received adequate security auditing or are liable to change with future development.

Recommendations

Ensure the development team is aware of the risks of using a draft contract or consider waiting until the contract is finalised.

Resolution

The development team are aware of the risks of using a draft OpenZeppelin contract and have accepted the risk-benefit trade-off.

AGL-33	Reduce `SLOAD` Instructions when Reading Storage
Asset	contracts/
Status	**Resolved:** See Resolution
Rating	Informational

Description

Loading structs to memory is gas expensive as it will load every variable from storage, into memory requiring many expensive `SLOAD` instructions.

Consider using storage or loading just the variables needed for the following cases:

- `StableMaster.getCollateralRatio()`, line [145],

  `Collateral memory collat = collateralMap[managerList[i]];`

  may be replaced with:

  `Collateral storage collat = collateralMap[managerList[i]];`

- `PerpetualManagerFront.removeFromPerpetual()`, line [219]

  `Perpetual memory perpetual = perpetualData[perpetualID];`

  may be replaced with:

  `Perpetual storage perpetual = perpetualData[perpetualID];`

- `PerpetualManagerInternal._liquidatePerpetual()`, line [22]

  `Perpetual memory perpetual = perpetualData[perpetualID];`

  may be replaced with the lines:

  `uint256 committedAmount = perpetualData[perpetualID].committedAmount;`

  `uint256 cashOutAmount = perpetualData[perpetualID].cashOutAmount;`

- `PerpetualManagerInternal._cashOutPerpetual()`, line [48]

  `Perpetual memory perpetual = perpetualData[perpetualID];`

  may be replaced with the lines:

  `uint256 committedAmount = perpetualData[perpetualID].committedAmount;`

  `uint256 oldCashOutAmount = perpetualData[perpetualID].cashOutAmount;`

Recommendations

Ensure the comments are understood and consider implementing the gas optimisations.

Resolution

The first recommendation to update `StableMaster.getCollateralRatio()` was implemented. However, the remaining recommendations were no longer applicable due to design alterations which updated the code base.

AGL-34	Miscellaneous `AngleProtocol` General Comments
Asset	`contracts/*`
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Confusing function name `deploy()` in `AgToken.sol` and `StableMaster.sol`**

   The function `deploy()` is used to grant role `GUARDIAN_ROLE` to a list of governors (in `governorList`) and to `guardian`. The function names may confuse readers because of its similarities with contract deployment. Furthermore, the name does not properly reflect the purpose of the function.

   The testing team recommends renaming this function (e.g. `assignGuardianRole()`).

2. **Unclear revert message in `Core.sol`**

   On line [134], consider changing the revert message `"only one governor"` with `"must have at least one governor"`.

3. **Inconsistent Variable Naming in `PerpetualManager.sol`**

   - Inconsistent variable naming between `feeDeposit` and `feesWithdraw`. Should be consistent whether to use `fee` or `fees`.
   - Inconsistent variable naming between `rewardDistribution` and `rewardDistributor` in function `setNewRewardsDistributor()`.
   - Inconsistent variable naming between `_rewardsDistribution` on `setRewardDistribution()` and `newRewardsDistributor` on `setNewRewardsDistributor()`
   - `secureBlocks` and `perpetual.creationBlock` use `block` as a part of their names, although they store time information, based on the following snippet from `PerpetualManagerInternal.sol` line [54].

     ```
     require(perpetual.creationBlock + secureBlocks <= block.timestamp,
         "invalid timestamp");
     ```

     The testing team recommends to use "timestamp" instead of "block"

4. **Typo in `PerpetualManager.sol`** line [282] `FeeManagerù` -> `FeeManager`

5. **Variable `uint256` does not have negative value in `PerpetualManagerFront.sol`**

   Function `removeFromPerpetual()` handles negative `uint256` in line [216] when the zero case is sufficient.

   ```
   if (cashOutAmount <= 0) {
   ```

6. **Lack of Information for Owner in `forceCashOutPerpetual()` in `PerpetualManagerFront.sol`**

   The function `forceCashOutPerpetual()` enables a keeper to force cashing out a perpetual identified by an ID. Upon cashing out, there is a possibility that the perpetual owner receives an amount of coins being returned from the contract. There are two ways the system refund the perpetual owner, that is by transferring in collateral tokens or converted into `SLP` token (`sanToken`) if the contract does not have enough collateral token balance.

   The testing team recommends emitting an event to let the perpetual owner know how they are refunded by the contract.

7. **No Event for `approve()` and `setApprovalForAll()` in `PerpetualManagerFront.sol`**

   The functions `approve()` and `setApprovalForAll()` authorise a third-party to act on behalf of the perpetual owner. These functions modify the contract state but the transactions do not return any feedback to the caller. Only when the caller calls the functions `getApproved()` and `setApprovalForAll()` respectively, that they know the transactions have been successful.

   The testing team recommends introducing new events into both functions.

8. **Variable naming consistency: `rewardsDistribution` or `rewardsDistributor` in `StakingRewards.sol`**

   Both variables seem to refer to `RewardsDistributor` contract. Using a consistent term could be clearer.

9. **Typos in `PoolManager.sol`**

   - line [`126`]: yous -> you
   - line [`149, 163`]: transfered -> transferred

10. **Wrong variable is emitted in event `StrategyReported` in `PoolManager.sol`**

    The event `StrategyReported` specifies `address indexed strategy` as the first variable. However, line [`348`] in `PoolManager.sol` emits `msg.sender` , which should be `strategy` . Since this function can only be called by a guardian or a governor with `GUARDIAN_ROLE` , then a `Strategy` contract cannot call this function.

11. **Unfinished sentence in `OracleMulti.sol`**

    There is an unfinished sentence in line [`121`], *"... in case of"*

12. **Keyword `immutable` for unchangeable parameter**

    Keyword `immutable` can be used to ensure that a variable that holds contract parameter cannot be changed beyond constructor. For example, contract `OracleMulti` has `uniFinalCurrency` and `outBase` . The parameters are set only in the constructor, and hence can be treated as `immutable` .

13. **Potential naming confusion in `PoolManager.sol`**

    `PoolManager.StrategyParams.totalDebt` is easily confused with `PoolManager.totalDebt` , consider changing one or both names.

14. **Inconsistent naming in `OracleChainlinkSingle.sol`**

    `_chainIsMultiplied` and `isChainMultiplied` are both used to represent the same values, consider changing one or both names.

15. **Potentially confusing comments in `OracleMulti.sol`**

    The comments *"The current uni rate is in 'outBase' we want our rate to all be in base"* and *"The current amount is in 'inBase' we want our rate to all be in base"* on line [86] and line [88] respectively, the phrase *"in base"* may easily be confused with `inBase` .

16. **Repeated comments in `Core.sol`**

    The comments *`"/// @dev The ‘StableMaster‘ is initialized with the correct references"`* on line [56] are essentially repeated on the following line.

17. **Functions in `StableMaster.sol` under the wrong header**

    The functions `accumulateInterest()` and `signalLoss()` appear under the header *"CONSTRUCTORS AND DEPLOYERS"* when they are SLP functions.

18. **Use of the word "test" in production variable names**

    Consider renaming the function `_testMaxCAmount()` in `PerpetualManagerInternal.sol` to use *"check"* or *"verify"* instead of *"test"*.

19. **Inaccurate comments in `OracleMath.sol`**

    The comments *"(token1/token0) * decimals(token1)"* should be *"(token1/token0) * base(token1)"*.

20. **Simplification of equation in `OracleMath.sol`**

    line [74] `rate = ((price * 1e18) / (1 « 59));` can be simplified to `rate = ((price * 1e18) » 59));` as `a / (1 « x) = a » x`.

21. **Multiple fetches to `_poolManager.stableMaster()` in `CollateralSettlerERC20.sol` constructor**

    Multiple calls to `_poolManager.stableMaster()` may be cached in local variables to save gas. Additionally, consider removing the parameter `stableMaster`, and instead fetching it from `_poolManager`, in the constructor for mainnet deployment.

22. **No incentive to call `setAmountToRedistributeEach()` in `CollateralSettlerERC20.sol`**

    The function `setAmountToRedistributeEach()` must be called to allocate the payable amount based on the submitted claims by users, HA, and SLP. This function is callable by anyone. However, the caller does not receive incentive although the function call spends averagely 86,884 gas.

23. **Unnecessary import in `OracleMath.sol`**

    The import `import "@uniswap/v3-core/contracts/interfaces/IUniswapV3Pool.sol"` is not required and may be safely deleted.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

These findings have been acknowledged by the development team and actioned where appropriate.

The current actions are as follows:

1. Access control logic in `AgToken` contract was removed and therefore it is no longer applicable.
2. The revert message was corrected.
3. - `feesWithdraw` was replaced with `feeWithdraw`
   - `rewardDistributor` was replaced with `rewardDistribution`

- `newRewardsDistributor` was replaced with `_rewardsDistribution`
- The function was removed, so it is no longer applicable.

4. Fixed.
5. The cash out amount is now compared to 0 and the check was moved to `checkLiquidation()`.
6. Acknowledged and no changes were made. The owner can check through `Transfer` event.
7. Events were added.
8. `rewardsDistribution` is used instead of `rewardsDistributor`.
9. Fixed.
10. Event `strategyReported` on function `withdrawFromStrategy()` now emits `strategy`.
11. Fixed.
12. The keyword `immutable` was added whenever possible on non-upgradeable contracts.
13. `StrategyParams.totalDebt` was replaced with `StrategyParams.totalStrategyDebt`.
14. Fixed the related variable on `OracleChainlinkSingle` and `OracleChainlinkMulti`.
15. Fixed.
16. Fixed.
17. Fixed.
18. The function was removed so it is no longer relevant.
19. Fixed.
20. Recommendation applied.
21. Recommendation applied.
22. The development team assumes that the users will be motivated to call the function to receive collateral assets; or otherwise, the development team will call it. Alternatively, there can be an off-chain reward mechanism.
23. Recommendation applied.

AGL-35	Reentrancy When Closing Perpetuals		
Asset	`PerpetualManagerFront.sol`		
Status	**Open**		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The function `forceClosePerpetuals()` can be run when the quantity of funds being hedged by hedging agents has breached the `limitHAHedge`. This case occurs when enough users burn stablecoins, which may result in the hedging agents over compensating for the remaining stablecoins.

When a perpetual is force closed there is an external call through `_secureTransfer()` on line [315] which will call `transferFrom()` on the collateral token. The `forceClosePerpetuals()` continues to perform calculations after the external call based off a combination of variables stored in state and memory, thus opening up a potential reentrancy vector.

Exploiting this reentrancy vector allows an attacker to bypass the `estimatedCost` and `keeperFeesClosingCap` limits. These limits prevent users from earning more in fees than the cost of performing a flashloan attack. The reenterancy attack can be perform by reentering `forceClosePerpetuals()` multiple times which creates parallel executions of cashing out the perpetuals. These parallel executions split the amount of `cashOutFees` over each execution without changing the value for `estimatedCost`, thus avoiding the limits to the fees earned.

For this reentrancy to be viable the collateral ERC20 token must allow the attacker to gain control of execution during `transferFrom()`. Most ERC20 tokens do not relinquish control of execution to a user, however some do. One example is the ERC777 extension, which performs an execution call to the `to` address alerting the user the funds have been transferred. An attacker could use this call to gain control of the execution and reenter `forceClosePerpetuals()`.

Recommendations

We recommend performing all external calls after all calculations have been performed in accordance with the Checks-Effects-Interactions pattern.

This can be achieved by storing the `(owner, netCashOutAmount)` variables in an array and iterate through this array calling `_secureTransfer()` after all calculations have been performed.

AGL-36	Unnecessary Update of `lastUpdateTime`
Asset	`PerpetualManager.sol`
Status	Open
Rating	Informational

Description

The variable `lastUpdateTime` in `notifyRewardAmount()` is written to twice without being read inbetween.

The variable is written to first on line [80] then again on line [99].

```
lastUpdateTime = _lastTimeRewardApplicable();

if (block.timestamp >= periodFinish) {
    // If the period is not done, then the reward rate changes
    rewardRate = reward / rewardsDuration;
} else {
    uint256 remaining = periodFinish - block.timestamp;
    uint256 leftover = remaining * rewardRate;
    // If the period is not over, we compute the reward left and increase reward duration
    rewardRate = (reward + leftover) / rewardsDuration;
}
// Ensuring the provided reward amount is not more than the balance in the contract.
// This keeps the reward rate in the right range, preventing overflows due to
// very high values of ‘rewardRate‘ in the earned and ‘rewardsPerToken‘ functions;
// Reward + leftover must be less than 2^256 / 10^18 to avoid overflow.
uint256 balance = rewardToken.balanceOf(address(this));

require(rewardRate <= balance / rewardsDuration, "reward too high");

lastUpdateTime = block.timestamp;
```

Recommendations

line [80] `lastUpdateTime = _lastTimeRewardApplicable();` may be safely removed.

AGL-37	Reverts for Nonexistent Perpetuals in `liquidatePerpetuals()`
Asset	`PerpetualManagerFront.sol`
Status	Open
Rating	Informational

Description

Perpetuals may be liquidated by users other than the owner in the function `liquidatePerpetuals()` when the value falls below the `maintenanceMargin`. To save on gas perpetuals may be liquidated in batches by passing an array of IDs.

If one ID in the array `perpetualIDs` has already been liquidated (say by another user) the pereptual will be considered nonexistant and thus the transaction will revert, resulting in no liquidations.

Recommendations

Consider skipping any nonexistent perpetuals rather than reverting to reduce the likelihood of a liquidator's transaction reverting.

AGL-38	Gas Optimisation - Remove `onlyOwnerOrApproved` from `addToPerpetual()`
Asset	`PerpetualManagerFront.sol`
Status	Open
Rating	Informational

Description

The function `addToPerpetual()` allows a user to increase their `margin` by transferring additional funds to the perpetual.

This function has the modifier `onlyOwnerOrApproved` which ensures the sender of the transaction has the permissions required to add funds to this perpetual.

Recommendations

Since the function only allows adding funds to a perpetual, gas can be saved and bytecode size reduced by removing this modifier and letting any user add funds to a perpetual.

AGL-39	Miscellaneous `AngleProtocol` General Comments 2
Asset	`contracts/*`
Status	Open
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Constructor allows empty `governorList` in `RewardsDistributor.sol`.**

   A governor plays an important role in `RewardsDistributor`. There are five functions that can only be called by a governor. However, one can instantiate a `RewardsDistributor` contract without a governor by assigning an empty list to input variable `governorList` in the constructor.

   The testing team recommends adding a check to ensure non-empty `governorList`.

2. **Typo in the comments on `PerpetualManagerInternal.sol` on line** [200].

   The comment says "... the last timestep" which should say "timestamp".

3. **Incorrectly named event in `PerpetualManagerEvents.sol`.**

   The event `HAFeesUpdated` has fields `_xHAFeesDeposit` and `_yHAFeesDeposit` although the function `PerpetualManager.setHAFees()` allows to see the fees for both deposits and withdraws. Consider renaming these events too `xHAFees` and `yHAFees` to account for the withdraw case.

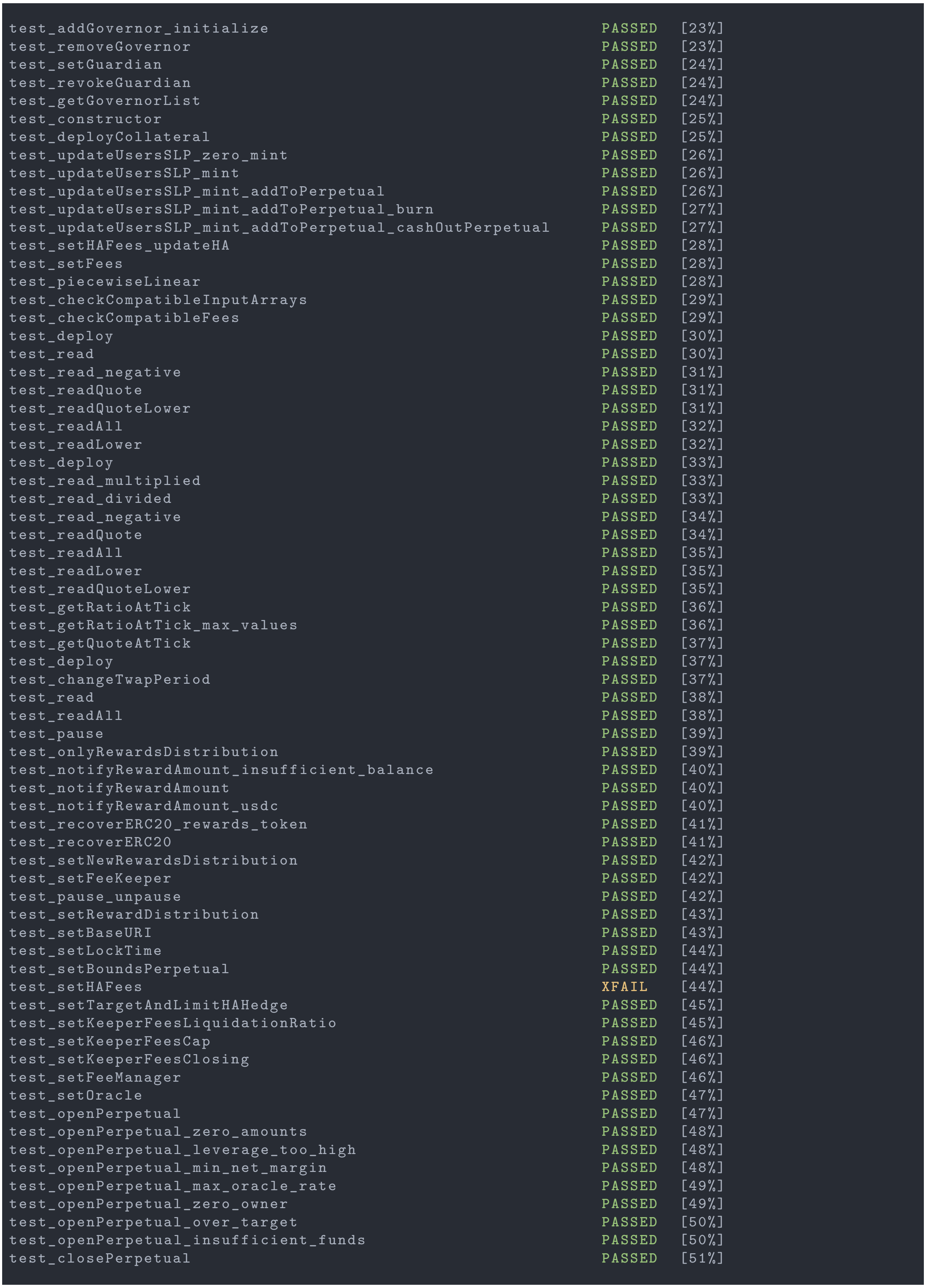
Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are provided alongside this document. The `brownie` framework was used to perform these tests and the output is given below.

```
test_constructor                                             PASSED   [0%]
test_setMinter_not_owner                                     PASSED   [0%]
test_setMinter_zero_address                                  PASSED   [1%]
test_setMinter                                               PASSED   [1%]
test_mint_not_minter                                         PASSED   [2%]
test_mint_exceed_minting_cap                                 PASSED   [2%]
test_mint                                                    PASSED   [2%]
test_mint_newToken                                           PASSED   [3%]
test_transfer_approve_transferFrom                           PASSED   [3%]
test_initialize_stableMaster_zero                            PASSED   [4%]
test_initialize                                              PASSED   [4%]
test_initialize_twice                                        PASSED   [4%]
test_onyRole                                                 PASSED   [5%]
test_mint                                                    PASSED   [5%]
test_burnNoRedeem                                            PASSED   [6%]
test_burnNoRedeem_insufficient_balance                       PASSED   [6%]
test_burnFromNoRedeem                                        PASSED   [6%]
test_burnFromNoRedeem_insufficient_balance                   PASSED   [7%]
test_burnSelf                                                PASSED   [7%]
test_burnFrom                                                PASSED   [8%]
test_transfer_approve_transferFrom                           PASSED   [8%]
test_deploy                                                  PASSED   [8%]
test_buySoldToken                                            PASSED   [9%]
test_buySoldToken_amount_zero                                PASSED   [9%]
test_buySoldToken_invalid_ag_token                           PASSED   [10%]
test_recoverERC20                                            PASSED   [10%]
test_allowNewStablecoin                                      PASSED   [11%]
test_allowNewStablecoin_invalid_reference                    PASSED   [11%]
test_allowNewStablecoin_zero_address                         PASSED   [11%]
test_allowNewStablecoin_two_references                       PASSED   [12%]
test_allowNewStablecoin_without_oracle                       PASSED   [12%]
test_changeOracle                                            PASSED   [13%]
test_changeOracle_zero_address                               PASSED   [13%]
test_revokeStablecoin                                        PASSED   [13%]
test_changeStartPrice                                        PASSED   [14%]
test_changeStartPrice_zero                                   PASSED   [14%]
test_changeTokensToSell                                      PASSED   [15%]
test_changeTokensToSell_insufficient_balance                 PASSED   [15%]
test_pausing                                                 PASSED   [15%]
test_constructor                                             PASSED   [16%]
test_triggerSettlement_zero                                  PASSED   [16%]
test_triggerSettlement                                       PASSED   [17%]
test_claimUser_agToken_govToken                              PASSED   [17%]
test_claimUser_govToken                                      PASSED   [17%]
test_claimHA                                                 PASSED   [18%]
test_claimSLP_sanToken_govToken                              PASSED   [18%]
test_claimSLP_sanToken                                       PASSED   [19%]
test_claimSLP_govToken                                       PASSED   [19%]
test_setAmountToRedistributeEach_noclaim                     PASSED   [20%]
test_setAmountToRedistributeEach_claimUser_agToken_govToken  PASSED   [20%]
test_setAmountToRedistribute                                 PASSED   [20%]
test_constructor                                             PASSED   [21%]
test_deployStableMaster                                      PASSED   [21%]
test_deployStableMaster_redeploy                             PASSED   [22%]
test_revokeStableMaster                                      PASSED   [22%]
test_addGovernor                                             PASSED   [22%]
```



```
test_addGovernor_initialize                                  PASSED   [23%]
test_removeGovernor                                          PASSED   [23%]
test_setGuardian                                             PASSED   [24%]
test_revokeGuardian                                          PASSED   [24%]
test_getGovernorList                                         PASSED   [24%]
test_constructor                                             PASSED   [25%]
test_deployCollateral                                        PASSED   [25%]
test_updateUsersSLP_zero_mint                                PASSED   [26%]
test_updateUsersSLP_mint                                     PASSED   [26%]
test_updateUsersSLP_mint_addToPerpetual                      PASSED   [26%]
test_updateUsersSLP_mint_addToPerpetual_burn                 PASSED   [27%]
test_updateUsersSLP_mint_addToPerpetual_cashOutPerpetual     PASSED   [27%]
test_setHAFees_updateHA                                      PASSED   [28%]
test_setFees                                                 PASSED   [28%]
test_piecewiseLinear                                         PASSED   [28%]
test_checkCompatibleInputArrays                              PASSED   [29%]
test_checkCompatibleFees                                     PASSED   [29%]
test_deploy                                                  PASSED   [30%]
test_read                                                    PASSED   [30%]
test_read_negative                                           PASSED   [31%]
test_readQuote                                               PASSED   [31%]
test_readQuoteLower                                          PASSED   [31%]
test_readAll                                                 PASSED   [32%]
test_readLower                                               PASSED   [32%]
test_deploy                                                  PASSED   [33%]
test_read_multiplied                                         PASSED   [33%]
test_read_divided                                            PASSED   [33%]
test_read_negative                                           PASSED   [34%]
test_readQuote                                               PASSED   [34%]
test_readAll                                                 PASSED   [35%]
test_readLower                                               PASSED   [35%]
test_readQuoteLower                                          PASSED   [35%]
test_getRatioAtTick                                          PASSED   [36%]
test_getRatioAtTick_max_values                               PASSED   [36%]
test_getQuoteAtTick                                          PASSED   [37%]
test_deploy                                                  PASSED   [37%]
test_changeTwapPeriod                                        PASSED   [37%]
test_read                                                    PASSED   [38%]
test_readAll                                                 PASSED   [38%]
test_pause                                                   PASSED   [39%]
test_onlyRewardsDistribution                                 PASSED   [39%]
test_notifyRewardAmount_insufficient_balance                 PASSED   [40%]
test_notifyRewardAmount                                      PASSED   [40%]
test_notifyRewardAmount_usdc                                 PASSED   [40%]
test_recoverERC20_rewards_token                              PASSED   [41%]
test_recoverERC20                                            PASSED   [41%]
test_setNewRewardsDistribution                               PASSED   [42%]
test_setFeeKeeper                                            PASSED   [42%]
test_pause_unpause                                           PASSED   [42%]
test_setRewardDistribution                                   PASSED   [43%]
test_setBaseURI                                              PASSED   [43%]
test_setLockTime                                             PASSED   [44%]
test_setBoundsPerpetual                                      PASSED   [44%]
test_setHAFees                                               XFAIL    [44%]
test_setTargetAndLimitHAHedge                                PASSED   [45%]
test_setKeeperFeesLiquidationRatio                           PASSED   [45%]
test_setKeeperFeesCap                                        PASSED   [46%]
test_setKeeperFeesClosing                                    PASSED   [46%]
test_setFeeManager                                           PASSED   [46%]
test_setOracle                                               PASSED   [47%]
test_openPerpetual                                           PASSED   [47%]
test_openPerpetual_zero_amounts                              PASSED   [48%]
test_openPerpetual_leverage_too_high                         PASSED   [48%]
test_openPerpetual_min_net_margin                            PASSED   [48%]
test_openPerpetual_max_oracle_rate                           PASSED   [49%]
test_openPerpetual_zero_owner                                PASSED   [49%]
test_openPerpetual_over_target                               PASSED   [50%]
test_openPerpetual_insufficient_funds                        PASSED   [50%]
test_closePerpetual                                          PASSED   [51%]
```

```
test_closePerpetual_price_decrease                            PASSED  [51%]
test_closePerpetual_negative_position                         PASSED  [51%]
test_closePerpetual_twice                                     PASSED  [52%]
test_closePerpetual_lock_time                                 PASSED  [52%]
test_closePerpetual_minCashOutAmount                          PASSED  [53%]
test_onlyApprovedOrOwner                                      PASSED  [53%]
test_addToPerpetual                                           PASSED  [53%]
test_addToPerpetual_negative_position                         PASSED  [54%]
test_removeFromPerpetual                                      PASSED  [54%]
test_removeFromPerpetual_amount_zero                          PASSED  [55%]
test_removeFromPerpetual_negative_position                    PASSED  [55%]
test_removeFromPerpetual_exceed_cash_out_amount               PASSED  [55%]
test_removeFromPerpetual_exceed_margin                        PASSED  [56%]
test_removeFromPerpetual_maxLeverage_margin                   PASSED  [56%]
test_removeFromPerpetual_maxLeverage_cashOutAmount            PASSED  [57%]
test_liquidatePerpetual                                       PASSED  [57%]
test_liquidatePerpetual_repeat_perpetual                      PASSED  [57%]
test_liquidatePerpetual_negative_position                     PASSED  [58%]
test_forceClosePerpetuals                                     PASSED  [58%]
test_forceClosePerpetuals_negative_position                   PASSED  [59%]
test_getCashOutAmount                                         PASSED  [59%]
test_earned                                                   PASSED  [60%]
test_getReward                                                PASSED  [60%]
test_ERC721_getters                                           PASSED  [60%]
test_ERC721                                                   PASSED  [61%]
test_initialize                                               PASSED  [61%]
test_deployCollateral                                         PASSED  [62%]
test_addGovernor_removeGovernor                               PASSED  [62%]
test_addGovernor_nostrategy                                   PASSED  [62%]
test_removeGovernor_nostrategy                                PASSED  [63%]
test_setGuardian_revokeGuardian                               PASSED  [63%]
test_setGuardian_notGuardian                                  PASSED  [64%]
test_setFeeManager                                            PASSED  [64%]
test_estimatedAPR_none                                        PASSED  [64%]
test_estimatedAPR_strategy_zero_APR                           PASSED  [65%]
test_updateStrategyDebtRatio                                  PASSED  [65%]
test_addStrategy_revokeStrategy                               PASSED  [66%]
test_withdrawFromStrategy_zeroAmount                          PASSED  [66%]
test_getBalance                                               PASSED  [66%]
test_getTotalAsset                                            PASSED  [67%]
test_creditAvailable                                          PASSED  [67%]
test_debtOutstanding_loan_not_taken                           PASSED  [68%]
test_debtOutstanding_loan_taken                               PASSED  [68%]
test_report_take_loan                                         PASSED  [68%]
test_report_gain                                              PASSED  [69%]
test_report_loss                                              PASSED  [69%]
test_report_payDebt                                           PASSED  [70%]
test_recoverERC20                                             PASSED  [70%]
test_recoverERC20_overdraw                                    PASSED  [71%]
test_setStrategyEmergencyExit                                 PASSED  [71%]
test_constructor                                              PASSED  [71%]
test_constructor_no_governor                                  PASSED  [72%]
test_setStakingContract                                       PASSED  [72%]
test_drip_not_initialized                                     PASSED  [73%]
test_drip_too_soon                                            PASSED  [73%]
test_drip                                                     PASSED  [73%]
test_governorWithdrawRewardToken                              PASSED  [74%]
test_governorWithdrawRewardToken_not_governor                 PASSED  [74%]
test_governorRecover                                          PASSED  [75%]
test_governorRecover_withdraw_rewards_token                   PASSED  [75%]
test_governorRecover_staked_tokens                            PASSED  [75%]
test_setNewRewardsDistributor                                 PASSED  [76%]
test_setNewRewardsDistributor_zero_address                    PASSED  [76%]
test_removeStakingContract_not_initialized                    PASSED  [77%]
test_removeStakingContract                                    PASSED  [77%]
test_setUpdateFrequency                                       PASSED  [77%]
test_setIncentiveAmount                                       PASSED  [78%]
test_setAmountToDistribute_not_initialized                    PASSED  [78%]
test_setAmountToDistribute                                    PASSED  [79%]
test_setDuration                                              PASSED  [79%]
```

```
test_initialize                                              PASSED  [80%]
test_mint                                                    PASSED  [80%]
test_onlyStableMaster                                        PASSED  [80%]
test_burnNoRedeem                                            PASSED  [81%]
test_burnSelf                                                PASSED  [81%]
test_burnFrom                                                PASSED  [82%]
test_burnFrom_insufficient_allowance                         PASSED  [82%]
test_transfer_approve_transferFrom                           PASSED  [82%]
test_deploy                                                  PASSED  [83%]
test_initialize                                              PASSED  [83%]
test_deploy_revoke_Collateral                                PASSED  [84%]
test_add_remove_Governor                                     PASSED  [84%]
test_set_revoke_Guardian                                     PASSED  [84%]
test_contractMapCheck                                        PASSED  [85%]
test_pause_unpause                                           PASSED  [85%]
test_pause_invalid_role                                      PASSED  [86%]
test_setOracle                                               PASSED  [86%]
test_setFeeManager                                           PASSED  [86%]
test_setUserFees                                             PASSED  [87%]
test_getCollateralRatio_zero_mints                           PASSED  [87%]
test_getCollateralRatio                                      PASSED  [88%]
test_getCollateralRatio_two_collaterals                      PASSED  [88%]
test_setCore                                                 PASSED  [88%]
test_mint_paused                                             PASSED  [89%]
test_mint                                                    PASSED  [89%]
test_burn                                                    PASSED  [90%]
test_burn_pause                                              PASSED  [90%]
test_deposit                                                 PASSED  [91%]
test_withdraw                                                PASSED  [91%]
test_withdraw_paused                                         PASSED  [91%]
test_constructor                                             PASSED  [92%]
test_stake                                                   PASSED  [92%]
test_stakeOnBehalf_zero_address                              PASSED  [93%]
test_stakeOnBehalf                                           PASSED  [93%]
test_withdraw_nostake                                        PASSED  [93%]
test_exit_nostake                                            PASSED  [94%]
test_withdraw_zero_amount                                    PASSED  [94%]
test_withdraw                                                PASSED  [95%]
test_exit                                                    PASSED  [95%]
test_notifyRewardAmount_no_rewards                           PASSED  [95%]
test_notifyRewardAmount                                      PASSED  [96%]
test_getReward                                               PASSED  [96%]
test_recoverERC20_staking_token                              PASSED  [97%]
test_recoverERC20_rewards_token                              PASSED  [97%]
test_recoverERC20                                            PASSED  [97%]
test_setNewRewardsDistribution                               PASSED  [98%]
test_setNewRewardsDistribution_zeroAddress                   PASSED  [98%]
test_setup_protocol                                          PASSED  [99%]
test_setup_BondingCurve                                      PASSED  [99%]
test_prerequisite_deployCollateral                           PASSED  [100%]
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

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

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].

[3] Sigma Prime. Solidity Security - Front Running. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html#race-conditions. [Accessed 2018].

[4] NCC Group. DASP - Front Running. Website, 2018, Available: http://www.dasp.co/#item-7. [Accessed 2018].

σ'