PUBLIC

Code Assessment

of the Angle Borrowing Module Smart Contracts

May 17, 2022

Produced for

Angle

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Angle Team,

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

Angle implements a new way to borrow Angle's stable token agEUR by using over-collateralized loans with liquidation mechanism.

The most critical issue uncovered in our audit is a call to an untrusted address. The amount of issues uncovered are usual for a project of this size. The documentation of the project is good and the communication with the team was very professional. All issues were fixed accordingly or (in case of some low severity issues) acknowledged.

In summary, we find that the codebase provides a good level of security. It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

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

Critical-Severity Findings	1
• Code Corrected	1
High-Severity Findings	0
Medium-Severity Findings	3
• Code Corrected	3
Low-Severity Findings	20
• Code Corrected	11
• Specification Changed	5
• Acknowledged	4

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the following source code files inside the Angle Borrowing Module repository based on the documentation files.

- `AgToken.sol`
- `CoreBorrow.sol`
- `FlashAngle.sol`
- `BaseOracleChainlinkMulti.sol`
- `OracleWSTETHEURChainlink.sol`
- `BaseReactor.sol`
- `BaseReactorStorage.sol`
- `Treasury.sol`
- `VaultManager.sol`
- `VaultManagerERC721.sol`
- `VaultManagerStorage.sol`

The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	04 Apr 2022	d522f132bb7a814a066bdcc9ad518c2934ba7be8	Initial Version
2	09 May 2022	11dd806015eef3c24de62fd56ea394e4f481e8e8	Version 2
3	16 May 2022	0363b6a137a44e22ee06b3187ba74f7798c1af08	Version 3

For the solidity smart contracts, the compiler version `0.8.12` was chosen.

2.1.1 Excluded from scope

All other files were excluded from scope and not audited. The economic model behind the protocol and the chosen economic parameters of the system are out of scope. Furthermore, the contract `Settlement` is out of scope, hence the shutdown process is excluded from scope of this review.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

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

Angle offers users to lend stable coins. To lend stable coins users need to deposit collateral tokens worth more than the stable coin loan (over-collateralized). In case the value of the collateral falls below a defined threshold the collateral is sold with a discount to liquidators against the stable coin. In case of under-collateralized loans (bad loans) an inbuilt insurance mechanism tries to cover the losses. The insurance funds build up by taking certain fees. The system has two additional components: (1) First, it offers users flash loans denominated in the stable coin and (2) it offers the ability to get a stable coin loan which is automatically invested into yield earning strategies. The funds are always managed by the system and not accessible by the investor. The system tries to balance the loan in such a way that it should not default.

The system charges a fee on minting, a stability fee on the loan as kind of interest and a liquidation surcharge in case of liquidations.

2.2.1 System Setup

The system has a centralized contract (`CoreBorrow`) to manage all access rights and permissions. There are three types of users. The first type uses the system to simply borrow agEUR tokens against a collateral. These investors will use the `VaultManager` as primary interaction contract. The second type might be an investor (e.g., a DAO) owning certain tokens that want to invest and earn a yield on their funds. These kind of investors will use the `reactor` contract which tries to manage their investment and balance the loan in a way that a liquidation is unlikely. The third kind of investors use the `FlashLoan` contract to borrow a flash loan in agEUR tokens.

There is one `VaultManager`, `Settlement` and `Reactor` contract per collateral type for a given stablecoin, one treasury overseeing the `VaultManager` contracts per stablecoin. One `FlashLoan` contract manages the flash loans of all different stable coins. Each system on a chain needs additionally oracle contracts and one `Swapper` contract as well as the respective stable coin.

The following contacts are upgradable `AgToken`, `CoreBorrow`, `FlashAngle`, `Reactor`, `Treasury`, `VaultManager`. To ensure a consistent access/permission model managed by the `CoreBorrow` contract, the admin privileges need to be kept in line with the upgradable contracts and the `CoreBorrow` contract.

2.2.2 Core borrow

The `CoreBorrow` contract is the contract responsible for handling the access control across all the borrowing modules and is read by all `Treasury` contracts belonging to a certain stable coin. The role definitions are built upon OpenZeppelin's `AccessControlEnumerable` contract. By default, there are a `GUARDIAN_ROLE`, `GOVERNOR_ROLE` and a `FLASHLOANER_TREASURY_ROLE` role managed by the contract. An emergency function to check consistent roles in a newly deployed core contract, the `setCore` function could be used to update the `FlashLoan` contract's core contract. Similarly, the `FlashLoan` contract can be update in the core contract via `setFlashLoanModule`

The contract is deployed with proxy contract.

2.2.3 Treasury

The treasury contract keeps the accounting for all vault managers and flash loan contracts given a stable coin. It aggregates the profits and losses from the vault managers. Each Treasury contract has the ability to grant minting rights on its associated AgToken: it is hence the contract granting permissions to each new `VaultManager`. A share of the profits can be transferred to a surplus manager contract. Most functions are accounting functions that rely on being called to keep the accounting up to date. These functions could be called by anyone to keep the accounting up to date and collect potential profits:

- `fetchSurplusFromAll` to collect profits from flash loan and vault manager contracts or account for losses.
- `fetchSurplusFromFlashLoan` to only collect from flash loan contracts or account for losses.
- `fetchSurplusFromVaultManagers` to only collect from vault managers or account for losses.

- `pushSurplus` transfer part of the profits to the ANGLE's surplus management contract.
- `updateBadDebt` burns agEUR to balance out losses (bad debt).

Additionally, the contract offers the following governor permissioned functions:

- `addMinter` and `removeMinter` to add or remove an account as minter to the stable coin (agEUR).
- `addVaultManager` and `removeVaultManager` to add or remove a vault manager from the treasury contract.
- `recoverERC20` to transfer any ERC20 tokens out of the contract.
- `setTreasury` to set a new treasury contract in all associated vault managers and the stable coin contract.
- `setSurplusForGovernance` sets the relative amount that can be withdrawn by the governance to the surplus manager address.
- `setSurplusManager` sets the address that will receive the surplus amount when withdrawn.
- `setCore` sets the core contract for access management.
- `setFlashLoanModule` sets the flash loan contract for this treasury.

2.2.4 Vault Manager

The vault manager creates new vaults by minting an NFT and storing the vault specific parameters. In Version 1 a fee is charged when borrowing, while in Version 2 fees can be charged when borrowing, repaying, or in both actions. Via the `angle` function, it offers the main functionality of the protocol for users taking out agEUR loans:

- creating a vault (minting the vault NFT).
- adding collateral or removing it as long as the vault remains healthy.
- approving the collateral to the VaultManager via `permit`.
- borrow and repay debt.
- transfer debt between two vaults via `getDebtIn`.
- liquidate vaults (for keepers that monitor the health status).

Each `VaultManager` contract implements the ERC-721 standard and, hence, has the corresponding functionality.

Furthermore, the contract implements some admin functionality:

- setter to set the liquidation booster parameters.
- toggle function to turn on or off if the receiver of a mint or transfer of a vault needs to be whitelisted.
- toggle the whitelisting of a user account (in case whitelisting would be needed).
- setter for the oracle contract to query the price information.
- setter to set the treasury contract overseeing the vault.
- toggle function to pause the functions: `angle`, `createVault` and `liquidate`.

The contract has minting right on its corresponding AgToken.

2.2.5 Reactor Contract

At the time of the audit, only one example of a reactor contract was present. The Euler reactor allows to borrow agEUR and manages the investment in Euler finance. It provides functionality on top of the vault manager. The contract tries to avoid liquidations as it is in full control of the borrowed stable coins and they are not accessible by the investing users. The user might access the returns generated or close the investment to get the collateral back. The contract offers the following main functionality:

- `deposit` and `mint` for a user to deposit collateral in the strategy.
- `withdraw` and `redeem` for a user to redeem collateral from the strategy.
- `claim` to `claim` stablecoin rewards accruing from the strategy.

2.2.6 Flash Loan Contract

The `FlashAngle` contract is the flash loan contract to flash loan Angle's stable token agEUR (and eventually others in the future). The module needs minting permissions for each token contract and checks after the flash loan that the borrowed tokens plus fees are transferred back and the borrowed amount is burned again.

The main function for this functionality is `flashLoan`. Additionally, the contract has the permissioned (Governor or Guardian) setter `setFlashLoanParameters` for the flash loan's parameters (maximum amount to be borrowed and flash loan fee), the function `accrueInterestToTreasury` to transfer the surplus to the treasury of a particular stable coin. The functions:

- `addStablecoinSupport` and `removeStablecoinSupport` allows the core borrow contract (set at initialization or via `setCore`) to add a treasury and stable coin to the flash loan contract and remove it. The function can only be called by the governor role from the core borrow contract.
- `setCore` sets a new core contract and is restricted to be called from the old core borrow contract. Only governance is allowed to call the function to trigger the change in the old core borrow contract.

2.2.7 The Stable Coin Token Contract

The token contract is an extended standard ERC20 token contract based on OpenZeppelin's `ERC20PermitUpgradeable`. It is mintable and burnable. The following additional functions support the individual system setup and extend the functionality.

- `burnNoRedeem`, `burnFromNoRedeem` and `burnStablecoin` are different ways to burn the stable tokens a user borrowed. Depending on which option is chosen the collateral is redeemed or remains as debt.
- `burnSelf`, `burnFrom` and `mint` are functions that can be called by addresses having the minuter role. The functions burn and mint stable tokens.
- `addMinter` allows the governor role to add a minter via the treasury contract.
- `removeMinter` can either be called by the governance via the treasury contract to remove a minter (except for the stable minter) or the minter themselves can call it to remove themselves.
- `setTreasury` allows to set a new treasury address for the stable coin. It can only be called by the governor via the old treasury contract.

2.2.8 Swapper Contract

The swapper contract `Swapper` allows to swap tokens by calling `swap`. The function `changeAllowance` lets the governor or guardian role set an amount of stable tokens to be spend by a spender.

2.2.9 Oracle Contracts

These contracts are used to query the price information from Chainlink. The Chainlink oracles are assumed to behave correctly and non-maliciously. The contract performs sanity checks for the freshness of prices reported by oracles and reverts if the price is outdated.

3 Limitations and use of report

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

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

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

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

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

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

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

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

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

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

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	4

5.1 Approximated Fee Charged in Debt Transfer

Design Low Version 1 Acknowledged

In function `vaultManager.getDebtOut`, a fee is charged according to the different `borrowFee` and `repayFee` between two vault managers. It will be an approximation which slightly round if both `borrowFee` and `repayFee` are enabled.

We assume the `borrowFee` and `repayFee` are `f1` and `r1` on vaults `A`. And `f1` and `r1` for `B` as well. We assume `f1<f2` and `r1>r2`. Then if a debt `X` is transferred from `A` to `B`, the following fee will be charged.

$$\delta fee = (f_2 - f_1) * X + \left(\frac{r_1}{1-r_1} - \frac{r_2}{1-r_2}\right) * X$$
$$\delta fee = \left[(f_2 - f_1) + \left(\frac{r_1 - r_2}{(1-r_1)(1-r_2)}\right)\right] * X >= (\delta f + \delta r) * X$$

where

$$\delta f = f_2 - f_1$$
$$\delta r = r_1 - r_2$$

This is slightly larger than the formula used in the project given that both `f` and `r` are small:

$$\delta fee = (\delta f + \delta r - \delta r * \delta f) * X$$

Acknowledged

Angle replied:

```
It is possible that it slightly rounds down, overall we do not expect to
have both fees taken up at the same time on the same vaultManager.
And as we're aware that it's an approximation, fees should be set accordingly.
```

5.2 Ignored Return Value of _repayDebt

Design Low Version 1 Acknowledged

The return value of the call to `_repayDebt` in the function `VaultManager.liquidate` is ignored, although it gives the correct amount of stable coins that need to be burned for the debt payment. Instead `amounts[i]` is used, as shown below:

```
if (vault.collateralAmount <= collateralReleased) {
    ...
} else {
    ...
    _repayDebt(
        vaultIDs[i],
        (amounts[i] * liquidationSurcharge) / BASE_PARAMS,
        liqData.newInterestAccumulator
    );
}
...
liqData.stablecoinAmountToReceive += amounts[i];
```

Acknowledged

Angle replied:

```
The repayDebt function rounds down the stablecoin amount in the case where it is
bigger than the total debt of the vault. In a liquidation setting, the amount in
repayDebt is: 'amounts[i]*liquidationSurcharge / BASE_PARAMS' where
'amounts[i]<=maxStablecoinAmountToRepay' and
'maxStablecoinAmountToRepay <= debt of the vault + 1'. As such, in the worse
scenario possible, the output value of the repayDebt function will very slightly
be rounded down from what should theoretically be taken: we should therefore view
this as a slightly higher fee taken by the protocol on the liquidation.
```

5.3 Possible Gas Optimization in Mappings

Design Low Version 1 Acknowledged

Several contracts of the system use mappings in the format: mapping(key_type => bool). Solidity uses a word (256 bits) for each stored value and performs some additional operations when operating bool values (due to masking). Therefore, using uint instead of bool is slightly more efficient. A list of such mappings:

- `isMinter` in `agToken`.
- `vaultManagerMap` in `Treasury`.
- `isWhitelisted` and `_operatorApprovals` in `VaultManagerStorage`.

Acknowledged:

The mappings `vaultManagerMap`, `isWhitelisted` and `_operatorApprovals` have been modified and now use `uint256` instead of `bool` as pointed out above.

Only the mapping `isMinter` remains unchanged because the Angle has already deployed a version of the contract.

5.4 Unchecked Collateral Amount

Design | Low | Version 1 | Acknowledged

In the contract `VaultManager`, when a user calls `_addCollateral` or `_removeCollateral`, no checks are performed on the collateral amount. Hence, a vault can have an amount of collateral which is below the `_dustCollateral` parameter.

Acknowledged

Angle replied:

```
There is no need to check for the `_dustCollateral` parameter when people are adding or removing
collateral from their vault. What is important is that people with a debt have an amount of collateral
in their vault which is higher than `_dustCollateral` and this can be for sure implemented if `_dust`
parameter is set accordingly with the `collateralFactor` parameter and the `_dustCollateral` parameter.

It is not a problem for the protocol if people decide to add collateral little by little or remove
their collateral little by little if they are no longer in debt or their debt is small.
```

6 Resolved Findings

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

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

Critical-Severity Findings	1

- Unchecked VaultManager Address Code Corrected

High-Severity Findings	0

Medium-Severity Findings	3

- Inconsistent Access Control Code Corrected
- Incorrect Accounting of Global Debt Code Corrected
- Stuck Ether Code Corrected

Low-Severity Findings	16

- Incomplete Specifications BaseReactor Code Corrected
- Mismatch of Specifications in _repayDebt Code Corrected
- Unclear Specifications for Swap Function Code Corrected
- Incomplete Specifications Specification Changed
- Inconsistent Error Message Code Corrected
- Misleading Function Name Code Corrected
- Mismatch of Specifications for Function _isSolvent Specification Changed
- Missing Description of Variable Decimals Specification Changed
- Missing Sanity Checks on Vault Creation Code Corrected
- No Event Emitted on Flashloan's Parameters Update Code Corrected
- Possible to Optimize Struct Code Corrected
- Precision Loss in Division Code Corrected
- Specification Mismatch in _handleRepay Specification Changed
- Specification Mismatch setUint64 Specification Changed
- Unchecked Array Length Code Corrected
- Unchecked VaultID When Adding Collateral Code Corrected

6.1 Unchecked VaultManager Address

Security Critical Version 1 Code Corrected

The `fetchSurplusFromVaultManagers` function is an external function of the contract `treasury/Treasury.sol`, as displayed below, there is no check towards the user input `VaultManager` address. An adversary can deploy a contract with a function `accrueInterestToTreasury` which can return arbitrary numbers to maliciously update the state variables `surplusBufferValue` and `badDebtValue`.

```
function fetchSurplusFromVaultManagers(address[] memory vaultManagers) external returns (uint256, uint256) {
    (uint256 surplusBufferValue, uint256 badDebtValue) = _fetchSurplusFromList(vaultManagers);
    return _updateSurplusAndBadDebt(surplusBufferValue, badDebtValue);
}

function _fetchSurplusFromList(address[] memory vaultManagers) internal returns (uint256 surplusBufferValue, uint256 badDebtValue) {
    badDebtValue = badDebt;
    surplusBufferValue = surplusBuffer;
    uint256 newSurplus;
    uint256 newBadDebt;
    for (uint256 i = 0; i < vaultManagers.length; i++) {
        (newSurplus, newBadDebt) = IVaultManager(vaultManagers[i]).accrueInterestToTreasury();
        surplusBufferValue += newSurplus;
        badDebtValue += newBadDebt;
    }
}
```

Code corrected:

The vulnerable function `fetchSurplusFromVaultManagers` has been removed from the updated code. Hence, the functionality to collect the surplus only from a subset of vault managers is not available anymore. In order to collect the surplus accrued by all `VaultManager` contracts, function `fetchSurplusFromAll` should be called.

6.2 Inconsistent Access Control

Security **Medium** **Version 1** **Code Corrected**

The setup of roles for the contract `CoreBorrow` is implemented in the function `initialize`. The admin of the `GUARDIAN_ROLE` is set to `GUARDIAN_ROLE`, which may lead to an exploit as a malicious guardian can remove all governors from the `GUARDIAN_ROLE`. In such scenario, the functions `addGovernor`, `isGovernorOrGuardian`, and all the functions in other contracts that call `isGovernorOrGuardian` with a governor address would revert, as they no longer have the `GUARDIAN_ROLE`, and thus are no longer the admin of the `GUARDIAN_ROLE`.

```
function initialize(address governor, address guardian) public initializer {
    require(governor != address(0) && guardian != address(0), "0");
    require(governor != guardian, "12");
    _setupRole(GOVERNOR_ROLE, governor);
    _setupRole(GUARDIAN_ROLE, guardian);
    _setupRole(GUARDIAN_ROLE, governor);
    _setRoleAdmin(GUARDIAN_ROLE, GUARDIAN_ROLE);
    _setRoleAdmin(FLASHLOANER_TREASURY_ROLE, GOVERNOR_ROLE);
}

function addGovernor(address governor) external {
    grantRole(GOVERNOR_ROLE, governor);
    grantRole(GUARDIAN_ROLE, governor);
}

function isGovernorOrGuardian(address admin) external view returns (bool) {
    return hasRole(GUARDIAN_ROLE, admin);
}
```

Code corrected:

The issue has been addressed in code **Version 2**, the governor is set as the admin of the `GUARDIAN_ROLE`, hence a guardian cannot change anymore the roles of a governor address.

Furthermore, the function `removeGovernor` has been updated to allow a governor address to remove its roles, i.e., revoke its roles as guardian and then as governor.

6.3 Incorrect Accounting of Global Debt

Correctness Medium Version 1 Code Corrected

The following issue was reported by Angle during the review process. The function `_closeVault` in the contract `VaultManager.sol` does not update the global debt state variable `totalNormalizedDebt`.

Code corrected:

The function `_closeVault` has been revised to update the global debt state when a vault is closed: `totalNormalizedDebt -= vault.normalizedDebt;`.

6.4 Stuck Ether

Security Medium Version 1 Code Corrected

The function `angle` in the contract `VaultManager` is declared as `payable`, however the code has no logic to deal with the incoming Ether. Therefore, the Ether sent when calling the function `angle` is not accounted and gets stuck into the contract.

Code corrected:

The keyword `payable` has been removed from the function `VaultManager.angle`, hence users cannot send Ether to the contract when calling this function.

6.5 Incomplete Specifications `BaseReactor`

Correctness Low Version 2 Code Corrected

The parameter `_protocolInterestShare` in `BaseReactor._initialize` is missing the NatSpec description.

Code corrected:

The NatSpec description has been added for the parameter `_protocolInterestShare` in the function `_initialize`.

6.6 Mismatch of Specifications in _repayDebt

Correctness Low Version 2 Code Corrected

The function `VaultManager._repayDebt` will not revert on a non-existing vault, however the NatSpec comments assume that it will revert.

```
/// @dev This function will revert if it's called on a vault that does not exist
```

Code corrected:

The sentence above has been removed from the NatSpec of the function `_repayDebt`.

6.7 Unclear Specifications for Swap Function

Correctness **Low** **Version 2** **Code Corrected**

The NatSpec descriptions for parameters in `ISwapper.swap` are confusing, for example the parameter `outTokenOwed` has the following description:

```
@param outTokenOwed Minimum amount of outToken this address should have at the end of the call
```

It is unclear if *this address* refers to the contract `Swap` or to the recipient address.

Code corrected:

The NatSpec description for the parameter `outTokenOwed` has been revised:

```
@param outTokenOwed Minimum amount of outToken the `outTokenRecipient` address should have at the end of the call.
```

6.8 Incomplete Specifications

Correctness **Low** **Version 1** **Specification Changed**

Several NatSpec descriptions for the function parameters are not complete. We provide a non-exhaustive list:

- The description of `` data`` in `VaultManager.liquidate`.
- `supply` in `BaseReactor._convertToShares`.
- Return values in `BaseReactor._getFutureDebtAndCF`.

Specification changed:

The NatSpec descirptions have been added for the examples listed above:

```
/// @param data Data to pass to the repayment contract in case of...
/// @param _supply Optional value of the total supply of the reactor, it is recomputed if zero
/// @return futureStablecoinsInVault Future amount of stablecoins borrowed in the vault
/// @return collateralFactor Collateral factor of the vault if its debt remains unchanged but `toWithdraw` collateral
```

6.9 Inconsistent Error Message

Design **Low** **Version 1** **Code Corrected**

The error codes in `modifier onlyTreasury` in `BaseAgTokenSideChain.sol` and `function setTreasury` in `BaseOracleChainlinkMulti.sol` are inconsistent with the respective descriptions in `errorMessages.json`.

Furthermore, most of the contracts use numbers as error messages, and the file errorMessages.json maps each error code to a meaningful description. However, in `BaseReactor` the following messages are used:

```
require((assets = _convertToAssets(shares, usedAssets + looseAssets, 0)) != 0, "ZERO_ASSETS");
require(currentAllowance >= shares, "ERC20: transfer amount exceeds allowance");
```

Code corrected:

The error messages have been revised on the whole codebase and a new approach is used:

```
if(!condition) revert CustomErrorMessage();
```

6.10 Misleading Function Name

Correctness **Low** **Version 1** **Code Corrected**

In contract `EulerReactor.sol`, the function name `_maxStablecoinsAvailable` does not match with the specifications and the code, which returns the maximum amount of assets that can be withdrawn.

Code corrected:

The function `_maxStablecoinsAvailable` has been renamed to `_maxAmountWithdrawable`, and the NatSpec description has been updated accordingly:

```
@return maxAmount Max amount of assets that can be withdrawn from the reactor
 considering Euler liquidity for the stablecoin.
```

6.11 Mismatch of Specifications for Function _isSolvent

Correctness **Low** **Version 1** **Specification Changed**

The NatSpec comment for the function `VaultManager._isSolvent` states:

```
/// @notice Verifies whether a given vault is solvent (i.e., should be liquidated or not)
...
/// @dev If the oracle value or the interest rate accumulator has not been called at the time of the
/// call, this function computes it
```

The first sentence above states that the function `verifies` if the vault is solvent, however the function does not verify the vault status, but only computes some parameters.

The second sentence above states that the function computes the interest rate accumulator if it has not been called before, however the implementation does not perform it.

Specification changed:

The NatSpec descriptions have been revised to reflect the behavior of the function implementation:

```
/// @notice Computes the health factor of a given vault. This can later be used to check whether a given vault is solvent
```

6.12 Missing Description of Variable Decimals

Correctness Low Version 1 Specification Changed

The documentation pages state that the codebase generally uses three bases: `BASE_TOKENS` (18 decimals), `BASE_PARAMS` (9 decimals) and `BASE_INTEREST` (27 decimals). However, to improve readability and integrations with other systems, the code would benefit from having a description of the expected base for each variable.

Specification changed:

The NatSpec descirption for `VaultManagerStorage.BASE_PARAMS` states that unless specified otherwise all the parameters are in 9 decimals:

```
/// @notice Base used for parameter computation: almost all the parameters of this contract are set in `BASE_PARAMS`
```

6.13 Missing Sanity Checks on Vault Creation

Design Low Version 1 Code Corrected

The function `VaultManager.angle` does not perform any sanity check on vault creation for the parameter `to`, which is the owner of the vault.

Code corrected:

The sanity check to prevent vaults being minted to `address(0)` has been added into the function `VaultManagerERC721._mint`:

```
if (to == address(0)) revert ZeroAddress();
```

6.14 No Event Emitted on Flashloan's Parameters Update

Design Low Version 1 Code Corrected

The function `FlashAngle.setFlashLoanParameters` updates the fee and the maximum amount that can be borrowed by the module; however, no event is emitted.

Code corrected:

The following event will be triggered every time the function is successfully called.

```
event FlashLoanParametersUpdated(IAgToken indexed stablecoin, uint64 _flashLoanFee, uint256 _maxBorrowable);
...
emit FlashLoanParametersUpdated(stablecoin, _flashLoanFee, _maxBorrowable);
...
```

6.15 Possible to Optimize Struct

Design Low Version 1 Code Corrected

The struct `FlashAngle.StablecoinData` can be optimized to occupy 2 storage slots instead of 3 if reordered.

Code corrected:

The struct is reordered to occupy 2 storage slots.

```
struct StablecoinData {
    uint256 maxBorrowable;
    uint64 flashLoanFee;
    address treasury;
}
```

6.16 Precision Loss in Division

Design Low Version 1 Code Corrected

This line below from the function `_checkLiquidation` of contract `VaultManager.sol` uses division, which is prone to rounding errors. In this case it is possible to use multiplication as needed to have both sides of the comparison operator in the same decimals instead of using division.

```
if (currentDebt <= (maxAmountToRepay * surcharge) / BASE_PARAMS + dust)
```

Code corrected:

The updated code avoids the division operator and evaluates the condition as follows:

```
if (currentDebt * BASE_PARAMS <= maxAmountToRepay * surcharge + dust * BASE_PARAMS) {
    ...
}
```

6.17 Specification Mismatch in _handleRepay

Correctness Low Version 1 Specification Changed

The NatSpec description for the parameter `to` in `VaultManager._handleRepay` states:

```
@param to Address to which stablecoins should be sent
```

However, the function only sends collateral tokens to the address `to`:

```
if (collateralAmountToGive > 0)
    collateral.safeTransfer(to, collateralAmountToGive);
```

Specification changed:

The NatSpec comments has been updated:

```
@param to Address to which collateral should be sent
```

6.18 Specification Mismatch setUint64

Correctness **Low** **Version 1** **Specification Changed**

The function `setUint64` in the contract `VaultManager.sol` is protected with the modifier `onlyGovernorOrGuardian`, however, in the specification, it says *When setting parameters governance should make sure ...*. The Angle team should assess and clarify the intended behaviour and update the specification or the modifier accordingly.

Specification changed:

The specification is changed to comply with the modifier:

```
/// @dev When setting parameters governance or the guardian should make sure that...
```

6.19 Unchecked Array Length

Design **Low** **Version 1** **Code Corrected**

The function `angle` in the contract `VaultManager` does not check if the input arrays `actions` and `datas` have the same length and trigger an early revert if the input parameters do not match, thus be more gas efficient.

Code corrected:

The updated code performs a check that arrays `actions` and `datas` have the same length. Furthermore, it also checks that the arrays have a non-zero length:

```
if (actions.length != datas.length || actions.length == 0)
    revert IncompatibleLengths();
```

6.20 Unchecked VaultID When Adding Collateral

Design Low Version 1 Code Corrected

The function `angle` does not perform any check to verify if a vault exists when the action is `addCollateral`. The internal function `_addCollateral` also does not perform such checks, hence it is possible to add collateral to vaults that are not created yet, or to vaults that have been burned, i.e., locking tokens.

Code corrected:

The function `_addCollateral` has been updated to check if the collateral is being added into an existing vault:

```
if (!_exists(vaultID)) revert NonexistentVault();
```

7 Notes

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

7.1 Default VaultID Value

Note Version 1

The function `angle` in the contract `vaultManager/VaultManager.sol` will use the latest `vaultID` if the action's parameter `vaultID` is `0`. Users should be aware of this default behavior and be careful to use `vaultID = 0` only when the first action of a batch operations is `createVault`.

7.2 Dependency on Freshness of Chainlink Oracle Prices

Note Version 1

The Angle Borrowing Module queries Chainlink oracles to get the price of an asset and the function `_readChainlinkFeed` performs the following sanity checks:

```
(uint80 roundId, int256 ratio, , uint256 updatedAt, uint80 answeredInRound)
    = feed.latestRoundData();
if (ratio <= 0
    || roundId > answeredInRound
    || block.timestamp - updatedAt > stalePeriod)
        revert InvalidChainlinkRate();
```

If the price is carried over from an old round (`answeredInRound < roundID`), or the price is outdated (`block.timestamp - updatedAt > stalePeriod`), then the function reverts. Therefore, actions that query oracles cannot be executed if the returned price do not pass the sanity checks, e.g., `closeVault`, `removeCollateral`, `borrow`, `getDebtIn`, `liquidate`. This might become problematic for the system if Chainlink oracles stop working at any point in future for collateral assets.

7.3 Event MinterToggled Can Be Emitted Multiple Times

Note Version 1

In contract `AgToken.sol`, functions `addMinter` and `removeMinter` will always emit a `MinterToggled` event without checking if the minter has already been added or removed.

7.4 Inconsistency Between Debt and Issued Stable Coins

Note Version 1

When a user wants to transfer debt from VaultManager `B` to VaultManager `A`, the function `angle` in the contract `VaultManager.sol` does not check if VaultManager `B` is a valid VaultManager. If a user deploys a contract with the VaultManager interface and set its address as `B`, then the debt of the user increases without issuing any stable coins.

7.5 Repay Fee Calculation

Note **Version 2**

The repay fee in the function `VaultManager.angle` is calculated with the following code:

```
uint256 stablecoinAmountPlusRepayFee = (stablecoinAmount * BASE_PARAMS) / (BASE_PARAMS - repayFee);
```

If the user wants to repay `100 USDC` when the repay fee is 3%, the formula above will calculate `103.0927835052` USDC as the total amount needed to be repaid.

7.6 System Inconsistency

Note **Version 1**

In contract `AgToken.sol`, functions `burnNoRedeem` and `burnFromNoRedeem` burn the stable tokens and interact with `IStableMaster` which is not part of the borrowing module reviewed in this audit. Users of the borrowing module have no incentive to call these functions.

7.7 Unbounded Loops in Treasury Contract

Note **Version 2**

Functions `setTreasury`, `_fetchSurplusFromList` and `removeVaultManager` loop through the array `vaultManagerList`. However, there are no bounds on the size of the array, which means there is a possibility that the transaction exceeds the block gas limit. In those cases, the transaction will revert. Hence, the governance should ensure that the number of entries in `vaultManagerList` is limited so the transaction cost remains under the block gas limit.