Aave Protocol Audit

JANUARY 15, 2020 | IN SECURITY AUDITS | BY OPENZEPPELIN SECURITY

The Aave team asked us to review and audit a pre-production version of their protocol. We looked at the code and now publish our results.

The audited commit is `1f8e5e65a99a887a5a13ad9af6486ebf93f57d02` and all Solidity contracts in the aave-tech/dlp/contracts/contracts folder were in scope. Note that all links to the project's code in this audit report refer to a private repository, thus are only accessible to Aave's development team.

Before moving to the full list of issues found in the project, some introductory remarks about the project's current status are in order.

Project status

With the Aave protocol, Aave set out to implement a decentralized on-chain lending platform based on a “pool strategy” (instead of peer-to-peer lending). Lenders become liquidity providers by depositing cryptocurrencies in a pool, from which borrowers take out loans after having placed enough collateral. Interest rates are calculated fully on-chain, algorithmically deduced from the state of the available pools. Novel features of the protocol include variable and stable-rate loans (with the ability to switch between them) and flash loans – a type of loan taken and repaid in the same transaction.

The Aave team is aware that the audited version of the code base is a work-in-progress and not ready for production. In view of the project’s maturity, this first security audit round should be taken as the initial step forward in the way to reach the highest levels of code quality and robustness demanded by systems intended to handle large sums of financial assets. We identified numerous opportunities for improvement in the project’s documentation, test suite, and code itself, which are highlighted throughout the report. They will require not only specific patches in several code segments, but also great efforts in terms of testing and documentation. While these issues could be considered symptoms of the inherent difficulty of building a sustainable complex financial system, by no means are they to be taken lightly. Further security reviews of the entire protocol are in order, which along with our recommendations in this report, should help bring the project to a production-ready state.

Privileged roles

The Aave team currently administers all aspects of the protocol to decide which assets can be loaned and how price feeds and market rates are obtained. They also control various economic parameters, such as the size of the incentive used to encourage third parties to liquidate under-collateralized loans. All sensitive actions that owners / privileged roles can carry out (the most sensitive ones listed below) are instant and forced, with no opt-in nor opt-out mechanism for users of the protocol.

The owner of the `LendingPoolParametersProvider` contract can:

- Update the maximum fixed rate borrow size.
- Update the delta between the current fixed rate and the user fixed rate at which any borrow position can be rebalanced.

The owner of the `FeeProvider` contract can:

- Update the fee charged in all loans (called “origination fee”).
- Update the address that receives all fees.

The owner of the `LendingPoolAddressesProvider` (if functions were marked with `onlyOwner`, see issue **“[H01] Lack of access controls”**) can:

- Update the address of the `LendingPool`, `DefaultReserveInterestRateStrategy`, `LendingPoolCore`, `LendingPoolConfigurator`, `LendingPoolLiquidationManager`, `LendingPoolDataProvider`, `LendingPoolParametersProvider`, `FeeProvider` and `NetworkMetadataProvider`

contracts. In other words, the owner is allowed to modify the whole logic of the protocol.
- Update the address of lending rate and price oracles.
- Update the address of the "LendingPool manager" role.

The "LendingPool manager" role, set in the `LendingPoolAddressesProvider` contract, can (regardless of the current state of the protocol):

- Enable / disable borrowing on any reserve.
- Enable / disable use of reserves as collateral.
- Enable / disable borrowing at a fixed-rate in any reserve.
- Activate / deactivate reserves.
- Update a reserve's liquidation threshold, liquidation bonus and interest rate strategy (*i.e.* how interest rates are calculated).

These decisions and parameters can significantly affect the usefulness and safety of the system. Although the Aave team plans to grant these privileges to a governance system that will watch over the protocol in the near future, it is unclear whether a single externally owned account or a multisig will represent these roles at the time of audit. As for now, it requires users to fully trust the Aave team with these privileged roles before the governance system is introduced.

Oracles

The Aave protocol relies on two different oracles that feed rates and prices to the system.

The first is the Lending Rate Oracle, which should provide information on the actual market rates of other lending platforms. This oracle should compute an average market lending rate, based on the lending rate and borrowing volume of each external platform. Interaction with the Lending Rate Oracle can be seen in the `calculateInterestRates` function of the `DefaultReserveInterestRateStrategy` contract.

The second oracle is the Price Oracle, which should provide an asset's price (in Ether) when queried. Interaction with this oracle can be seen when:

- Borrowing
- Calculating user's data across reserves
- Calculating available collateral to be liquidated
- Determining whether a decrease in a user's reserve balance is allowed.

We must state that none of these oracles were included in this audit's scope, so they were assumed to be uncompromised, reliable and always available. For further reference on how trusting on-chain decentralized oracles can affect lending platforms, read an insightful article by samczsun.

Following, we present the full list of findings.

Audit Update

The Aave team applied several fixes based on our recommendations. We address below the fixes introduced as part of this first audit. Note that some of the original issues reported have been identified as non-issues, but are still kept for completeness. Some final remarks after reviewing the fixes:

- We only reviewed specific patches to the issues reported. The code base underwent unrelated changes we have not audited, and can be reviewed in depth in a future round of auditing.
- As we always recommend to all audited projects, the development team should strive to keep high standards in their software development life cycle with practices of continuous integration testing and mandatory peer-reviews. This greatly adds to the project's overall quality and prevents regression errors such as the one reported in **"[M11] Miscalculation of requested borrow amount in ETH"**.
- Before the code is open-sourced and the project launched, we encourage the Aave team to add a "Security" section in the main README of the project, including instructions for the responsible disclosure of any security vulnerabilities found in the project.

Critical severity

[C01] Users can steal funds

Whenever users deposit assets into a particular reserve, they choose whether the assets can be used as collateral against future loans, in addition to earning interest. This decision is stored as a flag in a `UserReserveData` object that handles the association between a particular account and reserve contract. However, if the user already has deposits in the same reserve, the existing flag is simply overwritten. This can be leveraged by malicious users to unlock and withdraw any collateral, even if it is required to secure a loan.

The identified attack vector goes as follows:

1. A user deposits assets into a reserve with the `_useAsCollateral` flag set to true, receiving aTokens in exchange.
2. They borrow assets from a different reserve, using the original deposits as collateral. At this stage, they cannot transfer their `aTokens`, since the `balanceDecreaseAllowed` function would return `false`, causing the `isTransferAllowed` check to fail.
3. Next, they deposit any number of assets (even zero) into the initial reserve with the `_useAsCollateral` flag set to `false`.
4. Now the user can successfully transfer their `aTokens` to a fresh account, since the `balanceDecreaseAllowed` function will bypass the insolvency check and return `true`.

5. They can now redeem (or sell) the `aTokens` to receive their original collateral. At this point they have taken a loan with no collateral, effectively stealing from the lending pool.

Consider preventing users from setting the `_useAsCollateral` flag to `false` when their existing deposits in the reserve are required to secure an outstanding loan. Once the fix is applied, related unit tests are in order to avoid reintroducing this critical issue in future modifications to the code base.

Update: *Fixed in MR#38. Users can no longer choose whether deposits are considered collateral. Instead, when the user's balance in the reserve is zero, the* `deposit` *function will now default to mark the next deposit in the reserve as collateral. Users can opt-out by calling the* `setUserUseReserveAsCollateral` *function of the* `LendingPool` *contract, which performs the appropriate solvency checks. We suggest better documenting this scenario to avoid unexpected behaviors. Additionally, the event* `ReserveUsedAsCollateralEnabled` *should be emitted in the* `deposit` *function after calling the* `setUserUseReserveAsCollateral` *function of the* `LendingPoolCore` *contract.*

[C02] Borrowers can avoid liquidation

When a loan is liquidated, the reduction in the remaining principal and the corresponding reduction in the total borrows (either fixed or variable) is calculated by subtracting the accrued interest from the amount to be repaid.

Conceptually, this is a combination of two separate operations: the accrued interest is added to the principal when constructing the new loan and then the amount repaid is subtracted. However, combining them into a single operation will cause the transaction to revert whenever the repayment is less than the accrued interest.

In addition to preventing valid repayments, this behavior could be exploited by borrowers to prevent liquidation.

Since the size of each liquidation transaction is restricted by the amount of collateral that can be recovered from the specified reserve, a borrower could spread their collateral across many different assets in order to ensure the maximum liquidation amount is lower than their accrued interest.

Alternatively, the liquidation amount is also restricted by the protocol's close factor, which means a user could simply allow their loan to grow until the interest exceeds this threshold. In practice, this will likely take years to occur.

In either case, when a borrower cannot be liquidated they no longer have any incentive to remain collateralized.

Consider updating the principal and total borrows variables in two independent steps that account for the accrued interest and the loan repayment respectively.

Update: *Fixed in MR#56 and MR#58. The principal and total borrows are updated in two steps.*

[C03] Deposits not marked as collateral can still be liquidated

When a user deposits an asset into the lending pool, they can choose whether the asset functions as collateral by means of the `_useAsCollateral` flag. While the expected behavior is that only assets marked as collateral can be liquidated, this restriction is not enforced.

The issue is due to a logic flaw in the `liquidationCall` function. This function should require that the reserve is enabled as collateral and the user has marked that reserve as collateral. However, the conditional statement in lines 92 and 93 is erroneous. Consequently, assuming the other liquidation requirements hold, `liquidationCall` succeeds when either:

- The reserve `_collateral` is enabled as collateral (regardless of the user preference)
- The reserve `_collateral` is not enabled as collateral and the user did not mark it as collateral

Consider modifying the conditional statement in lines 92 and 93 to `core.isReserveUsageAsCollateralEnabled(_collateral) &amp;&amp; core.isUserUseReserveAsCollateralEnabled(_collateral, _user);`. Afterwards, implementing thorough related unit tests is highly advisable.

Update: *Fixed in MR#59. Specific unit tests covering this case are still missing.*

[C04] Rogue borrower can manipulate other account's borrow balance

The `rebalanceFixedBorrowRate` function of the `LendingPool` contract is intended to allow anyone to rebalance the fixed interest rate of a borrower when certain requirements are met. All the caller needs to specify is the reserve (in the `_reserve` parameter) and the borrower's account to be rebalanced (in the `_user` parameter).

When querying the borrow balances, the function calls the `getUserBorrowBalances` function mistakenly passing `msg.sender` as the argument. Consequently, the `compoundedBalance` and `balanceIncrease` local variables will hold the caller's borrow balances, and not those of the account to be rebalanced (*i.e.* the address in `_user`). From then on, `balanceIncrease` is used to update the reserve's data. It is fundamental to note that on line 514 the `increaseUserPrincipalBorrowBalance` function increases the `_user`'s borrow balance by `balanceIncrease`.

Any rogue borrower whose `compoundedBalance` is greater than zero can leverage this critical vulnerability to manipulate another borrower's borrow balance. An attacker holding a large borrow can call `rebalanceFixedBorrowRate` with the victim's address as the `_user` parameter. Thus increasing the victim's principal borrow balance.

The attacker can further benefit by particularly targeting accounts close to being liquidated. The increase in their borrow balance would effectively push victims into a "liquidatable" position, allowing the attacker to liquidate them. To prevent front-runs from other liquidators, this attack can be conducted in a single atomic transaction through a malicious contract that first increases the victim's borrow balance and then liquidates them.

A second attack vector allows rogue borrowers to distort their accrued interest. They can call the `rebalanceFixedBorrowRate` function passing their account in the `_user` parameter, from an account with a very small borrow. This would effectively update their own `lastUpdateTimestamp`

without accruing nearly as much interest as they should actually accrue.

To prevent malicious borrowers from exploiting this vulnerability, consider replacing `msg.sender` with `_user` as the argument passed to the `getUserBorrowBalances` function. Please note that this issue is closely tied to critical issue **"[H09] Fixed-rate loans can be repeatedly rebalanced"** and high severity issue **"[H06] It is impossible to rebalance another account's fixed borrow rate"**.

Update: *Fixed in MR#61.*

[C05] Loans of Ether cannot be repaid

Any user that has taken out a loan in the Aave protocol should be able to repay it by calling the `repay` function of the `LendingPool` contract. However, a flaw in this function makes it (typically) impossible to repay an Ether loan.

During the process of repaying an Ether loan the caller is expected to send Ether along with the transaction. That Ether should be split in two, to first pay the origination fee and then to pay down the loan. These operations are executed in lines 375 and 412. Both transfers are sent to the `LendingPoolCore` contract, which forwards the fee to a "fee collection address" during the first transfer. Even though the `LendingPoolCore` contract is the destination in both cases, it expects to receive the repayment component in the second transfer. Consequently, the `LendingPool` contract will (typically) have insufficient Ether to complete the second transfer, which will cause the entire transaction to revert.

Consider either returning the excess ether to the `LendingPool` contract in the `transferToFeeCollectionAddres` function or recognizing if Ether was sent with the `repay` function and forwarding the exact amounts in both transfers.

Update: *Fixed in MR#48. However, note that an erroneous inline comment has been introduced stating "sending the total msg.value if the transfer is ETH". In this case, when the function executes an ETH transfer, only the amount* `msg.value.sub(vars.originationFee)` *is transferred.*

High severity

[H01] Lack of access controls

Contracts `LendingPoolAddressesProvider` and `NetworkMetadataProvider` include several setter functions that allow the caller to change addresses of fundamental contracts of the protocol and modify system-wide parameters. The main purpose of these functions is to allow upgrades in the protocol's logic. However, none of these functions currently implement any access control mechanisms, thus allowing anyone to execute them. Affected functions are listed below.

In `LendingPoolAddressesProvider` contract: `setLendingPool`, `setInterestRateStrategy`, `setLendingPoolCore`, `setLendingPoolConfigurator`, `setLendingPoolManager`, `setLendingPoolDataProvider`, `setNetworkMetadataProvider`, `setLendingPoolParametersProvider`, `setPriceOracle`, `setLendingRateOracle`, `setFeeProvider` and `setLendingPoolLiquidationManager`.

In `NetworkMetadataProvider` contract: `setBlocksPerYear` and `setEthereumAddress`.

The Aave team acknowledges the lack of access controls for these extremely sensitive features (with inline comments in the code that read `// TODO: add access control rules under DAO`), and is set out to build a governance system that would be the only actor allowed to trigger them. We must highlight that such a governance system is currently not implemented and was left entirely out of the scope of this audit.

Regardless of the nature of the access control mechanisms chosen, it must be stressed that the system should not be put in production without the necessary restrictions (and related unit tests) in place.

Update: *Fixed in MR#52 by adding the missing `onlyOwner` modifiers. According to the Aave team:*

> "Ownership of these contracts will be assigned upon deployment to the correspondent governance infrastructure".

[H02] Anyone can disable the flash loan feature

The Aave protocol allows borrowers to take out a special type of loan that must be repaid (along with a fee) in the same transaction. This functionality is implemented in the `flashLoan` function of the `LendingPool` contract. Among the preconditions checked before effectively emitting the loan, the function validates in a `require` statement that the available liquidity matches the actual balance of the `LendingPoolCore` contract. From the comment above the check, this is done "for added security". Yet, this `require` statement allows anyone to permanently disable the entire flash loan mechanism.

All a malicious user would need to do to effectively conduct the attack would be to send a small amount of assets to the `LendingPoolCore` contract. As a consequence, the balance of the contract would be increased without increasing the available liquidity of the protocol. The mismatch would disable the flash loan feature for a reserve, as the `require` statement in line 624 would inevitably fail. It should be noted that to disable the flash loan for the Ether reserve, the attacker must send the ETH from a contract.

Since the validation in line 624 of `LendingPool.sol` does not appear to be mandatory to ensure the correct behavior of the `flashLoan` function, consider removing it.

Update: *Fixed in MR#53.*

[H03] Deactivated collateral reserve can be used in liquidation

All asset reserves in the Aave protocol can be deactivated by a high-privileged account. When a reserve is deactivated, no actions should be allowed to be executed over it. This validation is implemented by means of the `onlyActiveReserve` modifier.

The `liquidationCall` function in the `LendingPool` contract *does* use this modifier, but only to validate that the reserve specified in the `_reserve` argument is active. However, no similar validations are in place for the reserve address passed in the `_collateral` argument. As a consequence, a

liquidation can be executed over a collateral reserve that is expected to be deactivated and suffer no modifications at all. Furthermore, this issue is aggravated by the fact that the `LiquidationCall` event does not log which collateral reserve is being affected by the liquidation, hindering off-chain clients' task of effectively tracking modifications to a deactivated collateral reserve.

Consider preventing modifications to a deactivated collateral reserve via the `liquidationCall` function. This can be achieved by validating that the address passed in the `_collateral` argument corresponds to an active reserve.

Update: *Fixed in MR#64*.

[H04] Liquidators cannot reclaim underlying asset when there is enough liquidity

When calling the `liquidationCall` function to liquidate a borrow position, liquidators can choose to get the collateral either by reclaiming the aTokens or the underlying asset. This is indicated by the `_receiveAToken` boolean parameter of the function, where `false` means the caller is willing to receive the underlying asset. In this scenario, the protocol will first attempt to validate whether there is enough liquidity in the collateral's reserve, which should be achieved by comparing the available liquidity and the maximum amount of collateral to liquidate.

However, as the comparison is inverted, the function will return an error when there is enough liquidity to proceed with the liquidation. Specifically, a `LiquidationErrors.NOT_ENOUGH_LIQUIDITY` error is returned when `currentAvailableCollateral >= maxCollateralToLiquidate`.

While this issue does not pose a security risk on its own, the erroneous comparison breaks an entire feature, preventing liquidators from liquidating a position and getting the underlying asset in exchange. It should be noted that this issue could have been prevented with more thorough unit testing, but only tests where `_receiveAToken` is `true` are currently included (see `LiquidationCall.spec.js`).

Consider inverting the mentioned comparison so that `liquidationCall` returns an error when `currentAvailableCollateral < maxCollateralToLiquidate`. Once the fix is applied, it is highly advisable to add related unit tests to ensure the behavior is expected. Moreover, the new tests should help prevent this issue from being reintroduced in future changes to the code base.

Update: *Fixed in MR#73*.

[H05] Maximum size of fixed-rate loans can be bypassed

The `borrow` function of the `LendingPool` contract attempts to set a hard cap on the size of fixed-rate loans. The limit depends on the reserve's available liquidity, and is implemented solely for fixed-rate loans. An attack vector that allows any borrower to bypass the limit has been identified. First, the borrower takes out an arbitrarily large loan at a variable rate. Afterwards, the borrower takes out a second loan on the same reserve, even for an amount of zero – but this time setting the interest rate mode to fixed. Effectively, the borrower obtained an arbitrarily large loan at a fixed rate. Note that this operation can be carried out by a smart contract in a single transaction. To prevent this particular issue, consider tightening the restrictions to switch from variable to fixed rate loans. Nonetheless, we understand that this issue should not be analyzed in

isolation. The simplicity of the attack vector described stems from larger shortcomings at the design level of the Aave protocol and how it handles fixed and variable rate loans. Refer to the "**[N03] Fixed interest rate loans feature is loosely encapsulated**" issue for more details.

Update: *Not an issue. In Aave's words:*

> "The goal of the maximum size enforcement on stable- rate loans is to avoid having borrowers taking liquidity at a rate that is too competitive. In fact, there is a fundamental difference in the two scenarios of a) taking a stable-rate loan directly and b) taking a variable-rate loan, and switching to stable. The difference is in the interest rate. In case a) in fact, without a hard cap on the loan size, borrowers are theoretically able to snatch the whole liquidity at a very competitive rate, as the stable rate only increases after the loan has been taken. In scenario b), on the other hand, the loan at variable rate can be arbitrarily big, as it poses no issues on the interest rate (variable rates increase as borrowers take more liquidity). In scenario b) also, when a user borrows at variable it might cause the stable rate to rise as well, especially if the loan is of a relevant size. Hence in this case a rate swap poses no issues, as the borrower would swap to the most recent, increased stable rate, rather than to the lower stable rate that he would have been able to benefit if he would have borrowed with a stable rate directly".

[H06] It is impossible to rebalance another account's fixed borrow rate

In the `rebalanceFixedBorrowRate` function of the `LendingPool` contract, the call to `core.updateUserFixedBorrowRate` is executed passing the `msg.sender` address as argument. As a consequence, successful rebalance calls will only ever update the fixed borrow rates of the caller, rather than the target address defined by the `_user` parameter. This breaks the intended feature of being able to rebalance other accounts.

Consider changing the `msg.sender` address used in line 537 to `_user`. Related issues that must also be taken into consideration are "**[C04] Rogue borrower can manipulate other account's borrow balance**" and "**[H09] Fixed-rate loans can be repeatedly rebalanced**".

Update: *Fixed in MR#61*.

[H07] Users cannot fixed-rate borrow from a reserve no longer containing their collateral

To prevent abuses in the protocol, borrowing at a fixed rate (from the same reserve where the borrower deposited collateral) is only allowed if the amount being borrowed is greater than the collateral. This restriction is implemented within the `borrow` function on lines 216 to 221 of `LendingPool.sol`.

However, the restriction currently disallows users borrowing at a fixed rate from a reserve where they *previously* had collateral (but no longer do). After a user withdraws all collateral from a reserve, the system does not automatically toggle to `false` the `isUserUseReserveAsCollateralEnabled` flag. When the user attempts to borrow from that reserve, the condition in line 220 will fail, thus reverting the transaction. As a result, the user is unable to borrow from this reserve even if they are not currently holding any collateral in it.

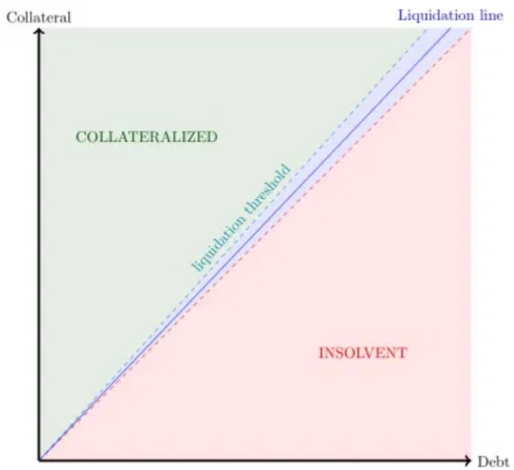
Consider programmatically toggling to `false` the `useAsCollateral` flag once a user has withdrawn all collateral from a reserve.

Update: *After applying a patch in MR#74, the Aave team correctly pointed out that we misinterpreted the function's behavior and this is not an issue. The fix will still remain in place, and the development team will include relevant test cases to programmatically confirm this is indeed not an issue.*

[H08] Counterproductive incentives

The Aave protocol requires all loans to be backed by more-than-100% collateral in other asset pools. To maintain this condition as the asset prices vary, there is a safety mechanism that allows anyone to repay loans that are at risk of becoming insolvent. Typically, this encourages arbitrageurs to improve the health of the loans but there are some conditions where the safety mechanism behaves counter-productively, and takes at-risk loans and actively pushes them into insolvency.

Consider a borrower with collateral in various markets worth a total of `C`, and a total debt worth `D`. They are located at point (`D`, `C`) in the following diagram:



After accounting for the distribution of collateral and the thresholds associated with each reserve, they are assigned a *liquidation threshold*. With these definitions, we can describe the important regions:

- Users in the green region above the liquidation threshold are fully collateralized. Their collateral exceeds their debt by an acceptable safety margin.
- Users in the red region are insolvent. Their debt is larger than their collateral and they no longer have any incentive to repay it. Users in this region can be a major problem for the system because they continue accruing interest while removing liquidity from the reserve, ensuring at least some aToken holders will be unable to redeem their assets.

- Users in the purple region are under-collateralized. They still have more assets than debt in the system, but the protocol deems them to be at risk of becoming insolvent.

When a borrower is under-collateralized, anyone can repay some of their debt in exchange for collateral at better-than-market rates. This mechanism is parameterized by two global values:

- Liquidation close factor percentage: the maximum fraction of the original loan that can be liquidated (currently set at 50%)
- Liquidation bonus `L`: how much collateral do arbitrageurs receive (as a multiplier of the amount paid)

The liquidation line on the diagram has slope `L` and passes through the origin. When an arbitrageur repays `x` debt, they receive `x*L` collateral from the borrower. In other words, they force the borrower to the bottom-left of the diagram on a trajectory parallel to the liquidation line. The crucial observation is that this process can never cause a borrower to cross the liquidation line. Borrowers above the line are liquidated until they are fully collateralized, but borrowers below the line are actually pushed further towards insolvency by the liquidation process.

Whenever the leftover debt is more than the leftover collateral after a liquidation event, the user is insolvent. If we additionally denote the close factor as `Cf`:

The maximum liquidation amount would be `Cf * D` with the remaining debt being `(1 - Cf) * D`.

The value of collateral the liquidator receives (including the bonus) would be `Cf * D * L`, so the remaining collateral becomes `C - Cf * L * D`.

The would become insolvent if the remaining debt is greater than the remaining collateral. This occurs when `( 1 - Cf ) * D &gt; C - Cf * L * D`, or rearranging, `D / C &gt; 1 / ( 1 + Cf * (L - 1))`. If we assume a liquidation bonus of 1.05 and a close factor of 0.5, then **when a borrower's debt rises to at least 1 / 1.05 = 95.2% of the value of their collateral the liquidations start increasing their debt-to-collateral ratio and at 97.6% it is possible for a liquidation to force them into insolvency**.

Ideally, they would have already been liquidated before they cross the threshold, but this may not happen depending on various factors outside of the protocol, such as the speed of price changes, the liveness of the oracles, the congestion of Ethereum and the responsiveness and liquidity of arbitrageurs. To this end, it may prove useful to run off-chain monitoring of all above mentioned parameters, along with auto-liquidation mechanisms, to address the risk as early as possible.

Nevertheless, in the event that the borrower crosses the liquidation line, the protocol can no longer sustain the same bonus, since each liquidation event now increases the risk to the protocol. This observation has the unfortunate consequence that the incentive for arbitrageurs to liquidate a risky loan is weaker when it is more crucial. Even so, a weakly productive incentive (or indeed no incentive at all) is still preferable to a counterproductive one. The reduced incentive may be compensated by increasing the close factor for the most at-risk loans or perhaps even allowing arbitrageurs to liquidate 100% of the loan in exchange for 100% of the remaining collateral.

Consider adjusting the parameters in the liquidation mechanism to ensure it is always pushes borrowers and the protocol towards solvency.

Update: *The Aave team acknowledges the issue:*

> "We consider this issue as a part of the dynamics of the protocol, and there is no easy solution as reducing or removing the liquidation incentives in case of extreme undercollateralization would remove the incentive for external liquidators to execute the liquidation, which would still bring insolvency. In the future, we plan to challenge this issue with i) Automated liquidation through decentralized exchanges, to be implemented together with external liquidation and ii) An insurance fund to cover liqudation risk."

[H09] Fixed-rate loans can be repeatedly rebalanced

Any user's fixed-rate loan can be rebalanced if the rate falls outside an acceptable range. The lower boundary of this range is the current liquidity rate, while the upper boundary is intended to be a configurable percentage higher than the reserve's fixed borrow rate.

However, the upper threshold is set below the reserve's fixed borrow rate. This means that new loans will start outside the acceptable range and will remain that way after rebalancing so that "fixed-rate loans" are actually vulnerable to any change in the market rate.

Consider redefining the upper threshold to be higher than the reserve rate (by the configurable down-rate delta parameter) rather than lower by that percentage. Related issues that must be taken into consideration are **"[H06] It is impossible to rebalance another account's fixed borrow rate"** and **"[C04] Rogue borrower can manipulate other account's borrow balance"**.

Update: *Fixed in MR#44. This issue was originally labeled as Critical since it implies fixed-rate loans can be made to follow the variable rate. After discussing with the Aave team, it has been downgraded to High. In Aave's words:*

> "Although the logic flaw exposed caused unwanted behavior that needed fix, this issue didn't actually pose any security risk for the protocol, and is actually ending up in a potentially acceptable use case".

Medium severity

[M01] Fee-less loans

The `calculateLoanOriginationFee` function of the `FeeProvider` contract is in charge of calculating the origination fee for a loan of the specified amount. To do so, it multiplies the given amount by the `originationFeePercentage` – a hardcoded value set during construction to `0.0025 * 1e18`.

Due to how the `wadMul` multiplication works in this particular case, `calculateLoanOriginationFee` can return 0 for amounts greater than 0. Such unexpected behavior would allow for a user to take loans that would not pay an origination fee. In particular, all loans with an amount lower than

200 will be granted without accounting for a fee (see this simple mathematical proof for validation).

Should loans be expected to always charge a fee, consider implementing the necessary validations in `calculateLoanOriginationFee` so that the transaction reverts in case the calculated fee is zero. Otherwise, consider clearly documenting this behavior to prevent unexpected outcomes. Either way, it is highly advisable to implement thorough unit tests related to this feature, as no unit tests were found for the `calculateLoanOriginationFee` function.

Update: *Fixed in MR#75. Now all loans that would not pay a fee are rejected.*

[M02] Anyone can open a flash loan for an unprotected receiver

The `flashLoan` function in the `LendingPool` contract allows anyone to execute a flash loan on the Aave protocol. The caller can specify in the `_receiver` argument any contract address that implements the `IFlashLoanReceiver` interface.

Should the receiver contract not implement the necessary validations to identify who originally triggered the transaction, it may be possible for an attacker to force any `IFlashLoanReceiver` contract to open arbitrary flash loans in the Aave protocol that would inevitably pay the corresponding fees. This can potentially drain all funds (ETH or tokens) from the vulnerable contract implementing the `IFlashLoanReceiver` interface. It should be highlighted that the provided `FlashLoanReceiverBase` contract does not include any security measure, nor warning documentation, to prevent this issue.

To reduce the attack surface, it is advisable to modify the `flashLoan` function so that only the receiver of the loan can execute it. If opening flash loans on behalf of `IFlashLoanReceiver` contracts is an intended feature, then consider adding user-friendly documentation to raise awareness, along with sample implementations showcasing how to defend from attackers that attempt to open flash loans on behalf of unprotected `IFlashLoanReceiver` contracts.

Update: *The Aave team understands this is not an actual security issue of the Aave protocol, but will still provide enough documentation to raise awareness in developers. In Aave's words:*

> "We believe that this should not be reported as a security issue of the protocol as it doesn't actually show any risk for the protocol itself, but rather a potential risk on an unsafe implementation of a `IFlashLoanReceiver`. Moreover, the solution proposed greatly diminishes the possibility of avoiding front-running by using a multiple contracts strategy, which was the reason why the `flashLoan` method does not place any safety check on the caller of the function. As the potential security issue might not be immediately visible to a developer of a flash loan receiver, we will make sure to properly document this issue and provide the developers with adequate code samples."

[M03] Incorrect refund address during repay

The `repay` function allows repayment of a loan on behalf of other accounts. In a scenario where the caller overpays Ether on behalf of another account, the function will refund the excess Ether to the target address (*i.e.* the address passed in the `_onBehalfOf` parameter) and not the caller.

Whenever a loan is overpaid with Ether, consider returning all excess Ether to the actual repayer and not the account on whose behalf the loan is being repaid.

Update: *Fixed in MR#76. All excess Ether is now returned to the actual repayer (i.e. the caller).*

[M04] Borrower cannot partially repay interest of a loan

The `repay` function of the `LendingPool` contract allows a borrower to repay a loan in a specific reserve. However, it is currently impossible for the borrower to partially repay the interest of a loan. This is due to the fact that whenever the `borrowBalanceIncrease` is greater than the `paybackAmountMinusFees`, the transaction will be reverted.

Should this be the function's intended behavior, consider explicitly documenting it in docstrings. Otherwise, consider implementing the necessary logic to prevent `repay` from reverting when `borrowBalanceIncrease` is greater than the `paybackAmountMinusFees`, allowing borrowers to partially repay their loan's interest.

Update: *Fixed in MR#77.*

[M05] Push-payments pattern may render ETH deposits impossible to redeem

When redeeming ETH deposits, the `LendingPoolCore` contract follows the push-payments pattern to return the deposited Ether. Such a pattern, implemented using the `transfer` function, has some notable shortcomings when the redeemer is a smart contract, which can render ETH deposits impossible to redeem. Specifically, the redeem will inevitably fail when:

- The redeemer smart contract does not implement a payable fallback function.
- The redeemer smart contract implements a payable fallback function which uses more than 2300 gas units.
- The redeemer smart contract implements a payable fallback function which needs less than 2300 gas units, but is called through a proxy that raises the call's gas usage above 2300.

Note that the upcoming Istanbul fork can further aggravate this issue, since payable fallback functions that today *do not* consume more than 2300 gas units may effectively go above the threshold after the fork. This is due to EIP 1884.

To prevent unexpected behavior and potential loss of funds, consider explicitly warning end-users about the mentioned shortcomings to raise awareness *before* they deposit Ether into the Aave protocol. Additionally, note that the low-level call `call.value(_amount)("")` can be used to

transfer the redeemed Ether without being limited to 2300 gas units. Risks of reentrancy stemming from the use of this low-level call can be mitigated by tightly following the "Check-effects-interactions" pattern and using OpenZeppelin's `ReentrancyGuard` contract.

Update: *Partially fixed in MR#78. We consider it "Partially fixed" since the fact that redeemer smart contracts must implement a payable fallback function is still not documented. As suggested, functions* `transferToUser`, `transferToFeeCollectionAddress` *and* `transferToReserve` *replaced the call to* `transfer` *with a low-level call. Furthermore, we have our reservations around hard-coding a gas limit on Ether transfers, but the fix works as intended. In Aave's words:*

> "As a further safety check, we decided to limit the maximum gas consumption of the receiver default function to 50.000, which is below the minimum cost of any user-oriented function on the `LendingPool` contract. This limit might be revised in the future and/or moved to a global configuration parameter".

[M06] Successful redeem of aTokens may not pay assets in exchange

The `redeem` function in the `AToken` contract allows holders of aTokens to redeem them for the underlying asset. Assuming all necessary conditions for a successful redeem are met, the expected behavior would be that for every token redeemed a certain amount (given by the exchange rate) of the underlying asset is taken from the pool and transferred to the redeemer.

However, when the amount of aTokens to be redeemed is lower than the amount of aTokens the system exchanges per unit of the underlying asset, the transaction does not revert. In such cases, the system would still take the aTokens, burn them, and transfer 0 units of the underlying asset to the redeemer. Therefore, the caller would lose all redeemed aTokens and the total supply of aTokens would be reduced without modifying the underlying asset reserve's liquidity.

To prevent unexpected behaviors that may lead to losses of aTokens, consider reverting the transaction when the amount of redeemed tokens is not enough to receive at least one unit of the underlying asset in exchange.

Update: *Fixed in MR#79. The transaction is now reverted when the amount of redeemed tokens is not enough to receive at least one unit of the underlying asset in exchange.*

[M07] Lack of event emission after rebalancing fixed borrow rate

The `rebalanceFixedBorrowRate` function in the `LendingPool` contract allows anyone to rebalance the fixed interest rate of a user under specific circumstances. However, the function does not emit an event after the rebalancing is executed.

As such a sensitive change is of utter importance to users, consider defining and emitting an event in order to notify clients about it.

Update: *Fixed in MR#80. A* `RebalanceStableBorrowRate` *event has been defined and is now emitted after successful rebalances of stable-rate loans.*

[M08] Interest may compound unpredictably

In the Aave protocol, loans' interest is compounded after relevant "interest accruing" transactions occur (with a difference between fixed-rate and variable-rate loans, reported in **"[N02] Fixed-rate loans may never compound"**). Between two such transactions, the system uses a simple interest rate model.

The code is designed to accrue interest as frequently as possible, but this requirement expands the responsibility of accruing interest into otherwise unrelated functions. Additionally, the size of the discrepancy between the computed and theoretical interest will depend on the volume of transactions being handled by the Aave protocol, which may change unpredictably.

To improve predictability and functional encapsulation, consider calculating interest with the compound interest formula, rather than simulating it through repeated transactions. The `modexp` precompile may assist in lowering gas fees. Alternatively, consider informing users that the protocol's interest rates are merely *estimations* rather than exact rates.

Update: *The Aave team acknowledges this issue:*

> "We acknowledge this issue, as also strictly correlated with N02. As a result, we will evaluate before the mainnet release what will be the implementation cost and the benefits of switching to a compounded interest rate formula, and eventually modify the implementation accordingly."

[M09] Sensitive mathematical operations are not explicitly documented

Intending to make the platform as transparent as possible, the Aave team has implemented most of the calculations the Aave protocol relies on in their smart contracts. Such calculations usually entail complex arithmetic operations over balances, timestamps, rates, percentages, prices, decimals, among others, that are measured in several different units. It is of utmost importance for such operations to work flawlessly, considering that the Aave protocol is set out to handle large amounts of valuable assets, and any error may cause outstanding financial losses. However, such sensitive operations were found to be sparsely documented, the most important shortcoming being the lack of explicit units for each term involved.

This lack of explicit units for state variables, parameters and return values greatly hindered the auditing process. While attempts to validate all calculations spread throughout the code base were made, still the manual process was unreliable and error-prone. Mapping formulas to the provided whitepaper was not straightforward either, because there are many mismatches – as reported in **"[L18] Whitepaper issues"**. Assessing for correctness becomes difficult when there is no way to straightforwardly understand the units used in each calculation, regardless of their simplicity. These are the reasons why we are listing this issue with Medium severity.

Great efforts must be made in term of documenting calculations and explicitly stating all units of the terms involved. This should greatly improve the readability of the code, which should add to the platform's transparency and the users' overall experience. As the process of manually auditing

all sensitive arithmetic operations has been proven hard-to-follow, unreliable and potentially error-prone, thorough unit testing of all critical calculations is in order to programmatically ensure that the code's current behavior is expected.

Update: *Partially fixed. The most recent whitepaper shared with us is significantly clearer. However, the units of the variables in the* `CoreLibrary.sol` *structs are still undocumented.*

[M10] Missing test coverage report

There is no automated test coverage report. Without this report it is impossible to know whether there are parts of the code never executed by the automated tests; so for every change, a full manual test suite has to be executed to make sure that nothing is broken or misbehaving. High test coverage is of utter importance in projects like the Aave protocol, where large sums of valuable assets are expected to be handled safely and bugs can cause important financial losses.

Consider adding the test coverage report, and making it reach at least 95% of the source code.

Update: *Acknowledged, and fix in progress. The Aave team was working on setting up coverage on a separate branch during our audit. Given the tools are still immature and unstable, some problems arised in the process which are now being solved by customizing the coverage tools to Aave's needs. The team is fully aware of the importance of high test coverage, and is striving to reach at least 95% coverage before launch.*

[M11] Miscalculation of requested borrow amount in ETH

Note for the reader: This issue was detected during our review of the fixes for the first audit round. The specific commit where the issue was introduced is `8521bcd`*, which was not present in the audited commit. It must be noted that the PR that finally merged this commit to the* `master` *branch was created and merged by the same author, without any kind of peer-review nor CI testing process. The commit to which we link in the issue's description is the latest in the* `master` *branch at the moment of writing.*

To validate whether there is enough collateral to cover a borrow, the `borrow` function of the `LendingPool` contract first calculates how much ETH the amount borrowed represents. This calculation is intended to take into account the borrow fee paid by the borrower. However, the borrow fee is only calculated *after* it is first used. This means that when the requested borrow amount in ETH is calculated, the `vars.borrowFee` variable is zero. As a consequence, the calculated borrow amount in ETH will be lower than expected, inevitably lowering the actual amount of collateral needed in ETH to accept the borrow operation.

Consider refactoring the `borrow` function to first calculate the borrow fee, and only then using the `vars.borrowFee` variable for further operations. Note that this issue should have been caught if the `borrow` function and its calculations were thoroughly tested (as suggested throughout our original assessment).

Update: *Fixed in MR#116.*

Low severity

[L01] Minimum interest can be within one block

To lessen the impact of potential interest-free loans (refer to the first high-severity issue in our Compound audit), the Aave protocol adds 1 wei as "symbolic cumulated interest". However, this mechanism does not check whether time has passed, so it would accrue the symbolic interest in the same block as the loan.

Before adding the symbolic cumulated interest of 1 wei, consider adding a check to ensure that the last update's timestamp is different from the current's block timestamp.

Update: *Fixed in MR#81. The* `getCompoundedBorrowBalance` *function now checks the block timestamp before accruing symbolic interest.*

[L02] Truncation when casting Ray to Wad

The `rayToWad` function of the `WadRayMath` library truncates the input `Ray` value (with 27 decimal digits of precision) when casting it to a `Wad` value (with only 18 digits of precision). However, the other functions in the library round instead of truncating when discarding precision.

Consider creating a second function that rounds to the nearest `Wad`. This could be used in the `getCompoundedBorrowBalance` function of the `CoreLibrary` to return a slightly more accurate value, favoring accruing more interest for the protocol. The original `rayToWad` functionality in `underlyingAmountToATokenAmount` and `aTokenAmountToUnderlyingAmount` functions should be kept, as it favors rounding down (truncation) when calculating how many tokens to send *out* of the protocol.

Update: *Fixed in MR#82. The* `rayToWad` *function has been modified, and now always rounds to the nearest* `Wad`*. Note that the applied fix does not strictly follow our suggestion of having two separate functions.*

[L03] Conversion to aToken units implicitly assumes 18 decimals

The `underlyingAmountToATokenAmount` function of the `AToken` contract never takes into account the decimals of the overlying asset (*i.e.* the aToken). While it works under the assumption that all aTokens have 18 decimals, there is nothing in the code to ensure this assumption always holds.

This issue does not pose a current security risk, as the Aave developers are in control of how many decimals aTokens have. However, the `underlyingAmountToATokenAmount` function should programmatically enforce all conditions needed to operate normally, since dangerous unexpected behavior could arise otherwise.

To favor explicitness and prevent bugs in future modifications to the code base, consider modifying the `underlyingAmountToATokenAmount` function to explicitly account for the decimals of the overlying aToken. Otherwise, warning documentation should be included in the function docstrings. One alternative, less flexible, course of action is to programmatically enforce that all created aTokens have 18 decimals in the `AToken` contract constructor.

Update: *Fixed in MR#83. The `AToken` constructor now programmatically enforces all aTokens to have 18 decimals.*

[L04] Redundant underflow prevention

Function `decreaseUserPrincipalBorrowBalance` unnecessarily implements an underflow protection ensuring that `user.principalBorrowBalance` is greater or equal than `_amount`. This validation is already in place in OpenZeppelin SafeMath's `sub` function, so consider removing it.

Update: *Not an issue, as the code correctly follows the "Fail early and loudly" principle.*

[L05] Fixed-rate borrow can be rebalanced in inactive reserve

The `rebalanceFixedBorrowRate` function of the `LendingPool` contract does not validate whether the reserve passed as an argument is active. As a consequence, an inactive reserve can suffer unexpected changes in its parameters.

This issue should be disregarded if the behavior is expected. Otherwise, consider requiring that the reserve is active by means of the `onlyActiveReserve` modifier.

Update: *Fixed in MR#84.*

[L06] Collateral can be deposited in a reserve where usage as collateral is disabled

The `deposit` function of the `LendingPool` contract allows users to deposit assets as collateral (by setting the `_useAsCollateral` parameter to `true`) in reserves where usage as collateral may be disabled system-wise. Specifically, in reserves where `usageAsCollateralEnabled` is `false`.

This issue does not pose a security risk, as such deposits will not account for collateral when the reserve is disabled as collateral system-wise. Yet, such behavior can cause confusion in users. Consider reverting the transaction whenever a user attempts a deposit of assets as collateral (with the `_useAsCollateral` argument set to `true`) in a reserve where usage as collateral is disabled system-wise.

Update: *Not an issue. The Aave team understands this is the protocol's intended behavior:*

> "The dominant configuration parameter which determines first and foremost if a deposit can be used as collateral is the `usageAsCollateralEnabled`. The `_useAsCollateral` parameter was intended only as a user preference. [By fixing C01] The possibility of setting this preference has been removed from the `deposit` function. We believe that being an user preference, the user should be able to set in

> wathever way he wants independently from the platform configuration, especially considering that the platform configuration may change in the future. We will make sure to document more in detail this function though to ensure better understanding of the meaning of the flag for users."

[L07] Potential division by zero

The `balanceDecreaseAllowed` function in the `LendingPoolDataProvider` contract is used to validate decreases in an account's collateral balance. Once it calculates the collateral balance after the decrease, the function computes the resulting liquidation threshold. However, in this last calculation the new collateral balance is used as a divisor without previously validating whether it is zero.

While this does not pose a security issue, since the `div` function of OpenZeppelin `SafeMath` is used, it should be noted that the division by zero will cause the transaction to revert with an unexpected and not user-friendly error message from the `SafeMath` library. Therefore, to avoid unexpected behaviors, consider returning `false` if the `collateralBalancefterDecrease` variable is zero.

Update: *Fixed in MR#85*.

[L08] Erroneous data logged in LiquidationCall event

The `liquidationCall` function in the `LendingPool` contract emits a `LiquidationCall` event to notify off-chain clients about successful liquidations. However, it currently logs erroneous data.

- While the event logs the `_purchaseAmount` argument passed by the liquidator, this argument can be higher than the actual amount liquidated (as seen in lines 110 to 114 of `LendingPoolLiquidationManager.sol`). The amount logged should match the actual amount liquidated.
- The first argument passed to the event should be `_collateral` (instead of `liquidationManager`).

It must be highlighted that currently there are no unit tests covering the emission of the `LiquidationCall` event and the data it logs. Therefore, consider implementing related unit tests to prevent these errors from being reintroduced in future changes to the code base.

Update: *Fixed in MR#87*.

[L09] Erroneous data logged in Repay event

The `repay` function in the `LendingPool` contract emits a `Repay` event to notify off-chain clients about successful repayments. When a user passes a value higher than the debt (including `UINT_MAX_VALUE`) in the `_amount` parameter, the function assumes the user is willing to repay the entire borrow (whatever the actual amount to be repaid is). In this scenario, after the successful repayment is completed, the function will emit the `Repay` event logging `_amount` as the repaid amount, where it should actually log the amount effectively paid (*i.e.* `paybackAmount`).

Consider modifying the amount logged by the `Repay` event so that it matches the actual amount paid by the user. It should be noted that currently there are no unit tests covering the emission of the `Repay` event and the data it logs. Therefore, consider implementing related unit tests to prevent this error from being reintroduced in future changes to the code base.

Update: *Fixed in MR#47*.

[L10] Redundant BurnOnRedeem event

The `burnOnRedeemInternal` function of the `AToken` contract emits a `BurnOnRedeem` event that logs the account and the number of tokens burned. However, this immediately follows a call to OpenZeppelin's ERC20 contract `_burn` function which emits a `Transfer` event with the same information.

To favor simplicity and avoid redundant operations, consider removing the `BurnOnRedeem` event.

Update: *Fixed in* `bc43147`.

[L11] MintOnDeposit event logs amount of underlying asset

The `MintOnDeposit` event logs the amount of underlying asset sent instead of the number of aTokens minted. This might be considered intended behavior since the `_mint` function already emits a `Transfer` event logging the number of aTokens. However, it is inconsistent with the data logged in the `Redeem` event of the `redeem` function.

Consider modifying the `MintOnDeposit` event to log both the amount of the underlying asset sent and the number of aTokens minted.

Update: *Fixed in MR#36*.

[L12] Unfulfillable condition

The first condition in the `require` statement in line 211 of `LendingPool.sol` (inside the `borrow` function), checks whether the user-controlled argument `_interestRateMode` is equal to `uint256(CoreLibrary.InterestRateMode.VARIABLE)`. This is an unfulfillable condition, considering that the `if` statement in line 207 already ensures `_interestRateMode` is going to be equal to `uint256(CoreLibrary.InterestRateMode.FIXED)`.

To favor readability and avoid unnecessary validations, consider removing the unfulfillable condition.

Update: *Fixed in MR#89*.

[L13] Inconsistent validations

The `transferToFeeCollectionAddress` function of the `LendingPoolCore` contract does not currently prevent users from sending ETH along an ERC20 transfer operation like the `transferToReserve` function does. To favor consistency and prevent unexpected behaviors, consider implementing this validation in the `transferToFeeCollectionAddress` function.

Update: *Fixed in MR#88.*

[L14] Tests not passing

After following the instructions in the README file of the project, the testing suite finishes with six failing tests. Such tests are:

```
- Contract: LendingPool - token economy tests
- BORROW - Test user cannot borrow using the same currency as collateral

- Contract: LendingPool FlashLoan function
- FLASH LOAN - Takes ETH Loan, returns the funds correctly:

- Contract: LendingPool FlashLoan function
- FLASH LOAN - Takes ETH Loan, does not return the funds:

- Contract: LendingPool FlashLoan function
- FLASH LOAN - Takes out a 500 DAI Loan, returns the funds correctly:

- Contract: LendingPool FlashLoan function
- FLASH LOAN - Takes out a 500 DAI Loan, does not return the funds:

- Contract: LendingPool liquidation
- LIQUIDATION - Liquidates the borrow
```

As the test suite was left outside the audit's scope, please consider thoroughly reviewing the test suite to make sure all tests run successfully. Furthermore, it is advisable to only merge code that does not break the existing tests (nor decreases coverage).

Update: *The Aave team acknowledges this issue, and has now revamped the test suite:*

> "Test were breaking on the audited commit because of an issue in the migration scripts. As the test system has been completely revamped, these tests are not failing anymore and more than 60 other unit tests have been added."

[L15] Missing comprehensive docstrings

Many of the contracts and functions in the code base lack comprehensive 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. In general, docstrings should explicitly explain the purpose or intention of functions and its parameters, the scenarios under which they can fail, the roles allowed to call them, the values returned and the events emitted.

Consider thoroughly 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. When writing docstrings, consider consistently following the Ethereum Natural Specification format (NatSpec).

Update: *The Aave team acknowledges this issue:*

> "We will make sure to add improved documentation following the NatSpec format."

[L16] Erroneous docstrings and comments

Several docstrings and inline comments throughout the code base were found to be erroneous and should be fixed. In particular:

- In lines 93 and 94 of `CoreLibrary.sol` the docstrings for the `getNormalizedIncome` function appear to be a copy from the previous function `getReserveUtilizationRate`.
- Line 70 of `CoreLibrary.sol` should say "usageAsCollateralEnabled" instead of "usageAsCollateral".
- Within the `borrow` function of the `LendingPool` contract, enumeration of steps in inline comments jump from step 2 to 4.
- Comment in line 366 of `LendingPool.sol` should be removed.
- Comment in line 641 of `LendingPool.sol` should say "greater or equal" instead of "greater of equal". Additionally, note that the following `require` statement is just checking for strict equality.
- Docstrings for the `burnOnRedeemInternal` function state that "only lending pools can call this function". Yet the function is marked as `internal`, and cannot be called from other contracts.
- Docstrings for the `repay` function state that "repays a borrow [...] for the specified amount". Which not always holds true – if the caller specifies an amount equal to `UINT_MAX_VALUE` then the entire borrow is repaid.
- In the `ILendingRateOracle` interface all units specified in docstrings should be in Ray instead of Wei.
- Lines 18 and 23 of `IPriceOracle.sol` should say "ETH price" instead of "asset price".
- Line 23 of `LendingPoolLiquidationManager.sol` should say "LendingPoolLiquidationManager" instead of "LiquidationManager".
- Line 33 of `LendingPoolDataProvider.sol` should say "the loan can be liquidated" instead of "the loan gets liquidated".

- Line 144 of `LendingPoolDataProvider.sol` references a `transferInternal()` function in the `AToken` contract that does not exist.

Update: *Fixed in MR#90 and MR#98.*

[L17] Misleading error messages

- The `LendingPoolCore` contract's fallback function intends to only allow transfers of ETH from contracts. However, given the current implementation, transfers of ETH would fail with a misleading error message whenever executed from contracts' constructors. Consider clarifying such behavior in an inline comment to prevent unexepected behaviors.
- In line 463 of `LendingPool.sol`, the error message does not match the actual checked condition, and should therefore be modified to *"[...] an amount that is equal or smaller than [...]"*.
- While the error message in line 610 of `LendingPool.sol` states that *"The caller of this function must be a contract"*, the `flashLoan` function can be called by both contracts and externally-owned accounts.

Update: *Not fixed. The issue is tracked in #98.*

[L18] Whitepaper issues

The commit audited includes the Aave protocol whitepaper in the `docs` folder. Following we include a list of issues related to the whitepaper that must be addressed:

- Throughout the entire whitepaper, "ray" and "Ray" are used interchangeably.
- The definition for the Health Factor does not match the code, and should be updated.
- In section 1.2, "Block height at which [...]" should say "Timestamp at which [...]". Additionally, `Bl` should be replaced with `Tl`.
- There is an unfinished sentence in section 1.9, where it says "Check paragraph X where [...]".
- There are two sections numbered 1.15. More importantly, they correspond to a circular definition between the "Current liquidity rate" and the "Borrow/Liquidity Rate delta".
- In section 1.16, "`Ci` Is" should say "`Ci` is" (a blank space is missing). Moreover, the last sentence is not coherent and should say "The formula to calculate `Ci` at a specific point in time [...]".
- In section 1.17, the definition for the "Reserve normalized income" is incorrect, as it was copied from section 1.16.
- In the last sentence of section 1.21, `BΔu` should be replaced with `TΔu`.
- The last paragraph of section 2.3 refers to "chapter 4" that does not exist in the whitepaper.

- In section 3.1, the formula of “Average Market Lending Rate” defines a weighted sum. To obtain the average, it should be divided by the total volume of the platforms.
- The symbol `Bf` is mistakenly used instead of `Rf` to refer to the “Current fixed borrow rate”.
- In the “Flash Loan” section, the actual method executed on the external contract is `executeOperation` and not `executeAction` as stated.
- In the “Current fixed borrow rate” section, the formula does not specify what occurs when the utilization rate `U` is equal to the threshold `Tr`. Additionally, the whitepaper states that the utilization rate threshold is set to `0.25 * 1e18`, but in the code it is set to `1e27 / 2`.
- In the “Fixed rates Rebalancing” section, the inequality for the rebalancing up is in the wrong direction (it should be `Bfu > Lr`).
- In the “Fixed rates Rebalancing” section, the formula for rebalancing down should say (1 + delta) instead of (1 – delta).

Update: *Partially fixed. The whitepaper has been significantly improved. The new version of the whitepaper still has minor issues to be addressed. In particular:*

- The new version of the whitepaper states that the utilization rate threshold is now set to `1e27/2`, but in the code it is set to `1e27 / 4`.
- In the “The Rebalancing Process” section, the function `rebalanceFixedBorrowRate` should be renamed to `rebalanceStableBorrowRate`.
- Section 1.2 mentions a `LendingPoolLibrary.sol` file that does not exist.
- References to `decreaseTotalBorrowsFixedAndUpdateAverageRate` and `increaseTotalBorrowsFixedAndUpdateAverageRate` function should be updated to account for the “fixed” to “stable” renaming.
- In the introduction to section 4, the term “fixed rate” is used
- In section 4.4, “fixed borrow rate of user x” should be changed to “stable borrow rate of user x”.
- Figure 14 is missing a “Yes” label in the second conditional block.
- Consider using the subscript `s` (instead of `f`) to refer to the stable rate parameters. Since `s` is already used to reference scaling variables, consider using a synonym like multiplier with the subscript m`.
- Typo in section 2.6: “all votes are biding”.

Notes and Additional information

[N01] Markets can become insolvent

When the value of all collateral is worth less than the value of all borrowed assets, we say the market is insolvent. Approaches to reduce the risk of market insolvency include: prudent selection of collateral-ratios, incentivizing third-party collateral liquidation, and careful selection of which tokens are listed on the platform, among others. However, the risk of insolvency cannot be entirely eliminated, and there are numerous ways a market can become insolvent. As examples:

- The price of the underlying (or borrowed) asset makes a big, quick move during a time of high network congestion — resulting in the market becoming insolvent before enough liquidation transactions can be mined.
- The price oracle temporarily goes offline during a time of high market volatility. This could result in the oracle not updating the asset prices until after the market has become insolvent. In this case, there will never have been an opportunity for liquidation to occur.
- Administrators list an ERC20 token with a later-discovered bug that allows minting of arbitrarily many tokens. This bad token is used as collateral to borrow funds that it never intends to repay.

In any case, the effects of an insolvent market could be disastrous. It would mean that aToken contracts would effectively be running a fractional reserve. This could result in a "run on the bank", with the last suppliers losing their money.

This risk is not unique to the Aave protocol, and was also highlighted in our Compound audit. All collateralized loans (even non-blockchain loans) have a risk of insolvency. However, it is important to raise awareness on this risk, and that it can be difficult to recover from even a small dip into insolvency.

Update: *The Aave team acknowledges the note, accepting it as a inherent risk of the basic economic model.*

[N02] Fixed-rate loans may never compound

The Aave protocol allows borrowers to take out loans either at a fixed or variable interest rate.

Variable-rate loans compound interest every time any user updates the reserve from which the loan was taken. Conversely, fixed-rate loans do not compound when there are updates on the reserve. They only compound interest when the particular loan is updated. In other words, unless the borrower who took out the fixed-rate loan borrows again, repays part of the loan, swaps the rate mode, rebalances the fixed rate, or is liquidated, their loan interest will never be compounded.

Even though this issue does not pose a security risk, it is important to explicitly state this fundamental difference between the two types of loans, so as to raise end-user awareness.

Update: *The Aave team acknowledges the note and will clarify better in the documentation this particular difference between variable and stable rate loans.*

[N03] Fixed interest rate loans feature is loosely encapsulated

Throughout the code base, the boundary and interplay between fixed-rate loans and variable-rate loans is not well defined, which can lead to surprising or undesirable behavior. To investigate this idea it is useful to explore a few different examples, some that are already mentioned in this report, while viewing them as consequences of an underlying architecture.

Since each user can have only one active loan from each reserve, the `borrow` function must close the existing loan and create a new one. This may implicitly convert between fixed and variable rate loans. However, this allows users to bypass the size and collateral restrictions placed on new fixed-rate loans, or loans that are explicitly swapped with the `swapBorrowRateMode` function.

In fact, the reason for the size restrictions in the first place is that it is possible to manipulate the utilization rate (through deposits and borrows). This in turn effects the rate applied to a new fixed-rate loan, which will not respond to the subsequent changes to utilization rate when the manipulation is reversed. The Aave team acknowledges that the size and collateral requirements adds a barrier, but does not remove the possibility of such a manipulation occurring at a small scale using a single account, or at a larger scale using multiple accounts. Ultimately, this attack is possible due to the fact that fixed-rate and variable-rate loans share the same asset pool and utilization rate, but respond differently to market changes.

This also leads to another security requirement, where the fixed rate must always be higher than the liquidity variable rate to avoid users borrowing from and loaning into the same pool. The solution of rebalancing the loans when this occurs has some undesirable consequences. Firstly, it means that the rate of fixed-rate loans can actually vary. Secondly, it introduces a discontinuity, where someone with a loan at the bottom of the window can suddenly find their rate increase to the current market rate. Lastly, this condition is manipulable through the utilization rate, which means that users can intentionally make themselves or other users vulnerable to rebalancing (in either direction).

Lastly, variable-rate loans compound whenever any user updates the reserve but fixed-rate loans only compound when the particular loan is updated. This may be surprising to users, and even to developers with some familiarity with the code given the similarity between the supporting data structures and update functions.

Although these issues can be addressed individually, we believe them to be symptoms of the loose encapsulation of fixed-rate loans, where the implementation shares data structures and the reserve pool with variable-rate loans and lenders. Consider defining clear boundaries around the fixed-rate loan feature (with documentation and code separation) to simplify the system and make it easier to reason about.

Update: *The Aave team understands that the current behavior of the feature is intended, and is satisfied with its design. While we still keep our reservations around the stable rate feature, the renaming from "fixed" to "stable" is a step forward in better clarifying the real behavior of the protocol. The upcoming documentation for N02 should also help to better delineate the nature of both types of rates.*

[N04] Upgrades should update cached addresses

The Aave protocol is intended to be upgraded in the future by means of a to-be-implemented governance system. These upgrades should occur when new addresses are set by privileged accounts in the `LendingPoolAddressesProvider` contract (see **"[H01] Lack of access controls"**) related issue).

Several components of the system are not isolated but depend on each other. For efficiency, they cache addresses in storage (to avoid querying the `LendingPoolAddressesProvider` contract too often). As a consequence, setting a new address in the `LendingPoolAddressesProvider` must

always be considered as just one of potentially many steps of the upgrade process. All addresses cached in contracts that depend on the upgraded one must be updated as well.

Following we include a list detailing which addresses are cached in contracts of the system.

- `LendingPool` saves in storage the address of: `LendingPoolCore`, `LendingPoolParametersProvider`, `LendingPoolAddressesProvider` and `LendingPoolDataProvider`.
- `LendingPoolLiquidationManager` saves in storage the address of: `LendingPoolCore`, `LendingPoolParametersProvider`, `LendingPoolAddressesProvider` and `LendingPoolDataProvider`.
- `DefaultReserveInterestRateStrategy` saves in storage the address of: `LendingRateOracle` and `LendingPoolCore`.
- `LendingPoolDataProvider` saves in storage the address of: `LendingPoolCore` and `LendingPoolAddressesProvider`.
- `LendingPoolConfigurator` saves in storage the address of: `LendingPoolAddressesProvider`.
- `LendingPoolCore` saves in storage the address of: `LendingPool` and `LendingPoolAddressesProvider`.

Update: *The Aave team acknowledges the issue:*

> "The future Aave governance framework will be in charge of controlling and updating the cached addresses once any update on any affecting contract is applied."

[N05] Missing relevant data in events

Some emitted events may benefit from logging additional relevant data. In particular:

- The `Borrow` event should log the borrow fee paid.
- The `Repay` event should log the address of the actual repayer, considering repayments can be made on behalf of other accounts.
- The `LiquidationCall` event should log the liquidator's address.

Update: *Fixed in MR#47.*

[N06] Unused LiquidationCompleted event

Line 51 of `LendingPoolLiquidationManager.sol` declares a `LiquidationCompleted` event. As it is never emitted, consider removing the declaration or emitting the event appropriately.

Update: *Fixed in MR#57.*

[N07] Flash loan reverts when amount equals available liquidity

The `flashLoan` function of the `LendingPool` contract requires the amount of the loan to be strictly lower than the reserve's available liquidity. However, this restriction will revert the transaction when both amounts are equal, with a misleading error message informing that there is not enough liquidity.

Update: *Fixed in MR#55*.

[N08] Function redeem can be called with amount zero

The `redeem` function of the `AToken` contract can be called with argument `_amount` as zero. To prevent unncessary event emissions, and following the "fail early and loudly" pattern, consider adding a `require` statement to validate that `_amount` is greater than zero.

Update: *Fixed in MR#60*.

[N09] Refactor getReserveUtilizationRate function

To calculate the total borrows of a reserve, the `getReserveUtilizationRate` function of `CoreLibrary` should reuse the available getter `getTotalBorrows` instead of repeating the code `_self.totalBorrowsFixed.add(_self.totalBorrowsVariable)`.

Update: *Fixed in MR#62*.

[N10] Reuse modifier onlyReserveWithEnabledBorrowingOrCollateral

The `require` statement and assignment of `reserve` in lines 677 to 682 of `LendingPoolCore.sol` accomplish the same functionality as the `onlyReserveWithEnabledBorrowingOrCollateral` modifier.

Consider removing these lines and adding the modifier to the `transferToReserve` function declaration.

Update: *Fixed MR#63*.

[N11] Redundant getters

- Getters `getUserReserveData` in `LendingPool` contract and `getUserReserveData` in `LendingPoolDataProvider` are two external functions that currently return the same data. Note that the former getter internally calls the later.
- There are three public getters to read the `reservesList` state variable of the `LendingPoolCore` contract. Namely: `getReserves` in `LendingPoolCore`, `reservesList` (automatically generated by Solidity) and `getReserves` in `LendingPool`.

To favor simplicity and encapsulation, consider removing all these redundant functions, ensuring that there is always at most one publicly accessible getter for exposed data.

Update: *Not an issue. Redundancy is intended.*

[N12] Implicit upcasting of uint40 variable

To favor explicitness and code readability, consider explicitly casting the `_lastUpdateTimestamp` variable from `uint40` to `uint256` in line 314 of `CoreLibrary.sol`.

Update: *Fixed in MR#65.*

[N13] Unclear use of a struct for function local variables

The `borrow` function of the `LendingPool` contract uses a struct called `BorrowLocalVars` instead of declaring local variables. A similar situation occurs in the `calculateUserGlobalData` function of `LendingPoolDataProvider`, which uses the `UserGlobalDataLocalVars` struct.

As the purpose of such implementations is unclear, consider including an inline comment to clearly explain why these functions favor the use of the structs over local variables. This should add to the code's readability, preventing developers from introducing undesired changes to the code base in the future.

Update: *Fixed in MR#66.*

[N14] Unused state variable

In the `LendingPoolLiquidationManager` contract, the state variable `parametersProvider` is never used and should therefore be removed.

Update: *Not an issue. Even though the state variable `parametersProvider` is not used, it does help keep the storage's layout aligned to the storage of the `LendingPool` contract (which `delegatecall`s to `LendingPoolLiquidationManager`).*

[N15] Lack of explicit visibility in state variables

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

- In the `WadRayMath` library:
  `WAD_RAY_RATIO`

- In the `CoreLibrary` library:
  `SECONDS_PER_YEAR`
- In the `DefaultReserveInterestRateStrategy` contract:
  `FIXED_RATE_INCREASE_THRESHOLD`,
  `core`,
  `lendingRateOracle`,
  `baseVariableBorrowRate`,
  `variableBorrowRateScaling`,
  `fixedBorrowRateScaling`,
  `borrowToLiquidityRateDelta`,
  `reserve`
- In the `LendingPool` contract:
  `addressesProvider`,
  `core`,
  `dataProvider`,
  `parametersProvider`,
  `UINT_MAX_VALUE`
- In the `LendingPoolConfigurator` contract:
  `poolAddressesProvider`
- In the `LendingPoolCore` contract:
  `ethereumAddress`,
  `lendingPoolAddress`,
  `addressesProvider`,
  `reserves`,
  `usersReserveData`
- In the `LendingPoolDataProvider` contract:
  `core`,
  `addressesProvider`,
  `HEALTH_FACTOR_LIQUIDATION_THRESHOLD`
- In `FlashLoanReceiverBase`:
  `addressesProvider`
- In `FeeProvider`:
  `originationFeePercentage`,
  `feesCollectionAddress`

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

Update: *Fixed in MR#68.*

[N16] Named return variables

There is an inconsistent use of named return variables across the entire code base. Consider removing all named return variables, explicitly declaring them as local variables, and adding the necessary return statements where appropriate. This should favor both explicitness and readability of the project.

Update: *Aave acknowledges the inconsistency and plans to consistently unify the return statement declarations in the future, in one way or another.*

[N17] Naming issues

Several variables, parameters and functions throughout the code base might benefit from better naming to favor readability and avoid confusions. Specifically:

- Modifier `whenTranferAllowed` should be called `whenTransferAllowed`.
- State variable `initialExchangeRate` of `AToken` contract is not self-explanatory enough. It should denote the "direction" of the exchange rate. That is, either [aToken / asset] or [asset / aToken].
- The struct field `collateralBalancefterDecrease` should be called `collateralBalanceAfterDecrease`.
- All forms of `liquidationDiscount` should be changed to `liquidationBonus`, as they refer to a value greater than 1.
- The `currentLtv` variable used in the `calculateUserGlobalData` function of the `LendingPoolDataProvider` contract is not the actual "current loan-to-value" because it is not averaged over the total collateral balance in ETH yet. This step is done by reassigning this variable in line 119. The same can be said for the `currentLiquidationThreshold` variable.
- The same name `usageAsCollateralEnabled` is used to refer to two different states of a reseve. On one hand, it can mean that the user has enabled a certain reserve to be used as collateral (see line 369 of `LendingPoolDataProvider.sol`). On the other, it denotes whether a reserve has been set up to be used as collateral system-wide (see line 273 of `LendingPoolDataProvider.sol`) by the administrators.

Update: *Acknowledged by Aave:*

> "As this issue poses further groundwork since it impacts other off-chain services, we acknowledge this issue and provide better naming for those variables in a future revision."

[N18] Typos

- Line 199 of `LendingPool.sol` should say “need” instead of “needs”.
- Line 382 of `LendingPool.sol` should say “cumulate” instead of “comulate”.
- Line 389 of `LendingPool.sol` should say “payback” instead of “payaback”.
- Lines 389 and 392 of `LendingPool.sol` should say “subtracting” instead of “substracting”.
- Line 646 of `LendingPool.sol` should say “is inconsistent” instead of “in inconsistent”.
- Line 71 of `AToken.sol` should say “underlying” instead of “undelying”.
- Lines 128 and 154 of `AToken.sol` should say “Optimization” instead of “Optmization”.
- Line 281 of `CoreLibrary.sol` should say “subtract” instead of “substract”.
- Line 293 of `CoreLibrary.sol` should say “subtracted” instead of “substracted”.
- Line 97 of `LendingPoolLiquidationManager` should say “liquidated” instead of “liuquidated”.
- In the `README` file, “accross” should say “across”.
- In the `README` file, all instances of “docker compose” should say “docker-compose”.

Update: *Fixed in MR#69*.

[N19] Unnecessary use of Ownable contract

The `LendingPoolCore`, `LendingPoolDataProvider` and `LendingPoolConfigurator` contracts all unnecessarily inherit from OpenZeppelin’s Ownable contract. Given features from `Ownable` are never used in these contracts, consider removing it from their inheritance chains.

Update: *Fixed in MR#70*.

[N20] Unnecessary imports

There are several import statements that are not used and can be removed. Namely:

- Imports in lines 7, 11, 15 and 16 of `LendingPoolLiquidationManager.sol`.
- Import in line 3 of `AToken.sol`.
- Import in lines 6 and 10 of `LendingPool.sol`.
- Import in line 6 and 9 of `LendingPoolConfigurator.sol`.
- Import in line 10 of `LendingPoolCore.sol`.

Update: *Fixed in MR#71*

[N21] TODOs in code

There are "TODO" comments in the code base that should be removed and instead tracked in the project's issues backlog. See for example line 627 of `LendingPool.sol`.

Update: *Fixed in MR#55, MR#52 and MR#97.*

[N22] Coding style

Minor deviations from the Solidity Style guide were seen throughout the code base. To favor code readability, consider consistently following Solidity's coding style guide, which can be enforced across the entire project by means of a linter tool, such as Solhint.

Update: *Aave acknowledges the code style deviations and will adapt them in the next iterations of the code base.*

[N23] Declare uint as uint256

To favor explicitness, all instances of `uint` should be declared as `uint256`.

Update: *Fixed in MR#72.*

Conclusion

Originally, 6 critical and 8 high-severity issues were found. Several changes were proposed to follow best practices and reduce potential attack surface. After reviewing all issues reported with the Aave team, we downgraded 1 critical issue and considered 2 high-severity issues as non-issues. The severe attack vectors and broken features identified, together with the meager test suite and sparse documentation, are the reflection of a work-in-progress code base. As stated in the **Audit Update** section, and in each particular issue as well, **all severe issues identified have been correctly fixed**.

This first audit round has been Aave's initial step on its way to reach the needed level of maturity for projects intended to handle large sums of financial assets. So as to further help the project reach a production-ready state, we highly advise additional rounds of security reviews.

- If you are interested in smart contract security, you can continue the discussion in our forum, or even better, join the team 🚀
- If you are building a project of your own and would like to request a security audit, please do so here.

RELATED POSTS

SECURITY AUDITS

InstaDApp Audit Summary

The InstaDApp team asked us to audit their proxy wallet and wallet registry contracts. Here is a...

READ MORE

by OpenZeppelin Security

SECURITY AUDITS

Compound Audit

The Compound team asked us to review and audit their platform's smart contracts. We examined their...

READ MORE

by OpenZeppelin Security

SECURITY AUDITS

Solidity Compiler Audit

The Augur team and the Ethereum Foundation (through a joint grant) asked us to review and audit the...

READ MORE

by OpenZeppelin Security