Aave

Aave Protocol v2.0

Smart Contract Security Assessment

Version: 5.0

June, 2023

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Aave v2 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 contract. 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 Aave v2 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 Aave v2 smart contracts.

Overview

Aave is a platform that permits users to lend and borrow tokens (and Ether) on the Ethereum blockchain. The contracts reviewed in this report constitute version 2 of the Aave protocol. The version 2 contracts provide a number of improvements and extra features to their version 1 predecessors. Some of the core updates include:

- **Debt Tokenization -** Users who borrow funds are now attributed a debt token to allow users to more easily manage their debt position.
- **Collateral Transfers -** Users can trade their deposited assets and interest accruing Aave tokens. This allows users to exit their position and swap their deposited assets.
- **Flash Loan updates -** Users can now create flash loans of multiple assets in a single transaction and use flash loans to liquidate undercollateralised loans.
- **Structure Upgrade -** The core structure of the underlying smart contracts have been redesigned which reduce the gas usage and makes it easier to test and verify the functioning of core components.

Security Assessment Summary

This review was conducted on the files hosted on the protocol-v2 repository and were assed at commit 16e67c0.

Note: the OpenZeppelin libraries and dependencies were excluded from the scope of this assessment.

The design of the Aave protocol is such that an administrator account exists that has the ability to update and replace existing contracts and in essence modify core components and functionality of the protocol. As this is by design, this review does not explicitly list attacks where the administrator is malicious, rather we assume the Aave protocol maintainer keys are secure and honest. Should the administration keys be stolen by a malicious actor, the protocol can be severely compromised.

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 [**?**, **?**].

The testing team identified a total of eleven (11) issues during this first assessment, of which:

- One (1) is classified as high risk,
- One (1) is classified as low risk,
- Nine (9) are classified as informational.

All these issues have been acknowledged and/or resolved by the development team.

Additionally, the testing team reviewed the changes related to optimisations in new market deployments. These changes are introduced in this pull request. No issues were identified by the testing team in this second assessment.

Furthermore, the testing team reviewed a proposal to implement a `PermissionedLendingPool`. These changes can be see in this merge request. One (1) issue was identified and classified as low risk.

An additional review was undertaken over changes to the intialisation and upgrading of the `AToken`, `StableDebtToken` and `VariableDebtToken` contracts, in conjuction with added functionality in `handleRepayments()` and finally a refactoring of `calculateInterestRates()`. These changes can be seen in this diff. Three (3) informational issues were identified by the testing team during this review (**AAV-13** to **AAV-15**).

Finally, a subsequent review was conducted targetting the following Pull Requests:

- PR # 95
- PR # 108
- PR # 140
- PR # 143
- PR # 165
- PR # 167
- PR # 178
- PR # 180
- PR # 181
- PR # 184
- PR # 186
- PR # 187
- PR # 193
- PR # 197

During this round of testing, eleven (11) issues were identified (**AAV-16** to **AAV-26**):

- Two (2) are classified as medium risk,
- Three (3) are classified as low risk,
- Six (6) are classified as informational.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Aave v2 smart contracts. 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 comments not directly related to the security posture of the protocol, 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
AAV-01	Ineffective Check in `validateBorrow()`	High	Resolved
AAV-02	Available Liquidity Incorrectly Calculated in `flashLoan()`	Low	Resolved
AAV-03	Issues when Stable Borrow Rate is Zero	Informational	Closed
AAV-04	Debt Allowances are Front-runnable	Informational	Closed
AAV-05	Lack of Validation for the Zero Address	Informational	Closed
AAV-06	Inconsistent Event and Interface Naming	Informational	Closed
AAV-07	Unused Variables in `ValidationLogic`	Informational	Resolved
AAV-08	Gas Optimisation in `_calculateBalanceIncrease`	Informational	Closed
AAV-09	Contracts Do Not Implement Safe Ownership Transfer Pattern	Informational	Closed
AAV-10	Delegate Call to Force a Self-Destruct	Informational	Resolved
AAV-11	Miscellaneous General Statements	Informational	Closed
AAV-12	Mint `ATokens` to Unpermissioned User	Low	Open
AAV-13	Import of Debugging Tool `Hardhat`	Informational	Resolved
AAV-14	Initialization `params` are Shared Between Multiple Contracts	Informational	Open
AAV-15	Potential Overminting During Configurator Updates	Low	Open
AAV-16	Circumvention of Checks in `validateBorrow()`	Medium	Open
AAV-17	Limitations of Exposure Ceiling	Medium	Resolved
AAV-18	Ineffective `!isContract()` Check	Low	Resolved
AAV-19	Incorrect Value Emitted for the Event `ReserveDataUpdated` in `flashLoan()`	Low	Open
AAV-20	Potential Misconfiguration of Flash Loan Premiums	Low	Resolved
AAV-21	`isRiskAdmin()` Modifier Unnecessarily Restrictive	Informational	Open
AAV-22	Further Unused Variables	Informational	Open
AAV-23	Unused Errors	Informational	Open
AAV-24	Removal of Deprecated Functions	Informational	Resolved
AAV-25	Gas Optimisation - Reduce Storage Loads	Informational	Open

AAV-26 Miscellaneous General Statements - 2 Informational Open

AAV-01	Ineffective Check in `validateBorrow()`		
Asset	`ValidationLogic.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The function `validateBorrow()` in `ValidationLogic.sol` is used to ensure that a user is allowed to perform a borrow. It verifies conditions such as, if the user has sufficient collateral to make a borrow.

There is an ineffective control statement on line [200], `if (vars.rateMode == ReserveLogic.InterestRateMode.STABLE)`. This statement is ineffective as the variable `vars.rateMode` is never initialised and thus the check is equivalent to `if (0 == 1)` and so will never pass.

As a result the following requirements for stable borrowing are not validated:

- stable rate borrowing is enabled;
- the user has less collateral in the currency than borrow amount OR
- the borrow amount is less than 25% of the liquidity.

The impact is that users may make a stable borrow when it has not been enabled. If done on an asset that has not been configured to allow stable borrows then `LendingRateOracle.getMarketBorrowRate(asset)` would return zero. As a result, a user may borrow at a rate of 0%. See AAV-03 for further details on the impact of this vulnerability.

Users may also borrow a large percentage of the liquidity at a low rate and in the same currency as their collateral which would potentially allow them to earn more in revenue than they pay in fees by then depositing the borrowed funds. This is because both the stable and variable rate will be increased for future users.

Recommendations

We recommend using the variable `interestRateMode` instead of `vars.rateMode` in the afore mentioned control statement. `interestRateMode` is a parameter to the function which is initialised to the correct value, thereby correctly triggering the `if` statement.

Resolution

The contracts have been updated such that the control statement is now `if (interestRateMode == uint256(DataTypes.InterestRateMode.STABLE))` thereby activating the if statement when there is a stable borrow.

The contracts were immediately redeployed with the patch when the bug was discovered. Later the GitHub repository was updated to match the contacts as seen in commit 29448c1.

AAV-02	Available Liquidity Incorrectly Calculated in `flashLoan()`		
Asset	`LendingPool.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The function `flashLoan()` allows users to first acquire the funds and execute some logic, then either repay the amount of the flash loan plus a premium or convert it to a loan of either stable or variable interest.

When converting the flash loan into an interest bearing loan the interest rates are updated. The interest rates are based of the utilisation rate which is:

$$U_t = \frac{totaldebt}{totaldebt + availableliquidity}$$

There is an error in the calculation of `availableLiquidity` when the interest rate mode is `NONE`. It is calculated as `IERC20(reserveAddress).balanceOf(aTokenAddress)` plus `currentPremium`. This incorrectly calculates the available liquidity as the the amount of the flash loan has been transferred out and so is not included in `IERC20(reserveAddress).balanceOf(aTokenAddress)`.

As a result, the utilisation rate is overstated significantly. This will result in both the variable borrow rate and the stable borrow rate also being significantly overstated.

Recommendations

We recommend using `currentAmountPlusPremium` rather than `currentPremium` for the case where interest rate mode is `NONE`, to account for the modified balance of the aToken. This will ensure the available liquidity is correctly calculated.

Resolution

The mainnet contracts were deployed using `currentAmountPlusPremium` rather than `currentPremium` when calculating the available liquidity, thus resolving the issue.

The resolution was implemented in commit 584a567.

AAV-03	Issues when Stable Borrow Rate is Zero
Asset	`StableDebtToken.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

In the case of AAV-12 or if `LendingRateOracle.getMarketBorrowRate()` returns zero (which should only be possible by misconfiguration) then a user may obtain a stable rate of zero.

When deposits have been made but there are no borrows, the utilisation rate will be zero. Therefore the stable rate will be equal to `LendingRateOracle.getMarketBorrowRate()`. If `LendingRateOracle.getMarketBorrowRate()` is also zero then the user will obtain a stable rate of zero.

The first obvious implication of having a stable rate zero is that a user will pay zero fees for a loan.

Another implication is that a following call to `StableDebtToken.burn()` (e.g. if a user calls `repay()`) will cause the `StableDebtToken.totalSupply()` to be set to zero. This occurs since both `_avgStableRate` and `usersStableRate` will both be zero. Hence the `if` statement on line [185] will get triggered. This will cause `newStableRate = _avgStableRate = _totalSupply = 0;`, setting the total supply to zero.

This is an issue as the user may still have a significant balance, yet the total supply is set to zero. For example if a user has a balance of $10,000$ tokens and repays 1 token their remaining balance will be set to $9,999$ but the total supply will be set to zero.

Recommendations

Consider handling the case where both `_avgStableRate` and `_usersStableRate` are zero.

Resolution

It is an underlying assumption of the protocol that `LendingRateOracle.getMarketBorrowRate()` should never return zero for a reserve with stable rate borrowing enabled. It is prevented at configuration time, thus these issues will not be patched.

AAV-04	Debt Allowances are Front-runnable
Asset	`DebtTokenBase.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

Debt tokens (both stable and variable) allow users to delegate a borrow allowance to other addresses (users or contracts). This allowance can be used to make a borrow on behalf of the delegater.

The function `approveDelegation(address delegatee, uint256 amount)` is used to give another user an allowance. The allowance that the `delegatee` may use is set to `amount`. This function is vulnerable to double spending via front-running.

For example consider a situation where Alice has given Bob an allowance of 1 ETH. Alice would now like to reduce Bob's allowance to 0.5 ETH. Alice will then make the transaction `approveDelegation(Bob, 0.5 ETH)`.

Bob may abuse this mechanism by front-running Alice's transaction and use the current allowance of 1 ETH, leaving the remaining allowance as 0 ETH. Alice's transaction will then be executed setting Bob's allowance up to 0.5 ETH. Bob may now spend the 0.5 ETH allowance, thereby spending a total of 1.5 ETH when he should only have had at most 1 ETH to spend.

Note that this issue also affects the `approve()` function of the ERC20 standard.

Recommendations

Consider using functions such as `increaseDelegation()` and `decreaseDelegation()` which will take the current allowance and increment it or decrement it respectively, thereby preventing double spending.

Resolution

The development team have acknowledged this issue as an issue equivalent to the ERC20 `approve()` front-running issue and are therefore not implementing a fix.

AAV-05	Lack of Validation for the Zero Address
Asset	`LendingPool.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

Historically, some wallets and applications default to using the `0x0` address if an address parameter is omitted when interacting with smart contracts. This has lead to cases of inadvertent fund transfers to the `0x0` address.

One mitigation strategy (that requires a small amount of extra gas) is to verify that Ether and tokens are not transferred to the `0x0` address in the smart contract. This is done, for example, when minting ATokens in the `IncentivisedERC20` `_mint()` function.

In reference to Aave, users currently can inadvertently withdraw tokens to the `0x0` address (if the `to` parameter is set to `0x0`).

Recommendations

A check could be added to the `withdraw()` function to prevent accidental transfers to the `0x0` address.

Resolution

The recommendation has been marked as a potential improvement and will be considered through the governance procedures.

AAV-06	Inconsistent Event and Interface Naming
Asset	`contracts/`
Status	**Closed:** See Resolution
Rating	Informational

Description

Interfaces are used to describe the function specifications and events.

The following is a list of inconsistencies between the naming in interfaces and the core implementations:

- `ILendingPoolAddressesProvider.sol` and `LendingPoolAddressesProvider.sol`
  - in `setAddress()` there is `newAddress` vs `implementationAddress`
  - in `setAddressAsProxy()` there is `impl` vs `implementationAddress`
  - in `setEmergencyAdmin()` there is `admin` vs `emergencyAdmin`
- `ILendingPool.sol` vs `LendingPool.sol`
  - in `initReserve()` there is `assert` vs `reserve`
  - in `setConfiguration()` there is `asset` vs `reserve`
  - in `finalizeTransfer()` there is `balanceFromAfter` vs `balanceFromAfter`
- `ILendingPool.sol` vs `ILendingPoolCollateralManager.sol`
  - the event `LiquidationCall` there is `collateralAsset` vs `collateral`
  - the event `LiquidationCall` there is `debtAsset` vs `principal`
- `IReserveInterestRateStrategy.sol` vs `DefaultReserveInterestRateStrategy.sol`
  - in `calculateInterestRates()` there is `utilizationRate` vs `availableLiquidity`
- `IAToken.sol` vs `AToken.sol`
  - in `burn()` there is `underlyingTarget` vs `receiverOfUnderlying`
  - in `transferUnderlyingTo()` there is `user` vs `target`

The event `AddressesProviderUnregistered(address indexed newAddress)` in `ILendingPoolAddressesProviderRegistry` names the address to be removed as `newAddress` but it is not a new address.

The event `LendingPoolConfigurator.ReserveDecimalsChanged` is unused.

The event `StableDebtToken.Burn` labels the field `currentBalance` as the balance before the `amount` is burnt.

Recommendations

We recommend matching the naming of function parameters between interfaces and implementations to increase the usability and maintainability of the code base.

We also recommend matching the naming of events where `delegatecalls` are used such as the `LiquidationCall` event to prevent users incorrectly indexing emitted events.

Resolution

The recommendation has been marked as a potential improvement and will be considered through the governance procedures.

AAV-07	Unused Variables in `ValidationLogic`
Asset	`ValidationLogic.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

A struct `ValidateBorrowLocalVars` is used in the function `validateBorrow()` to store different variables used by the function.

The list of unused variables is:

- `principalBorrowBalance`
- `requestedBorrowAmountETH`
- `borrowBalanceIncrease`
- `currentReserveStableRate`
- `finalUserBorrowRate`
- `healthFactorBelowThreshold`
- `rateMode`

The variable `rateMode` is later used while unititialised as seen in AAV-12. The remaining variables are not used.

Recommendations

Consider removing the unused variables to save deployment costs, prevent accidental misuse and improve code maintainability.

Resolution

The `ValidationLogic` contract has been updated and redeployed to mainnet without the unused variables. The updates can be seen in commit 29448c1.

AAV-08	Gas Optimisation in `_calculateBalanceIncrease`
Asset	`StableDebtToken.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

There is a potential gas optimisation in `StableDebtToken._calculateBalanceIncrease()`. The gas optimisation occurs by decreasing the number of math operations.

```
uint256 balanceIncrease = balanceOf(user).sub(previousPrincipalBalance);
236

return (
238   previousPrincipalBalance,
  previousPrincipalBalance.add(balanceIncrease),
240   balanceIncrease
);
```

The following code would reduce the number of math operations from an addition and a subtraction to a single subtraction.

```
uint256 currentBalance = balanceOf(user);
236

return (
238   previousPrincipalBalance,
  currentBalance,
240   currentBalance.sub(previousPrincipalBalance)
);
```

Recommendations

Consider implementing the gas optimisation.

Resolution

The recommendation has been marked as a potential improvement and will be considered through the governance procedures.

AAV-09	Contracts Do Not Implement Safe Ownership Transfer Pattern
Asset	`Ownable.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The current transfer of ownership pattern calls the function `transferOwnership(address newOwner)` which instantly changes the owner to the `newOwner`. This allows the current owner of the contracts to set an arbitrary address (excluding the zero address).

If the address is entered incorrectly or set to an unowned address, the owner role of the contract is lost forever. Thus, a user would not be able to pass the `onlyOwner` modifier.

Similarly, the function `renounceOwnership()` allows an owner to remove themself as owner and prevent any future owners. Again this will cause any `onlyOwner` modifiers to always fail.

Recommendations

Transferring owner privileges can be mitigated by implementing a `transferOwnership` pattern. This pattern is a two-step process, whereby a new owner address is selected, then the selected address must call a `claimOwnership()` before the owner is changed. This ensures the new owner address is accessible.

Resolution

The recommendation has been marked as a potential improvement and will be considered through the governance procedures.

AAV-10	Delegate Call to Force a Self-Destruct
Asset	`LendingPool.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `LendingPool` makes a `delegatecall` to the `LendingPoolCollateralManager` when liquidating a users balance. This address of the `LendingPoolCollateralManager` is retrieved from the `LendingPoolAddressesProvider` which is passed to `LendingPool` during initialisation.

The actual `LendingPool` is initialised as a proxy which uses another contract to store the logic. The initialisation function can be called by any user and is called on the proxied `LendingPool` immediately by the `LendingPoolConfigurator` during updates and creation.

However, on mainnet, the underlying logic contract was deployed but not initialised. This would allow any arbitrary user to call the `initialize()` function. By itself this would not impact the proxy contracts and so would not impact the Aave protocol. However, `initialize()` sets the `LendingPoolAddressesProvider`.

A malicious user could call `initialize()` on the underlying logic contract with the `LendingPoolAddressesProvider` set to a contract of their design.

When the underlying `LendingPool` makes the delegate call to the address received from the maliciously designed `LendingPoolAddressesProvider`, if the function is replaced with one which is calls self-destruct then the underlying `LendingPool` logic contract will be self-destructed.

The impact of having the `LendingPool` self-destructed is that all calls made to the proxied `LendingPool` will fail. The underlying logic can be re-instated by the `LendingPoolConfigurator` and the state will not be lost.

Recommendations

This can be mitigated by ensuring that the underlying logic contract for `LendingPool` is initialised. We also recommend initialising all other logic contracts.

Resolution

The mainnet `LendingPool` logic contract has now been initialised mitigating this attack vector. Please see the Aave Security Newsletter for more details.

AAV-11	Miscellaneous General Statements
Asset	`contracts/`
Status	**Closed:** See Resolution
Rating	Informational

Description

This section describes general observations made by the testing team during this assessment that do not have direct security implications:

- A lowercase 'L' is use in `ValidationLogic.validateFlashloan()` vs uppercase in `LendingPool.flashLoan()` and `IFlashLoanReceiver`;
- `CONFIGURATOR_REVISION` is `internal` for `LendingPoolConfigurator` but `public` for all other similar contracts;
- In `StableDebtToken.sol` line [111] the comment writes 'accrueing' rather than 'accruing';
- In `ValidationLogic.validateRepay()` the following casting occurs multiple times `ReserveLogic.InterestRateMode(rateMode)` however, `rateMode` is already of type `ReserveLogic.InterestRateMode`;
- The comments in `Errors.sol` line [34–40] all say 'The liquidity of the reserve needs to be 0'. The comments do not match the functionality for most of these.
- In `LendingPoolAddressesProviderRegistry` the comments line [15] "for example with '0' for the Aave main market and '1' for the next created" But zero is not allowed in `registerAddressesProvider()` so the example should start at '1' rather than '0'.
- In `ReserveLogic.sol` the function `_accrueToTreasury()` was renamed from `_mintToTreasury()`. However, the helper struct `MintToTreasuryLocalVars` was not renamed. Consider also updating this struct name.

Recommendations

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

Resolution

The issues listed have been marked as a potential improvements and will be considered through the governance procedures.

AAV-12	Mint `ATokens` to Unpermissioned User		
Asset	`PermissionedLendingPool.sol`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `PermissionedLendingPool` allows whitelisting of users based on a set of roles. The `DEPOSITOR` role allows users to perform activities such as `deposit()` and `withdraw()`.

The function `deposit()` requires the message sender to have the `DEPOSITOR` role.

It is possible to circumvent the `DEPOSITOR` role requirements in the function `deposit()` by setting the `onBehalfOf` field to be an unpermissioned user. Note that the message sender must still have the `DEPOSITOR` role.

As a result, an unpermissioned user can receive funds. However, this user would be unable to call `withdraw()` or move the funds in any way without gaining appropriate permissions.

Recommendations

We recommend adding an additional check to ensure that the `onBehalfOf` user also has the role `DEPOSITOR`.

AAV-13	Import of Debugging Tool `Hardhat`
Asset	`DefaultReserveInterestRateStrategy.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The file `DefaultInterestRateStrategy.sol` imports a debugging tool on line [11], `import 'hardhat/console.sol';`. This is import statement unnecessarily increases the attack surface and should not be used in production.

Recommendations

The import `hardhat/console.sol` should be removed.

Resolution

`hardhat` has been removed from the list of imports.

AAV-14	Initialization `params` are Shared Between Multiple Contracts
Asset	`LendingPoolConfigurator.sol`
Status	Open
Rating	Informational

Description

When the contracts are first initialised or upgraded via `LendingPoolConfigurator.sol` each of `StableDebtToken`, `VariableDebtToken` and `AToken` take the parameter `bytes params`.

The function `_initReserve()` takes one variable `input.params` which is then sent to each of the three contracts `StableDebtToken`, `VariableDebtToken` and `AToken`.

It may be desirable for the `initialize()` function in each of these contracts to receive different `params`, depending on the use cases.

However, for simplicity it may also be desirable to have a single variable in `_initReserve()` which is shared between the token contracts, as is the current design.

Recommendations

Ensure that the design choice to share the `input.params` variable is the optimal case.

AAV-15	Potential Overminting During Configurator Updates		
Asset	`LendingPoolConfigurator.sol`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `LendingPoolConfigurator` is used to modify functionalities of the `LendingPool`. For example the function `LendingPoolConfigurator.setReserveFactor()` is used to change the `reserveFactor` percentage (i.e. the portion of fees paid to the protocol rather than creditors).

Each call to a reserve updating function (`deposit()`, `borrow`, ...) will first call `updateState()` which updates the indexes and accrues to treasury. Then `updateInterestRates()` which updates the borrowing rates and `liquidityRate`.

The `liquidityRate` is set as

$$LR_t = U_t * \overline{R_t} * (1 - reserveFactor) \approx debtorFeesRate * (1 - reserveFactor).$$

The amount accrued to the treasury in `updateState()` is

$$accrueToTreasury = debtorFees * reserveFactor.$$

From these equations we see that the fees from debtors is split between the `liquidityIndex` and `accrueToTreasury` based off the `reserveFactor`, adding to 100% of the debtorFees.

However, `setReserveFactor()` modifies the `reserveFactor` which impacts the amount accrued to the treasury without updating the `liquidityRate`. As a result the debtor fees may be over or under accounted for if we increase or decrease `reserveFactor` respectively.

For example if we increase the `reserveFactor` for 0 to 1%. Initially we'll have

$$LR_t \approx debtorFeesRate * 100\%$$

If we call `setReserveFactor()` then the `reserveFactor` will be 1%.

The next user to interact with the protocol will trigger `updateState()` (which is triggered before `updateInterestRates()`), which have the following equations

$$LI_t = (LR_t \Delta T_{year} + 1) LI_{t-1} = (debtorFeesRate * 100\% + 1) LI_{t-1}$$

$$accrueToTreasury = debtorFees * reserveFactor = debtorFees * 1\%$$

As a result we will have now acumulated 101% of the debtor fees. 100% to the liquidity index plus 1% accrued to the treasury. Thereby, overminting aTokens by 1%.

This issue is of low likelihood as it can only be triggered by the pool admin or risk admin. Furthermore, the

inaccuracies of the fees accumulated only exists for the fees generated since the last call to `updateState()`. Due to the high throughput of this protocol the discrepancy between fees earnt and amount minted is likely to be negligible.

Recommendations

Consider applying the following order of events when updating the `reserveFactor`:

1. `updateState()`
2. `setReserveFactor()`
3. `updateInterestRates()`

AAV-16	Circumvention of Checks in `validateBorrow()`		
Asset	`ValidationLogic.sol`		
Status	**Open**		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

If a user makes a borrow then repays it in the same transaction, the time difference will be zero as the block timestamp will not have changed between the calls to `borrow()` and `repay()`. Therefore users will not pay fees on a loan repaid in the same transaction since the interest accrued is zero, as it is based off time difference.

The following check was added to `validateBorrow()` to prevent users from borrowing and repaying in the same transaction.

```
require(
  lastBorrower != onBehalfOf || lastBorrowTimestamp != uint40(block.timestamp),
  Errors.VL_SAME_BLOCK_BORROW_REPAY
);
```

Here the `lastBorrowTimestamp` is the timestamp of the most recent call to `borrow()` on this reserve by any user. It is possible to circumvent this check by using a temporary account to make a borrow before the original account repays the loan.

Using two accounts `contractA` and `contractB` we may do the following:

1. contractA - `borrow()`
2. contractB - `deposit()` (with a very small amount)
3. contractB - `borrow()` (with an amount of minimal value e.g. 1 WEI)
4. contractA - `repay()`

The result is that `contractA` has borrowed and repaid within the same transaction, `contractB` will be left with a loan of minimal value.

Recommendations

Consider storing a timestamp for each user (as opposed to one for all users) per reserve which records the last borrowed time. The trade-off is that it will require additional storage costs for the user the first time they make a loan on the reserve as they will have to make an `SSTORE` instruction on an empty storage slot (this is more expansive than making an `SSTORE` on a non zero storage slot).

AAV-17	Limitations of Exposure Ceiling		
Asset	`ValidationLogic.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

An exposure ceiling was intended to be added as a control a) against supply attacks such as infinite minting which would allow draining of all funds in the pool and b) disabling an asset to be used as collateral on new loans.

The exposure ceiling would reduce the Loan to Value (LTV) ratio of an asset to zero once the total supply of an asset breached the exposure cap without reducing the ability of this asset to be used as collateral for existing loans (through the liquidation threshold).

Recommendations

It was deemed infeasible to implement the exposure cap solving both issues mentioned above without enforcing strict conditions which significantly reduce the user experience and increase gas cost.

Resolution

A contrary solution to the exposure cap was implemented where the LTV of an asset will be manually set to zero. Additional checks were added to `transfer()`, `withdraw()` and `setUserUseReserveAsCollateral()` which enforce the user to remove any collateralised assets with LTV set to zero before other collateralised assets may be modified. This enforces condition b) in that if the LTV of an asset is set to zero new loans cannot be made using the exposed asset as collateral (unless there is also sufficient collateral in assets with LTV non-zero).

The solution can be seen in PR #197

AAV-18	Ineffective `!isContract()` Check		
Asset	`ValidationLogic.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The OpenZeppelin library `Addrees.sol` provides the function `isContract()`. This function will check whether an address is a contract or not by checking if there is code associated with the address.

There are limitation to this function as described in the function documentation. One of the limitations is that a contract that is currently executing the constructor will have `isContract() = false` even though it is a contract and will eventually contain code.

The following is a check in `ValidationLogic.validateRebalanceStableBorrowRate()`,

```
require(!address(msg.sender).isContract(), Errors.LP_CALLER_NOT_EOA);
```

This check is an extra safety guarantee which prevents rogue whitelisted flash loaners from using flash loans to withdraw all the liquidity then rebalance user's stable borrow rates.

The impact of this attack is that users would have their stable interest rates increase, potentailly to the maximum value, in accordance with the interest rate strategy.

This attack is mitigated for non-whitelisted users due to the fees the user would be required to pay in order to withdraw the liquidity. The default fee for flashloans is 0.09% of the amount loaned. To rebalance a stable rate the usage ration needs to be over 95%. Hence, 95% of the available liquidity must be borrowed either by users or through the flash loan.

This cost is significantly higher than the potential reward (fees gained from depositing in this market) as the stable rate can be reversed in the following transaction by the borrower. However, for authorised flash loaners the fee is 0% and thus this attack has no cost.

Recommendations

Consider instead using a re-entrancy guard to prevent users from calling `rebalanceStableBorrowRate()` while a `flashLoan()` is in progress. This could be done by using a storage variable as a flag, it would be set when the `flashLoan()` begins and released when it ends.

The limitation of this remedy is the gas cost associated with using a storage variable. The initial gas cost for setting the storage variable to `true` will be partially recovered at the end of the function when the variable is unset.

Resolution

The check was deemed superfluous since only Aave approved smart contracts and accounts will be whitelisted. Thus it was removed in commit 325f559.

AAV-19	Incorrect Value Emitted for the Event `ReserveDataUpdated` in `flashLoan()`		
Asset	`LendingPool.sol`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

During a `flashLoan()` with the borrow rate set to `NONE`, part of the premium is accumulated to the liquidity index. This is done via the function `cumulateToLiquidityIndex()`.

The function takes the premium and calculates the percentage increase in the total supply of `aTokens`. That amount is cumulated to the liquidity index and is stored in `reserve.liquidityIndex`. However, this function does not update the value `nextLiquidityIndex` in `reserveCache`.

`nextLiquidityIndex` is later read from and used in the event `ReserveDataUpdated` during the call to `updateInterestRates()`, thereby emitting the incorrect liquidity index.

Recommendations

`nextLiquidityIndex` should be updated immediately after or during the call to `cumulateToLiquidityIndex()`.

AAV-20	Potential Misconfiguration of Flash Loan Premiums		
Asset	`LendingPool.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

As the pool admin it is possible to set the configuration of flash loan premiums such that `flashLoanPremiumToProtocol > flashLoanPremiumTotal`.

The impact of this would be that there would be a subtraction overflow when trying to calculate

```
vars.currentPremiumToLP = vars.totalPremiums[vars.i].sub(vars.currentPremiumToProtocol);
```

Since the `totalPremiums[vars.i]` will be less than `vars.currentPremiumToProtocol`. Any calls to `flashLoan()` with interest rate mode set to `NONE` would fail.

Recommendations

Consider adding sanity checks to `updateFlashloanPremiums()` to prevent the accidental misconfiguration of the flash loan premiums.

Resolution

Sanity checks have been added in PR #162.

AAV-21	`isRiskAdmin()` Modifier Unnecessarily Restrictive
Asset	`LendingPoolConfigurator.sol`
Status	Open
Rating	Informational

Description

A risk admin is a permissioned user who is able to limit the damage to the protocol during a potential attack. The function `isRiskAdmin()` is used to determine whether an address is a risk admin or not.

It has the modifier `onlyPoolAdmin` which means only the pool admin may call this function to determine if an address is a risk admin. The function is `view` and does not need to restrict users from this information.

Recommendations

We recommend removing the modifier `onlyPoolAdmin` from the function `isRiskAdmin()`.

AAV-22	Further Unused Variables
Asset	`contracts/`
Status	Open
Rating	Informational

Description

The following is a list of variables which exist inside structs but are never used:

- `LendingPool.sol`
  - `FlashLoanLocalVars`
    - `oracle`
    - `debtToken`
- `LendingPoolCollateralManager.sol`
  - `LiquidationCallLocalVars`
    - `liquidationRation`
    - `maxAmountCollateralToLiquidate`
    - `userStableRate`
    - `healthFactor`
    - `isCollateralEnabled`
    - `borrowRateMode`
  - `AvailableCollateralToLiquidateLocalVars`
    - `userCompoundedBorrowBalance`
- `ReserveLogic.sol`
  - `UpdateInterestRatesLocalVars`
    - `avgStableRate`
  - `AccrueToTreasuryLocalVars`
    - `avgStableRate`
    - `stableSupplyUpdatedTimestamp`
- `ValidationLogic.sol`
  - `ValidateBorrowLocalVars`
    - `currentLiquidationThreshold` (Note this is set but never read)
  - `validateHFAndLtvLocalVars` (Note this should be camel case)
    - `reserveDecimals`
    - `totalSupplyAtoken`

Recommendations

Consider removing the unused variables to save deployment costs, prevent accidental misuse and improve code maintainability.

AAV-23	Unused Errors
Asset	`Errors.sol`
Status	Open
Rating	Informational

Description

The following is a list of errors which are no longer used but appear as constants in `Errors`.

- `VL_CURRENT_AVAILABLE_LIQUIDITY_NOT_ENOUGH = '4'`
- `VL_DEPOSIT_ALREADY_IN_USE = '20'`
- `LP_NOT_ENOUGH_LIQUIDITY_TO_BORROW = '24'`
- `LP_REQUESTED_AMOUNT_TOO_SMALL = '25'`
- `LP_INCONSISTENT_PROTOCOL_ACTUAL_BALANCE = '26'`
- `LP_INCONSISTENT_FLASHLOAN_PARAMS = '28'`
- `CT_CANNOT_GIVE_ALLOWANCE_TO_HIMSELF = '30'`
- `CT_TRANSFER_AMOUNT_NOT_GT_0 = '31'`
- `LPC_INVALID_ATOKEN_POOL_ADDRESS = '35'`
- `LPC_INVALID_STABLE_DEBT_TOKEN_POOL_ADDRESS = '36'`
- `LPC_INVALID_VARIABLE_DEBT_TOKEN_POOL_ADDRESS = '37'`
- `LPC_INVALID_STABLE_DEBT_TOKEN_UNDERLYING_ADDRESS = '38'`
- `LPC_INVALID_VARIABLE_DEBT_TOKEN_UNDERLYING_ADDRESS = '39'`
- `LPC_INVALID_ADDRESSES_PROVIDER_ID = '40'`
- `LP_INVALID_FLASHLOAN_MODE = '47'`
- `LP_FAILED_REPAY_WITH_COLLATERAL = '57'`
- `LP_FAILED_COLLATERAL_SWAP = '60'`
- `LP_INVALID_EQUAL_ASSETS_TO_SWAP = '61'`
- `LP_REENTRANCY_NOT_ALLOWED = '62'`
- `LP_INCONSISTENT_PARAMS_LENGTH = '74'`

Recommendations

Consider commenting out these errors to save gas costs on deployment and reduce code size.

AAV-24	Removal of Deprecated Functions
Asset	`LendingPool.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The function `setPause()` and `paused()` in `LendingPool` have been deprecated in favour of handling the logic in `LendingPoolConfigurator` to save on storage reads. These functions are public in `LendingPool` and no longer provide accurate outputs, this may potentially confuse users.

Recommendations

Consider removing the functions to prevent accidental misuse.

Resolution

The deprecated functions have been removed in the PR #180.

AAV-25	Gas Optimisation - Reduce Storage Loads
Asset	`LendingPool.sol`
Status	Open
Rating	Informational

Description

Storage loads are an expensive gas operation with `SLOAD` costing 800 gas [?]. The functions `_executeRepay()` and `swapBorrowRateMode()` call the function `Helpers.getUserCurrentDebt()`. The function `Helpers.getUserCurrentDebt()` performs two `SLOAD` operations to obtain the stable debt token address and variable debt token address. These two addresses are already loaded in memory in `reserveCache` in the calling functions.

Recommendations

Consider updating the two functions such that they use the addresses stored in memory rather than those from state storage.

AAV-26	Miscellaneous General Statements - 2
Asset	`contracts/`
Status	Open
Rating	Informational

Description

This section describes general observations made by the testing team during this assessment that do not have direct security implications:

- In `ReserveLogic.sol` the function `_accrueToTreasury()` was renamed from `_mintToTreasury()`. However, the helper struct `MintToTreasuryLocalVars` was not renamed. Consider also updating this struct name.
- Spelling and Typos:
  - `ILendingPoolConfigurator.sol`
    - line [331] *"Freezes a reserve. A frozen reserve doesn't allow any new deposit, borrow or rate swap" -> "... deposits, borrows, or rate swaps"*
    - line [344] *"A paused reserve allow now user moves such as deposit, borrow, repay, swap interestrate, liquidate"*
  - `LendingPoolConfigurator.sol`
    - line [491] *"//might happen is a reserve was dropped" -> "//might happen if ..."*
  - `ReserveConfiguration.sol`
    - line [36] *"// bits 61 62 63 unused yet" -> "// bits 61 62 63 unused"*
  - `AToken.sol`
    - line [175] *"... the interest ccrued." -> "... the interest accrued"*

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_addresses_provider_set_lending_pool                          PASSED  [0%]
test_addresses_provider_set_address_as_proxy                      PASSED  [1%]
test_addresses_provider_set_lending_pool_configurator             PASSED  [1%]
test_addresses_provider_set_lending_pool_collateral_manager       PASSED  [2%]
test_addresses_provider_set_pool_admin                            PASSED  [2%]
test_addresses_provider_set_emergency_admin                       PASSED  [3%]
test_addresses_provider_set_price_oracle                          PASSED  [3%]
test_addresses_provider_set_lending_rate_oracle                   PASSED  [4%]
test_addresses_provider_set_address                               PASSED  [4%]
test_addresses_provider_only_owner                                PASSED  [5%]
test_proxy_overwrite                                              PASSED  [5%]
test_registry_register_addresses_provider                         PASSED  [6%]
test_registry_unregister_addresses_provider                       PASSED  [6%]
test_registry_multiple                                            PASSED  [7%]
test_registry_id_twice                                            PASSED  [7%]
test_registry_register_address_twice                              PASSED  [8%]
test_registry_register_address_and_id_twice                       PASSED  [8%]
test_registry_unregister_address_twice                            PASSED  [9%]
test_registry_register_id_zero                                    PASSED  [10%]
test_deposit_0x0                                                  PASSED  [10%]
test_withdraw_0x0                                                 PASSED  [11%]
test_transfers                                                    PASSED  [11%]
test_transfer_ltv_zero                                            PASSED  [12%]
test_validation                                                   PASSED  [12%]
test_deposit_withdraw                                             PASSED  [13%]
test_borrow_repay_attack                                          FAILED  [13%]
test_debt_token_base_approve_delegation                           PASSED  [14%]
test_stable_permit_delegation                                     PASSED  [14%]
test_stable_permit_delegation_invalid_signature                   PASSED  [15%]
test_stable_permit_delegation_expired                             PASSED  [15%]
test_stable_permit_delegation_zero_delegator                      PASSED  [16%]
test_variable_permit_delegation                                   PASSED  [16%]
test_variable_permit_delegation_invalid_signature                 PASSED  [17%]
test_variable_permit_delegation_expired                           PASSED  [17%]
test_variable_permit_delegation_zero_delegator                    PASSED  [18%]
test_disabled_functions                                           PASSED  [18%]
test_deploy_lending_pool                                          PASSED  [19%]
test_deploy_weth9                                                 PASSED  [20%]
test_deploy_atoken                                                PASSED  [20%]
test_deploy_delegation_aware_atoken                               PASSED  [21%]
test_deploy_default_reserve_interest_rates                        PASSED  [21%]
test_deploy_stable_debt_token                                     PASSED  [22%]
test_deploy_variable_debt_token                                   PASSED  [22%]
test_deploy_lending_pool_configurator                             PASSED  [23%]
test_deploy_lending_pool_addresses_provider                       PASSED  [23%]
test_deploy_lending_pool_addresses_provider_registry              PASSED  [24%]
test_deploy_lending_pool_collateral_manager                       PASSED  [24%]
test_deploy_aave_oracle                                           PASSED  [25%]
test_deploy_aave_protocol_data_provider                           PASSED  [25%]
test_deploy_weth_gateway                                          PASSED  [26%]
```

```
test_initial_reserve_state                                  PASSED  [26%]
test_deposits_no_collateral_and_borrowings                  PASSED  [27%]
test_deposits_with_permit                                   PASSED  [27%]
test_simple_deposit_on_behalf_of                            PASSED  [28%]
test_deposits_collateral_and_borrowings_off                 PASSED  [28%]
test_deposit_with_debt                                      PASSED  [29%]
test_withdraw_borrowings_and_collateral_off                 PASSED  [30%]
test_withdraw_to                                            PASSED  [30%]
test_withdraw_no_borrowings_and_collateral                  PASSED  [31%]
test_withdraw_with_debt                                     PASSED  [31%]
test_stable_borrow                                          PASSED  [32%]
test_borrow_on_behalf                                       PASSED  [32%]
test_borrow_ltv_zero                                        PASSED  [33%]
test_stable_borrow_base_rate_zero                           PASSED  [33%]
test_variable_borrow                                        PASSED  [34%]
test_repay_stable_base_rate_zero                            XFAIL   (...) [
test_repay_stable                                           PASSED  [35%]
test_repay_with_permit                                      PASSED  [35%]
test_repay_on_behalf_of                                     PASSED  [36%]
test_repay_variable                                         PASSED  [36%]
test_swap_borrow_rate_mode_variable_to_stable               PASSED  [37%]
test_swap_borrow_rate_mode_stable_to_variable               PASSED  [37%]
test_rebalance_stable_borrow_rate                           PASSED  [38%]
test_rebalance_stable_borrow_rate_with_no_debt              PASSED  [38%]
test_set_user_use_reserve_as_collateral                     PASSED  [39%]
test_liquidation_call_stable_rate_zero                      PASSED  [40%]
test_liquidation_call_variable                              PASSED  [40%]
test_liquidation_call_stable_and_variable                   PASSED  [41%]
test_liquidation_call_with_atokens                          PASSED  [41%]
test_liquidation_call_max_collateral                        PASSED  [42%]
test_liquidation_call_self                                  PASSED  [42%]
test_flash_loan                                             PASSED  [43%]
test_flash_loan_authorised_borrower                         PASSED  [43%]
test_flash_loan_variable                                    PASSED  [44%]
test_flash_loan_stable                                      PASSED  [44%]
test_flash_loan_multiple                                    FAILED  [45%]
test_mint_to_treasury                                       PASSED  [45%]
test_only_lending_pool_configurator                         PASSED  [46%]
test_init_reserve                                           PASSED  [46%]
test_init_reserve_twice                                     PASSED  [47%]
test_init_reserve_repeated                                  PASSED  [47%]
test_only_pool_admin                                        PASSED  [48%]
test_pausing                                                PASSED  [48%]
test_only_emergency_admin                                   PASSED  [49%]
test_update_stable_debt_token                               PASSED  [50%]
test_update_variable_debt_token                             PASSED  [50%]
test_update_atoken                                          PASSED  [51%]
test_reserve_freezing                                       PASSED  [51%]
test_reserve_activating                                     PASSED  [52%]
test_reserve_stable_rate_enabling                           PASSED  [52%]
test_reserve_borrowing_enabling                             PASSED  [53%]
test_set_reserve_factor                                     PASSED  [53%]
test_set_reserve_interest_rate_strategy_address             PASSED  [54%]
test_configure_reserve_as_collateral                        PASSED  [54%]
test_risk_admin_registry                                    PASSED  [55%]
test_only_risk_or_pool_admins                               PASSED  [55%]
```

```
test_drop_reserve                                                PASSED  [56%]
test_set_borrow_cap                                              PASSED  [56%]
test_supply_cap                                                  PASSED  [57%]
test_authorize_flash_borrower                                    PASSED  [57%]
test_update_flash_loan_premiums                                  PASSED  [58%]
test_linear_interest                                             PASSED  [58%]
test_linear_interest_one_year                                    PASSED  [59%]
test_linear_interest_max_values                                  PASSED  [60%]
test_linear_interest_zero                                        PASSED  [60%]
test_compound_interest                                           PASSED  [61%]
test_compound_interest_overflows                                 PASSED  [61%]
test_compound_interest_zero                                      PASSED  [62%]
test_percent_mul                                                 PASSED  [62%]
test_percent_mul_zero                                            PASSED  [63%]
test_percent_mul_overflow                                        PASSED  [63%]
test_percent_div                                                 PASSED  [64%]
test_percent_div_zero                                            PASSED  [64%]
test_percent_div_overflow                                        PASSED  [65%]
test_rebalance_attack                                            PASSED  [65%]
test_add                                                         PASSED  [66%]
test_sub                                                         PASSED  [66%]
test_mul                                                         PASSED  [67%]
test_div                                                         PASSED  [67%]
test_borrow_basic                                                PASSED  [68%]
test_collateral_basic                                            PASSED  [68%]
test_config_fuzz                                                 PASSED  [69%]
test_deposit_frozen                                              PASSED  [70%]
test_deposit_paused                                              PASSED  [70%]
test_deposit_deactivated                                         PASSED  [71%]
test_deposit_invalid_amount                                      PASSED  [71%]
test_deposit_supply_cap                                          PASSED  [72%]
test_withdraw_invalid_amount                                     PASSED  [72%]
test_withdraw_paused                                             PASSED  [73%]
test_withdraw_insufficient_balance                               PASSED  [73%]
test_withdraw_deactivated                                        PASSED  [74%]
test_borrow_while_frozen                                         PASSED  [74%]
test_borrow_when_paused                                          PASSED  [75%]
test_borrow_zero                                                 PASSED  [75%]
test_borrow_cap                                                  PASSED  [76%]
test_borrowing_disable_borrowing_on_reserve                      PASSED  [76%]
test_borrowing_bad_interest_rate_mode                            PASSED  [77%]
test_borrowing_no_collateral                                     PASSED  [77%]
test_borrowing_bad_health_factor                                 PASSED  [78%]
test_borrowing_insufficient_colalteral                           PASSED  [78%]
test_borrowing_disable_stable_borrowing_on_reserve               PASSED  [79%]
test_borrowing_stable_borrow_same_as_collateral                  PASSED  [80%]
test_borrowing_stable_borrow_more_than_max                       PASSED  [80%]
test_repay_deactivated                                           PASSED  [81%]
test_repay_paused                                                PASSED  [81%]
test_repay_zero                                                  PASSED  [82%]
test_repay_with_no_debt                                          PASSED  [82%]
test_repay_max_on_behalf_of                                      PASSED  [83%]
test_swap_rate_deactivated                                       PASSED  [83%]
test_swap_rate_paused                                            PASSED  [84%]
test_swap_rate_frozen                                            PASSED  [84%]
test_swap_rate_no_stable_debt                                    PASSED  [85%]
```

```
test_swap_rate_no_variable_debt                      PASSED  [85%]
test_swap_rate_to_stable_with_collateral             PASSED  [86%]
test_swap_rate_bad_mode                              PASSED  [86%]
test_rebalance_deactivated                           PASSED  [87%]
test_rebalance_paused                                PASSED  [87%]
test_rebalance_below_liquidity_threshold             PASSED  [88%]
test_rebalance_below_rate_threshold                  PASSED  [88%]
test_validate_reserve_as_collateral_no_balance       PASSED  [89%]
test_validate_reserve_as_collateral_no_reserve       PASSED  [90%]
test_validate_reserve_as_collateral_paused           PASSED  [90%]
test_liquidation_call_deactivated                    PASSED  [91%]
test_liquidation_call_paused                         PASSED  [91%]
test_liquidation_call_health_factor                  PASSED  [92%]
test_liquidation_call_not_used_as_collateral         PASSED  [92%]
test_liquidation_call_with_no_debt                   PASSED  [93%]
test_validate_flash_loan                             PASSED  [93%]
test_validate_flash_loan_paused                      PASSED  [94%]
test_validate_hf_and_ltv_health_factor               PASSED  [94%]
test_validate_hf_and_ltv                             PASSED  [95%]
test_validate_transfer_paused                        PASSED  [95%]
test_validate_drop_reserve                           PASSED  [96%]
test_getters                                         PASSED  [96%]
test_wad_mul                                         PASSED  [97%]
test_wad_div                                         PASSED  [97%]
test_ray_mul                                         PASSED  [98%]
test_ray_div                                         PASSED  [98%]
test_ray_to_wad                                      PASSED  [99%]
test_ray_mul_div_rounding                            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.

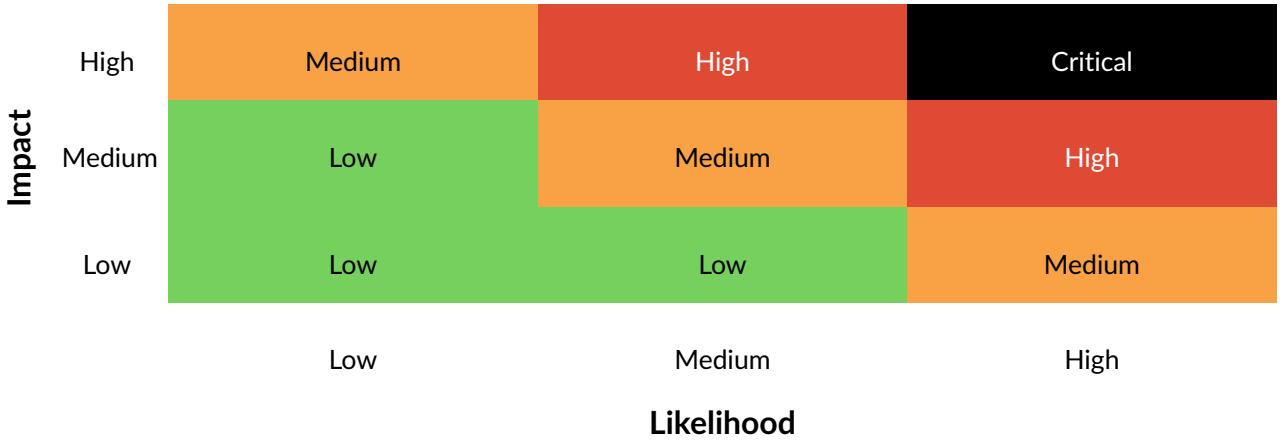

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

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] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 2014. Available: https://ethereum.github.io/yellowpaper/paper.pdf.