Introduction

The Aave team team asked us to review and audit their smart contracts for version 3 of their lending protocol. We looked at the code and now publish our results.

Scope

We audited commit `14f6148e21b477d78347db6a1603039c9559e275` of the aave-v3-core repository. The scope includes all files in the `protocol` and `interfaces` directory, excluding those that are not being used in any contract within the `protocol` directory.

System overview

Aave is an open-source, decentralized, and non-custodial liquidity market protocol where users can participate as depositors or borrowers. The depositors provide liquidity to specific asset markets to earn a passive income. Borrowers may take out loans secured by over-collateralized deposits, paying interest at a stable or variable rate. Users may also perform non-collateralized borrowing through flash loans while paying a fee.

Users must first deposit a supported asset to interact with the protocol. This deposit will accrue interest based on the amount deposited and the market borrowing demand. Users can also use these deposited assets as collateral for borrowing money in the protocol, contributing to the total interest rate accumulated.

Lending

When users deposit an amount of a particular asset in a pool, they receive a corresponding amount of aTokens that represent their position and continuously earn interest based on market conditions. All depositors share the interest paid by borrowers, based on the average borrow rate multiplied by the utilization rate. The higher the borrowed asset's amount is, the higher the yield for depositors, reaching the maximum amount when the total borrowing amount is close to the total deposit amount.

Borrowing and Liquidations

Users that wish to borrow some assets must first deposit collateral. Instead of selling their collateral asset and closing their position on it, users can obtain liquidity while still being exposed to the collateral's potential upside value gain.

For instance, a user may want liquidity to trade a particular asset A without losing exposure to asset B. They can then borrow an X amount of A from the protocol by giving a Y amount of B as collateral. Suppose the price of B goes up by 50% after three months, when closing their position of A. In that case, they will get the initial amount of B deposited as collateral (getting the upside) while being able to invest A elsewhere throughout those three months.

On the other hand, the collateral price can go down to the point where the collateral asset may not cover the borrowed amount. To avoid this situation, the protocol uses a health factor to track the users' positions. If the price of the collateral asset goes down to the point where the health factor goes below 1, a user's position will be liquidated so that the collateral can cover the loan. If a liquidation happens, a % of the borrower's loan and a liquidation fee will be taken from the user's available collateral. Anyone can take the role of a liquidator to receive a reward for contributing to the protocol's health.

Moreover, the protocol provides a non-collateralized borrowing feature known as flash loans. Flash loans are automated loans in which the borrowed principal must be returned in the same transaction where it was requested. It is a developer-focused feature that allows the user to borrow any available amount of an asset. The flash-borrower has to pay a 0.09% of the flash-loaned amount as a fee to pay the liquidity providers (depositors).

Swap and repay

The protocol allows users to swap their deposited assets for other assets and repay loans with deposited collateral. Users can swap a deposited asset A for another deposited asset B (which can be collateral) or repay asset A's loan with asset B's collateral.

High-level overview of V3's new features

Aave V3 introduces the following new features:

- **Portal**: A feature that allows users to move their assets deployed in V3 through different networks by burning aTokens on the source network and minting them on the destination network.

- **Efficiency mode**: A feature that allows borrowers to categorize assets based on: liquidation threshold, liquidation bonus, a custom price oracle, and LTV. Borrowers can choose a particular category of assets they want to borrow. An example of a "category" can be all the assets pegged to the USD (DAI, FEI, USDC). When a user uses the efficiency mode and supplies assets of the same category as their collateral, the LTV and liquidation threshold are overridden by the category's parameters to optimize capital efficiency.

- **Isolation mode**: Assets can be listed as isolated, and borrowers supplying an isolated asset as collateral will not be able to supply other assets as collateral. This isolated collateral can only be used to borrow stablecoins permitted by the protocol governance and define a debt ceiling, which means that users' borrow amounts are limited in quantity.

- **Risk management features**:

- Supply and borrow caps: Token holders, through governance, will be able to configure supply and borrow caps, to minimize liquidity insolvency with the latter and reduce protocol exposure to a particular asset by preventing attacks such as price oracle manipulation with the former.

- Granular borrowing power control: Token holders, through governance, are able to change the collateral factor for borrows that happen in the future without affecting existing borrows or triggering a liquidation event.

- Risk admins: A permit list of entities (that can be DAOs or any other automated agent), added by governance and that act as guardians, is introduced to modify specific risk parameters without going through governance voting.

- Price oracle sentinel: A sentinel introduced for L2 protocols to manage downtime periods of the sequencer by introducing a liquidation grace period and disabled borrows under certain circumstances.

- **Asset listing admins**: A permit list of entities (could be DAOs or any other agent) to implement new strategies to add new assets to the protocol without going through an on-chain voting process.

- **Simplified flash loans**: A new way of flash-borrowing an asset through the `flashloanSimple` function, which is simpler to use and less expensive in terms of gas consumption.

Upgradeability

The most critical modules of the protocol are upgradeable, following a similar pattern as the OpenZeppelin's transparent proxy pattern. The main contracts that are upgradeable are:

- `Pool.sol`
- `PoolConfigurator.sol`
- `AToken.sol`
- `StableDebtToken.sol`
- `VariableDebtToken.sol`

The Aave governance can vote for changes in the protocol. The functionality of the contracts mentioned above can be updated by the proxy admin, who has restricted access to the upgrade functionality.

Oracles

We assume all price oracles work as intended since they are out of scope for this audit. However it is essential to point out that they play critical roles in the Aave protocol. The Aave team and governance system should be extremely careful when selecting assets for the protocol to ensure their prices are manipulation-resistant to avoid any oracle manipulation attacks.

Three auditors have audited the code over the course of seven weeks, and here we present our findings.

Critical severity

[C01] Minting VariableDebtTokens uses requestor's state instead of onBehalfOf's state

When executing a new borrow, a requestor can act on behalf of another user which holds the collateral and has given enough allowance to execute it.

At the same time, each new borrow implies the minting of either stable or variable debt tokens, depending on the interest rate chosen when performing the borrow request. However, when the borrow functionality mints the variable debt tokens, they are sent correctly to the `onBehalfOf` address but `scaledBalance`, `_userState.additionalData` and `accumulatedInterest` are wrongly calculated, since they are using the requestor's info instead of `onBehalfOf`.

As a consequence, line 111 will return false if the requestor already has a positive balance of those variable tokens. This means that if it is the first time that the owner executes a borrow for a particular asset, it will avoid setting the asset as it will be used for borrowing with their configuration. If this happens, the owner's health factor will be wrongly calculated as it will not accrue debt for this specific borrow, potentially letting the owner execute more borrows having a bad health factor.

This can be dangerous for the protocol, since users may start new unhealthy debt positions. Moreover, the fact that `user` is used instead of `onBehalfOf` on its own is an issue that incorrectly calculates the `accumulatedInterest` and the `scaledBalance` variables, and also save the `index` in the wrong user configuration.

Conside replacing `user` with `onBehalfOf` in the calculation of `accumulatedInterest` and `scaledBalance`, and setting the `index` in the `additionalData` field of the correct `_userState`.

Update: *Fixed in PR#281.*

[C02] eMode category is skipped in borrows

One of the functionalities that the Aave protocol offers when supplying and borrowing a particular asset is to let the requestor use assets on behalf of other users through the `onBehalfOf` parameter. When borrowing on behalf of another user, the following happens:

- The requestor of the borrow needs to have enough borrow allowance set in the `DebtToken` contract
- The requestor executes the borrow, but the collateral check and the debt tokens mint are executed on the `onBehalfOf` parameter.
- The requestor will receive the borrowed asset, and the `onBehalfOf` will have a debt position opened.

For this reason, the `borrow` function of the `Pool` contract is taking into account the `userConfig[onBehalfOf]` user configuration when calling the borrow logic.

However, the `usersEModeCategory` being used in the requestor's one instead of the `onBehalfOF` one. During the validation of the borrow, instead of matching the owner `eMode` category, the requestor's category is checked instead.

This means that:

- If requestor has a category set that is not the same category as the reserve, the request will fail on this check.
- If the requestor does not have any category set, the request will skip the previously mentioned check, and the user can skip the owner's category if they have one set.

The latter is critical since the following will happen:

- Since `eMode` categories overwrites LTV, oracle price of the asset, liquidation threshold the requestor will end up potentially borrowing with less efficient parameters (higher liquidation threshold, lower ltv).

- `eMode` categories also overwrite the `liquidationBonus` and if the inefficient borrow executed in this case gets liquidated, the original owner's `eMode` parameters will apply now, mismatching the parameters used when executing the borrow (being the owner's category, potentially different from the requestor). For example, if the `liquidationThreshold` of the skipped `eMode` category is lower than the actual requestor's category liquidation threshold, a liquidator will have to wait until the owner liquidation threshold is reached before liquidating.

- After the check is skipped, a wrong `userEModeCategory` will be used calculate the user's account data, which internally calculates a health factor. This health factor is not the owner's one opening the debt position as it should be, but instead, it is the requestor health factor. This opens the door to the borrower to continue borrowing outside of his `eMode` category even with a dangerous health factor by just delegating the job to a healthy requestor.

Consider passing the right `eMode` category when executing a borrow on behalf of another user.

Update: *Fixed in PR#204.*

High severity

[H01] Isolation debt is not updated

Whenever a borrow position becomes unhealthy or under-collateralized, the Aave protocol will allow other users to liquidate the position through the `liquidationCall` function.

At the same time, the protocol offers a feature called *isolation mode*, a special category for users supplying/ borrowing *isolated assets*, which are assets with a cap on the total debt amount tracked by the `isolationModeTotalDebt` variable.

The `isolationModeTotalDebt` variable is updated when borrowing and repaying with an isolated asset. However, this variable is not updated if a user borrowing an isolated asset gets liquidated.

This means that every time a borrow of an isolated asset is liquidated, the debt is maintained. If this happens several times, it can potentially disallow the protocol from offering new borrows for these specific assets.

Consider properly tracking the `isolationModeTotalDebt` when liquidating debt positions to correctly account the debt removal for isolated assets.

Update: *Fixed in PR#251.*

[H02] Miscalculation of bonusCollateral on liquidations

In order to protect the protocol, Aave allows users to liquidate unhealthy positions through the `liquidationCall` function. When this happens, liquidators provide an amount as debt to cover for the unhealthy debt position in exchange for a reward for maintaining the protocol health.

During liquidations, the protocol calculates the `bonusCollateral` as the amount of collateral rewarded to the liquidator using the `liquidationBonus` percentage parameter. If the `liquidationProtocolFeePercentage` variable is set to a positive value, the protocol will take out a fee from the `bonusCollateral` by multiplying it by the `liquidationProtocolFeePercentage`. Notice that by definition, all percentages are expressed with two decimals, so, for example, 100% would be expressed as `10000`.

Also, according to v2 documentation the liquidation bonus is expressed as an amount on top of the principal value, so `5%` liquidation bonus is actually expressed as `105%`.

When trying to liquidate a big part of the entire position, the collateral to liquidate plus the bonus for the liquidator and/or the protocol might exceed user's collateral balance. In this case the liquidator reward is given as a discount on the debt amount to cover instead, which will require the `bonusCollateral` to be recalculated in order to get the final `liquidationProtocolFee` based on that.

In this second case, the `bonusCollateral` is calculated differently from the previous case. Specifically, suppose `baseCollateral == collateralAmount == 100 * 1e8` and a `liquidationBonus == 10500`. In the first case the system calculates `bonusCollateral` as `baseCollateral * liquidationBonus - baseCollateral = 5 * 1e18` while in the second case it is calculated as `collateralAmount / liquidationBonus = 95 * 1e18` where `collateralAmount` has been fixed to its maximum value, the user balance.

This last calculation wrongly takes the complementary value of the percentage calculation, resulting in a `95%` fee destinated to the protocol instead of `5%`, which discourages liquidators to run liquidations, finally undermining the health of the protocol.

Consider changing how `bonusCollateral` is calculated in line 349 to `bonusCollateral = vars.collateralAmount - vars.collateralAmount.percentDiv(liquidationBonus)`.

Update: *Fixed in PR#271.*

[H03] Incorrect amount of aTokens may be minted to users

The Aave protocol is designed to accept underlying tokens and issue corresponding aTokens back to the user on a 1 to 1 base. However, during this process, the protocol never validates the actual amount of underlying tokens received before issuing the aTokens out: it simply takes the `params.amount` given by the user during the transfer of underlying tokens and use it to mint the same amount of aTokens back to the user.

This exposes the protocol to risk since certain tokens may charge a fee or decide to charge a fee later during a transfer operation (e.g., USDT). If this happens, it will potentially cause the corresponding aTokens to be under-collateralized.

Consider using the delta value of the before and after balances of the pool's underlying token as the final minting amount.

Update: *Acknowledged. In the words of the Aave team "It is a design choice to not have any implementation to handle nonstandard behaviors in the default codebase".*

Medium severity

[M01] Lump-sum income distribution can be manipulated with liquidity

Throughout the Aave protocol, there are two processes where a lump sum of income is distributed to liquidity providers:

- Liquidity provider premium from flash loans in the `executeFlashLoan function` of the `FlashLoanLogic` library.
- Liquidity provider fees from the bridge in `backUnbacked function` of the `BridgeLogic` library

These lump-sum fees are distributed by calling the function `cumulateToLiquidityIndex function` of the `ReserveLogic` library, where the lump-sum amount is evenly distributed in the reserve by updating the liquidityIndex:

`newLiquidityIndex = lumpSumAmount / (totalLiquidityAmount / oldLiquidityIndex) + oldLiquidityIndex`

However, since the protocol only uses the spot liquidity to distribute the lump sum income, a malicious liquidity provider could manipulate the distribution by providing a large amount of liquidity right before the distribution and pull out the liquidity immediately afterwards. This would allow the attacker to potentially pocket a large proportion of the lump sum amount at the cost of other liquidity providers.

Consider redesigning the distribution mechanism to factor in more than just the spot liquidity to avoid manipulation.

Update: *Not fixed. The team has acknowledged the issue.*

Low severity

[L01] Addresses provider inconsistencies

The `unregisterAddressesProvider` function of the `PoolAddressesProviderRegistry`contract is doing a `reset of the _addressesProviders mapping` leaving `the _addressesProvidersList array unmodified`.

For this reason, the `getAddressesProvidersList` function will iterate over old values that might have been deleted, making it very inefficient.

Moreover, the documentation stated that `0` is used to represent the mainnet Aave market, but when unregistering an address provider, `0` value is used to deactivate that provider, conflicting with the previous definition.

Consider reducing the array size by swapping the last element with the one that is going to be deleted and popping the last element from the array to correctly update the array size and content. This will change the order of the elements in the array but will definitely be more storage efficient for both saving and retrieving those values. Lastly, consider being consistent in the indexes used and their meanings, avoiding using the zero value as a meaningful representation of a valid state.

Update: *Acknowledged.*

[L02] Incorrect amount of aTokens may be approved

To accommodate scaling, the latest design of the aToken does not change an individual aToken's value against its underlying backing while still providing ways to determine the interest accumulated.

This unique design means the aTokens minted to a user may be different from their balance: in fact, all aTokens' functions dealing with the aToken balance may require a scaled output. This is the case for aToken's `balanceOf`, `totalSupply`, and `transfer` functions, but not for the `approve` function.

The approving process uses the inherited `approve` function of the `IncentivizedERC20` contract, which does not deal with scaling. Rather, it approves whatever `amount` that the user provides as a parameter instead of approving the scaled-down base amount. This could lead to over-approving the user's amount and potentially further over-spending.

This issue also applies to the newly added `permit` function of the `AToken` contract, which uses the same `approve` function.

To avoid this behavior and to be consistent with the chosen design over all the ERC20 functions dealing with aTokens amounts, consider refactoring the `approve` function to handle scaled-down values of the given aToken amount.

Update: *Not fixed. The team has acknowledged the issue.*

[L03] Incorrect fees might be used for the bridge

The bridge logic has two separate functions to `mintUnbacked` and `backUnbacked`. Moreover protocol fee is charged in L108 of `backUnbacked` calculated with the current `protocolFeeBps`.

However, since the `protocolFeeBps` might change in between `mintUnbacked` and `backUnbacked`, it is not always suitable to use the latest `protocolFeeBps` as it might not be what the user has agreed on when calling the `mintUnbacked` function.

Considering saving the `protocolFeeBps` during `mintUnbacked` and use the saved value to charge user instead of the latest `protocolFeeBps`.

Update: *Acknowledged. In the words of the Aave team: "The implementation is minimal on purpose. Consistency of the protocol fee is to be handled at the bridge level".*

[L04] Incomplete event emission after sensitive actions

When defining events with the purpose of showing a storage modification, it is a good practice to emit both the old value and the new value of the modified variable. For example, the `AddressSet`, `PoolConfiguratorUpdated`, `PriceOracleUpdated`, and `PoolUpdated`, `PoolDataProviderUpdated` events in the `PoolAddressesProvider` contract should emit both old and new addresses (including the zero address on setup).

Additionally, the `BridgeAccessControlUpdated` event of `IPoolAddressesProvider` interface is defined but not emitted anywhere in the codebase, and the `MarketIdSet` event from the `IPoolAddressesProvider` has no indexed parameters.

As a general recommendation, consider emitting events after sensitive changes take place to facilitate tracking and notify off-chain clients following the contracts' activity, checking that all sensitive variables are being emitted (taking into account both old and new values), to avoid hindering the task of off-chain services interested in these events.

Update: *Fixed in PR#382.*

[L05] Not handling upgradeToAndCall error message on failure

The `upgradeToAndCall` function in the `BaseImmutableAdminUpgradeabilityProxy` contract allows the PoolAddressesProvider contract to update the implementation address and then delegates a call to this new implementation.

However, this delegate call does not handle errors that the implementation may return, nor reverts with a custom error message. This means that if the call fails, no information will be provided by the proxy contract, hindering valuable information in the upgrade process.

It must be noted that even if out of scope, this same pattern is being followed in the `initialize` function from the `InitializableUpgradeabilityProxy` contract.

Consider verifying the delegate call result, and return the `returndata` if the call succeeded. If the called function does not return, then consider either bubbling the error message or revert with a custom error message.

Update: *Not fixed. The acknowledge statement for this issue by the Aave team is: "The proposed fix requires changes to dependency libraries and the code has been working as expected for the previous two iterations of the protocol".*

[L06] Potential bridge failure

Through the `PoolConfigurator` contract, the `pool` or `risk admins` can set the `unbackedMintCap` parameter to any value, including `0`.

At the same time, the `unbackedMintCap` is used by the bridge in the `BridgeLogic` contract. There, the `mintUnbacked` function requires the `unbackedMintCap` to be positive, eventually reverting the transaction if it happens to be zero.

Moreover, the current check of `unbackedMintCap` to be positive is redundant since the second condition in the same require statements requires that `unbackedMintCap > unbacked/(10**reserveDecimals)` where `unbacked` is always non negative.

Regardless of the code intention, in order to improve consistency on the design, consider enforcing `unbackedMintCap` to not be zero or update the `BridgeLogic` implementation to correctly handle this specific case. In both cases, consider refactoring the require statement to avoid redundant checks.

Update: *Acknowledged. This is an expected behaviour. In the words of the Aave team: "If the* `unbackedMintCap` *is 0, we* ***do not*** *want to allow the bridge contract to mint anymore, therefore is not a failure". The redundant check has been fixed in PR#221.*

[L07] Wrong require statement for borrowCap

When validating a borrow execution, the `ValidationLogic` library requires the total borrowed amount to be less than the `borrowCap` parameter if it is set.

Moreover, when setting the `borrowCap`, there are no restrictions in setting it exactly as `MAX_VALID_BORROW_CAP` but the require statement is checking that the `vars.totalDebt / vars.assetUnit` is strictly less than `vars.borrowCap` and not less or equal.

Consider improving correctness and consistency by fixing the inequality, requiring `<=` instead of `<`.

Update: *Fixed in PR#269.*

Notes & Additional Information

[N01] Configuration changes might affect current users borrows

The governance of the system is able to change many parameters in the pool and reserve configurations. Some of these parameters might have a direct impact on user's active positions when the change takes place.

For example, the `reserve.liquidationThreshold`, the `eModeCategory` containing `liquidationThreshold`, `liquidationBonus`and `ltv` values and the `reserveInterestRateStrategyAddress`. If the current design intention is to allow these changes to take place while there are active borrow positions, consider properly warning the users and allow them to opt out of their position before the change.

If this is not the intended design, consider saving these parameters into the position itself at the time of executing a borrow, and use the saved values when calculating interests, executing liquidations or using a specific `eMode` category parameter.

Update: *Acknowledged. In the words of the Aave team: "This is intended behavior. The governance system has a proper timelock mechanism to allow users to opt out".*

[N02] Functions can be combined

The `isBorrowAllowed` and `isLiquidationAllowed` functions of the `PriceOracleSentinel` contract are performing the same exact operation which is to call the `_isUpAndGracePeriodPassed`.

To improve code efficiency and readability, consider removing those two functions and consider making the `_isUpAndGracePeriodPassed` function public, changing the corresponding interface accordingly. If this is a preparation for future development where the two functions can act differently, consider properly documenting this so that readers can understand the intentions behind this choice.

Update: *Acknowledged. In the words of Aave team: "Although they currently implement the same logic, it's highly likely that in the future the two implementations will diverge as borrow and liquidate are vastly different".*

[N03] Gas optimizations

In the codebase, there are several places where some gas can be saved either from storage use and from code executions. Some examples are:

- `CalculateUserAccountDataVars` and `ReserveCache` structs are all defining `uint256` while many of those parameters will never occupy such large numbers, some can be reduced in size and tightly packed together.
- Often, the `reserve` and `reserveCache` entities are recovered and then `updateState` is called right after recovering those. These grouped operations are performed with no modifications in many places and can definitely benefit from a refactor where the operations are performed in a modular function which is called in all occurrencies.
- Line 199 of the `ValidationLogic` contract can use `vars.reserveDecimals` instead of retrieving them again.
- Lines 139-143 of the `GenericLogic` contract produce the same exact result as doing `vars.userBalance = IScaledBalanceToken(currentReserve.aTokenAddress).balanceOf(params.user)` directly, reducing the lines involved. If this is intended due to the differences in call stack lengths in both cases, consider properly documenting it and justifying it.

To improve gas consumption, readability and code quality, consider refactoring wherever possible such examples, carefully reviewing the entire codebase.

Update: *Partially fixed in PR#373.*

[N04] Inconsistent or unclear code style

There are some places in the codebase that might benefit from a better coding style or format. Some examples are:

- The `setAddressAsProxy` function of the `PoolAddressesProvider` doesn't match exactly the named parameters defined in the interface.
- The `AddressesProviderUnregistered` event definition has `newAddress` as a named parameter while it should be `oldAddress`;
- The `_addressesProvider`, `_oracle` and `_gracePeriod` variables of the `PriceOracleSentinel` contract are all public but with a `_` prepended name, while this format is usually reserved for private/ internal variables. Since public variables will create getters, those getters will have a different format from the rest of the codebase.

To improve clarity and readability, consider carefully reviewing the style and format used in the whole codebase, making sure it is clear, standard, and consistent on its own.

Update: *Fixed. The* `setAddressAsProxy` *function has been fixed in commit* `f73c73d4cd601fb5be0dbe35720b77450b1091c1`*, the* `AddressProviderUnregistered` *event refactored in commit* `0395f1ee80d9915dfc520009d58f7d71c396bf61`*, while the variables of the* `PriceOracleSentinel` *in commit* `fe77bbc6a3b506c0589906d4cd88b0979d865e36` *and* `3c6352fa3907407e46c1df6d9338cd638d974ece`*.*

[N05] Inexistent or inconsistent require messages

The `Errors` library defines the error messages emitted by the different contracts of the Aave protocol, and are used throughout the codebase in `require` statements. However, there are some places where these error messages are not used, and custom error messages are used instead. Some examples are:

- Line 79 of `BaseImmutableAdminUpgradeabilityProxy.sol`.
- Line 32 of `ACLManager.sol`.

Additionally, line 72 of `BaseImmutableAdminUpgradeabilityProxy.sol` does not define any error message.

For consistency, consider defining these error messages in the Errors library, and consider including specific and informative error messages in all require statements.

Update: *Partially fixed. The issue has been address in PR#367 where other examples have been fixed too. However the issue related with the* `BaseImmutableAdminUpgradeabilityProxy` *is not fixed.*

[N06] Lack of explicit visibility in state variables

The following state variables and constants are implicitly using the default visibility:

- In the `PoolStorage` contract: `eModeCategories`, and `_usersEModeCategory`.
- In the `BaseImmutableAdminUpgradeabilityProxy` contract: `ADMIN`.

To favor readability, consider explicitly declaring the visibility of all state variables and constants.

Update: *Fixed in PR#379.*

[N07] Lack of input validation

In the codebase there are some places where input values are not checked to be meaningful or correct. Even if most of the occurrences are governance-callable functions, given the decentralization of the protocol, it is important to make sure that the protocol is protected by malicious values passed in.

In particular:

- The `ACLManager` and the `PoolAddressesProvider` contract has no checks on any value passed in any function.
- The `calculateCompoundedInterest` function of the `MathUtils` library is not checking whether the `currentTimestamp` is greater or equal than the `lastUpdateTimestamp`. If that's the case, the call will revert because of underflow in the calculation of `exp`.

These are just a few examples that we have found. We recommend reviewing the codebase looking for more occurrences, and setting some validation checks where fundamentally needed, taking into consideration the decentralized nature of the protocol.

Update: *Acknowledged. Most of the inputs are addresses and the lack of validation on those is an intended design to let the governance set null addresses on purpose. Regarding the invalid timestamp in* `calculateCompoundedInterest` *the Aave team says that "the revert in case of wrong timestamps is more than enough as validation", but we still suggest explicitly checking for it following the fail fast-and-loud principle.*

[N08] Missing or erroneous docstrings and comments

Several docstrings and inline comments throughout the codebase were found to be erroneous and should be fixed. *Some examples* are:

- Line 161 of `UserConfiguration.sol` should say "The address of the *only* asset used as collateral".
- Line 195 `UserConfiguration.sol` states that it returns the address of the first collateral asset, but it returns the index (key) inside the `reservesList` mapping.

- Lines 13 and 14 of the PoolAddressesProviderRegistry contract says "for example with 0 for the Aave main market and 1 for the next created.", but based on the code, when the id is 0, it means that the provider is inactive/unregistered.
- Lines 49 and 50 of the DataTypes contract are repeated.
- The P character is used in Errors contract either for Pool or Pausable as documented in the comments. Consider not using the same string for both.
- Lines 49 and 50 of the DataTypes contract are repeated.
- The P character is used in Errors contract either for Pool or Pausable as documented in the comments. Consider not using the same string for both.

Additionally, some contracts and functions lack documentation. This hinders reviewers' understanding of the code's intention, which is fundamental to correctly assess not only security, but also correctness. Additionally, docstrings improve readability and ease maintenance. They should explicitly explain the purpose or intention of the functions, the scenarios under which they can fail, the roles allowed to call them, the values returned, and the events emitted. *Some examples* are:

- The getPriceOracle and setPriceOracle functions of IPoolAddressesProvider.
- All events definitions of IPoolAddressesProvider.
- Line 297 of LiquidationLogic.sol misses the description of the third return parameter.
- Line 69 of ReserveLogic.sol should say "debt" instead of "income".

Consider correcting all the erroneous docstrings in the codebase, and documenting all functions (and their parameters) that are part of the contracts' public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well, following the Ethereum Natural Specification Format (NatSpec).

Update: *This issue has been fixed through several pull requests. In order those are PR#219, PR#334, PR#335, PR#336, PR#340, and PR#406.*

[N09] Mismatch between borrow approval and max amount to repay

In the context of debt tokens, a user can give delegation approval of uint256.max to any other user. However, if a user has that allowance and tries to repay a borrow using the total allowance amount, the transaction will definitely fail.

To improve consistency, consider either allowing a delegator to use such allowance or avoid setting the allowance to the maximum supported value.

Update: *Intended behaviour. In the words of the Aave team: "repay prohibits that because if we don't, the repayer on behalf could be frontrun by the borrower (if the repayer repays on behalf using uint256.max and*

the borrower sees the transaction incoming he could frontrun the repayer by borrowing more, the repayer would end up repaying more than expected, especially if he preemptively approved uint256.max)".

[N10] Typos

There are a few typos in the code base, e.g.

- ReserveLogic L227 `The reserve reserve to be updated` should be `The reserve to be updated`
- ReserveLogic L112 comment `The amount to accumulate` should be `accumulate`

Consider fixing these typos and review the entire code base looking for any other occurrences we might have missed.

Update: *Fixed in PR#371.*

[N11] Unclear bitmask docstrings

The `ReserveConfigurationMap` is a struct that stores an `uint256 data` inside. This `data` is the result of several bitmask operations which are better described in `ReserveConfiguration`.

There we can see which bits are assigned to which values. Among those, bits 61, 62, and 63 are reserved and not used but bits 253, 254, 255 and 256 are not used nor reserved, and they come right after the `debt ceiling` variable.

To improve docstrings and readability, consider properly documenting those bits whether they are reserved for later developments or not.

Update: *Fixed in PR#219.*

[N12] Unclear bridge functionality

When bridging tokens between networks, aTokens are burned in the source network and minted in the destination network. The `backUnbacked function` of the `Pool` contract is restricted to be called by the bridge through the `onlyBridge` modifier. This function actually moves tokens out from the bridge contract. Since the bridge contract is out of the scope of this audit, our team was not able to analyze how allowances are managed, but we recommend making sure that the bridge correctly handles the pool approval as spender before the `backUnbacked` function is triggered.

Update: *Acknowledged. In the words of the Aave team: "This is related to L02. The implementation is left minimal on purpose. It's up to the bridge implementation to handle the approval logic properly. Governance will validate the correct implementation of the functionality before giving access to it".*

[N13] Unnecessary imports

Throughout the codebase, there are interfaces that are being imported but never used in their corresponding contracts. Some examples that we have found are:

- In `SupplyLogic.sol`:`IStableDebtToken`, `IVariableDebtToken`, and `IFlasloanReceiver`.
- In `ReserveLogic.sol`: `IAToken`.
- In `LiquidationLogic.sol`: `VersionedInitializable`, and `Errors`.
- In `BorrowLogic.sol`: `Errors`.

Consider removing these and any other unused imports.

Update: *Fixed in PR#375.*

[N14] Unused restricted functions

The `upgradeTo`, `admin` and `implementation` functions of the `BaseImmutableAdminUpgradeabilityProxy` contract are restricted in access to be called exclusively by the `PoolAddressProvider` contract, which deploys the proxy in line 155 and sets itself as the admin. However, the latter two are not being called anywhere by the `PoolAddressProvider` contract.

Additionally, the `admin` function is a duplicate of the `ADMIN` public getter generated by the default public visibility of the `ADMIN` variable.

To improve code quality and size, gas consumption, and also readability, consider refactoring this code to avoid having duplicated getters, functions which are not callable by anyone, and that the implementation address can be retrieved easily without having to check the contract's storage.

Update: *Visibilities in the `PoolAddressesProvider` and `BaseImmutableAdminUpgradeabilityProxy` have been fixed in PR#379 while restrictions in access has been maintained as it is. For this the Aave team has provided the following acknowledge statement: "The current code allows for the implementation contract to have the functions 'admin' and 'implementation' as well. While the current implementation does not contain these functions, a future implementation could, therefore acknowledging this, and leaving this functions as it is".*

Conclusions

Two critical and three high severity issues were reported along with other issues of lower severity. We strongly recommend the Aave team fix these issues, refactor the code where necessary, and add unit tests to cover at least the most severe issues. Even if most of the issues require simple changes and fixes, we always recommend submitting the code for another round of auditing after all the fixes are applied. Lastly, technical documentation is still not mature. For this reason, we suggest the Aave team writes and publishes exhaustive documentation for this new version of the protocol as done for previous versions.

Update: *The Aave team acknowledged and fixed many of the reported issues. Fixes have been reviewed by the auditors and specific update statements have been provided. We are glad to see that the Aave team have been also working on improving technical documentation and improving the overall redability and quality of the codebase.*