Aave Protocol

Security Assessment

September 25, 2019

Prepared For:
Emilio Frangella | *Aave*
emilio@aave.com

Stani Kulechov | *Aave*
stani@aave.com

Prepared By:
Gustavo Grieco | *Trail of Bits*
gustavo.grieco@trailofbits.com

Dominik Czarnota | *Trail of Bits*
dominik.czarnota@trailofbits.com

Michael Colburn | *Trail of Bits*
michael.colburn@trailofbits.com

Changelog:

September 6, 2019:	Initial report delivered
September 25, 2019:	Added Appendix E with retest results
October 23, 2019:	Updated product name from DLPs to Aave Protocol

Executive Summary

From August 26 through September 6, 2019, Aave engaged Trail of Bits to review the security of its Aave Protocol smart contracts. Trail of Bits conducted this assessment over the course of four person-weeks with three engineers working from commit hash `493d75db59de33ad1f2bb13612ff8acc8654dd3d` from the `aave-tech/dlp` repository. Aave developed a decentralized lending pool (DLP) protocol in Solidity that allows users to deposit, borrow, and repay loans, using different types of compounding interest rates. The system keeps tracks of the liquidity and health metrics to make sure users can only perform certain operations (e.g., borrow or redeem) with acceptable levels of risk.

During the first week, the assessment focused on Aave Protocol's smart contracts. This included a broad pass of the codebase to familiarize ourselves with its architecture, as well as a specific focus on core functions. We dedicated the second week to investigating high-level interactions of benign and malicious users within the Lending Pool to identify potential attacks like front-running and race conditions. We also evaluated the potential use of Echidna, our properly-based testing tool (Appendix C) and provided feedback on the whitepaper and its corresponding code (Appendix D).

During the engagement, Trail of Bits identified 15 issues, ranging in severity from informational to high. We discovered:

- Issues related to missing access control code that is still in development or needs to be improved.
- Issues in the code related to insufficient data validation of untrusted inputs. This opened the door to several attacks to drain the available funds with a call to redeem or subvert normal behavior by changing the interest rates of the Lending Pool.
- Issues with race conditions and potential front-running attacks that may trick users into paying more than expected, or produce a denial-of-service of certain operations.
- Issues related to unsafe compiler optimizations, potentially invalid contract initialization, and leftover code used only for debugging.

Overall, the code reviewed is indicative of a work in progress. A significant portion of functionality, including distributed governance and the associated access control mechanisms, is still in development and will likely change in the future. The Aave Protocol is a large code base, comprised of an extensive amount of corner cases throughout the protocol.

Aave Protocol requires further development before it is mature enough for production deployment. Documentation of the expected system operation and environment is necessary. The Aave team is aware of gaps in its testing suite and is working to improve test coverage. Once this is completed, Trail of Bits recommends further assessment to re-evaluate the security posture of the system.

Update: From September 23 to September 25, 2019, Trail of Bits reviewed fixes proposed by Aave for the issues presented in this report. See a detailed review of the current status of each issue in Appendix E.

Project Dashboard

Application Summary

Name	Aave Protocol
Version	`493d75db59de33ad1f2bb13612ff8acc8654dd3d`
Type	Solidity Smart Contracts
Platforms	Ethereum

Engagement Summary

Dates	August 26 - September 6, 2019
Method	Whitebox
Consultants Engaged	3
Level of Effort	4 person-weeks

Vulnerability Summary

Total High-Severity Issues	7	■■■■■■
Total Medium-Severity Issues	2	■■■
Total Low-Severity Issues	3	■■■
Total Informational-Severity Issues	2	■■
Total Undetermined-Severity Issues	1	■
Total	15	

Category Breakdown

Undefined Behavior	1	■
Access Controls	5	■■■■■
Timing	3	■■■
Data Validation	6	■■■■■■
Total	15	

Engagement Goals

Trail of Bits and Aave scoped the engagement to provide a security assessment of the Aave Protocol smart contracts in the `aave-tech/dlp` repository.

Specifically, we sought to answer the following questions:

- Is it possible for the protocol to lose money?
- Can interest rates be manipulated?
- Can reserve data be manipulated?
- Are access controls well defined?
- Is it possible to manipulate the market by using specially crafted parameters or front-running transactions?
- Is it possible for participants to steal or lose tokens?
- Can participants perform denial-of-service attacks against any of the contracts?

Coverage

This review included the contracts comprising the Aave Protocol. The specific version of the codebase used for the assessment came from commit hash `493d75db59de33ad1f2bb13612ff8acc8654dd3d` of the `aave-tech/dlp` Gitlab repository.

Trail of Bits reviewed contracts for common Solidity flaws, such as integer overflows, reentrancy vulnerabilities, and unprotected functions.

Arithmetic. Complex calculations, often including fixed-point math, take place in many parts of the system. We reviewed these calculations for logical consistency and any issues that may result from rounding or overflow.

Access controls. Some parts of the system expose privileged functionality, such as modifying borrow rates or fees. We reviewed these functions to ensure they can only be triggered by the intended actors. However, as distributed governance was not implemented at the time of the assessment, it was not possible to review the adequacy of some of these mechanisms (e.g., TOB-DLP-003).

LendingPoolCore. We gave special attention to the core lending pool contract, as that is one of the few non-upgradable components of the Aave Protocol and contains important logic.

Market manipulation. Participants may manipulate market parameters, time their interactions, or collude with other participants to profit from crypto-economic incentives underlying the markets. In reviewing the design and implementation, we assumed any participant may be untrustworthy and malicious.

The distributed governance and the `collateralCall` functionality were still a work in progress and thus, out of scope of this assessment.

This engagement did not have sufficient time for review of the testing suite or deployment scripts. Aave also noted that several issues from a recent security assessment of Compound Finance may be relevant to their protocol. As the team was already aware of these issues, coverage was focused elsewhere and they were not included in this report.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

❑ **Measure the gas savings from optimizations, and carefully weigh that against the possibility of an optimization-related bug.** Disabling optimization will mitigate compilation for present and future issues introduced by the compiler.

❑ **Use the `onlyLendingPool` modifier with the `updateReserveTotalBorrowsByRateMode` function to restrict the access to that code**. It will avoid unauthorized users changing critical state variables in the Lending Pool.

❑ **Protect setter operations in `LendingPoolAddressesProvider`, `NetworkMetadataProvider`, and their base contracts (`AddressStorage` and `UintStorage`) from being called by unexpected addresses**. It will avoid unauthorized users changing critical state variables in the Lending Pool

❑ **Document potential race conditions and front-running attacks to make sure users are aware of this risk when they use redeem, borrow, or repay**. It will be easier to report and identify these types of attacks if users are aware of them.

❑ **Properly validate the amount of ETH sent to the Lending Pool during a repay.** It will avoid trapping user funds that were sent by mistake.

❑ **Properly validate the provided `aToken` address using the corresponding reserve**. It will prevent using untrusted addresses to read the amount of tokens available to redeem.

❑ **Properly validate the `interestRateMode` inputs to allow only valid modes for each reserve.** It will prevent the use of restricted interest modes from certain reserves.

❑ **Properly define input ranges, and validate the `DefaultReserveInterestRateStrategy` contract parameters on-chain**. It will prevent invalid setup parameters that can cause unexpected behavior in the Lending Pool.

❑ **Properly implement a mechanism to redeem all the available tokens, given the current liquidity and health liquidation factor. Alternatively, document the constraints to redeem tokens under situations of limited liquidity or low health liquidation factor**. This will help users to redeem as many tokens as possible during a bank run.

❑ **Add missing validation checks to the relevant functions in `LendingPoolCore`.** This will prevent using uninitialized values when the reserve values have not been previously initialized.

❑ **Remove the test mode, or refactor it to a separate contract that uses inheritance to expose this special function.** This will prevent users from distrusting the Lending Pool if they notice this special testing mode in the deployed contract.

Long Term

❑ **Monitor the development and adoption of Solidity compiler optimizations to assess its maturity**. This will avoid introducing any known issues caused by the compiler during future development.

❑ **Add proper unit tests for corner cases in which a user tries to perform an invalid or unauthorized operation**, including:

- Configuration contracts cannot be deployed with invalid parameters.
- A user cannot successfully call any state-modifying function in the `LendingPoolCore` contract.
- A user cannot successfully call any Lending Pool function using an arbitrary address as reserve or token contracts.
- The `LendingPool` contract cannot successfully take ETH and ERC20 tokens from a user at the same time.

❑ **Implement a governance system to agree on changes**. This will address the access control issues detected during this assessment.

❑ **Consider requesting the user input an upper bound of the price he is willing to repay. Revert if the load is higher than expected, or remove the feature to repay the entire loan using `uint(-1)`**. This will mitigate a race condition in the use of borrow/repay.

❑ **Consider monitoring the blockchain using the Repay events**. This will help catch front-running attacks.

❑ **Take into account that all information sent to Ethereum is public prior being accepted by the network**. Design the contracts to be robust to the unpredictable nature of transaction ordering.

❑ **Always ensure data has been initialized before accessing it**. This will avoid unexpected behavior when using uninitialized values.

❑ **Review all capabilities in your smart contracts to make sure you are not including any debugging code that could be used as a backdoor**. This will prevent users from distrusting your contracts if they notice debugging code in the deployed contract.

❑ **Consider using Echidna and Manticore to test important properties of the Lending Pool**, including:

- Deploying of configuration contracts will only valid parameters.
- User can redeem the maximum amount of their token using `redeem`.
- Users cannot change the interest mode of their loans if the reserve has restrictions.

Findings Summary

#	Title	Type	Severity
1	Solidity compiler optimizations can be dangerous	Undefined Behavior	Undetermined
2	Lack of access controls in updateReserveTotalBorrowsByRateMode	Access Controls	High
3	Lack of access control for LendingPoolAddressesProvider and NetworkMetadataProvider	Access Controls	Medium
4	Lack of access control for FeeProvider setter functions	Access Controls	High
5	borrow/repay calls race condition can be exploited to repay more than expected	Timing	High
6	repay calls can be blocked by front-running them with another repay call	Timing	Low
7	repay does not validate if the user borrows ERC20 tokens, but pays using ether	Data Validation	Low
8	redeem does not properly validate parameters and allows drainage of funds	Data Validation	High
9	updateReserveTotalBorrowsByRateMode does not properly validate InterestRateMode inputs	Data Validation	High
10	DefaultReserveInterestRateStrategy parameters are not validated and can block the Lending Pool	Data Validation	Low
11	There is no way to redeem all available tokens when Lending Pool parameters are close to the redemption limit	Data Validation	Medium

12	Race condition on LendingPool allows attacker to redeem their token first	Timing	Medium
13	Insufficient validation of reserve address in some LendingPoolCore functions	Data Validation	Informational
14	Users should check that test mode is disabled before interacting with the Lending Pool	Access Controls	Informational
15	swapBorrowRateMode does not check if reserve allows fixed interest rates	Access Controls	High

1. Solidity compiler optimizations can be dangerous

Severity: Undetermined
Type: Undefined Behavior
Target: `truffle-config.js`

Difficulty: Low
Finding ID: TOB-DLP-01

Description
The Aave Protocol contracts have enabled optional compiler optimizations in Solidity.

There have been several bugs with security implications related to optimizations. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default. It is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the emscripten-generated `solc-js` compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe. Moreover, the Common Subexpression Elimination (CSE) optimization procedure is “implemented in a very fragile manner, with manual access to indexes, multiple structures with almost identical behavior, and up to four levels of conditional nesting in the same function.” Similar code in other large projects have resulted in bugs.

There are likely latent bugs related to optimization and/or new bugs that will be introduced due to future optimizations.

Exploit Scenario
A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to `solc-js`—causes a security vulnerability in the Aave Protocol contracts.

Recommendation
In the short term, measure the gas savings from optimizations, and carefully weigh that against the possibility of an optimization-related bug.

In the long term, monitor the development and adoption of Solidity compiler optimizations to assess its maturity.

2. Lack of access controls in updateReserveTotalBorrowsByRateMode

Severity: High
Difficulty: Low
Type: Access Controls
Finding ID: TOB-DLP-02
Target: `LendingPoolCore.sol`

Description

`updateReserveTotalBorrowsByRateMode` modifies critical state variables, but it is not protected and can be called by any user.

The `updateReserveTotalBorrowsByRateMode` function (Figure 2.1) is used to update the borrow information for an arbitrary reserve and user.

```
   function updateReserveTotalBorrowsByRateMode(
        address _reserve,
        address _user,
        uint256 _principalBalance,
        uint256 _balanceIncrease,
        uint256 _newBorrowRateMode,
        uint256 _fixedRate
    ) external {
        CoreLibrary.InterestRateMode currentRateMode =
getUserCurrentBorrowRateMode(_reserve, _user);
        CoreLibrary.ReserveData storage reserve = reserves[_reserve];

        if (currentRateMode == CoreLibrary.InterestRateMode.FIXED) {
            // there was no previous borrow or previous borrow was fixed
            CoreLibrary.UserReserveData storage user = usersReserveData[_user][_reserve];
            reserve.decreaseTotalBorrowsFixedAndUpdateAverageRate(_principalBalance,
user.fixedBorrowRate);

            if (_newBorrowRateMode == uint256(CoreLibrary.InterestRateMode.FIXED)) {

reserve.increaseTotalBorrowsFixedAndUpdateAverageRate(_principalBalance.add(_balanceIncreas
e), _fixedRate);
            }  else if (_newBorrowRateMode ==
uint256(CoreLibrary.InterestRateMode.VARIABLE)) {
                //switching the whole amount borrowed to variable

reserve.increaseTotalBorrowsVariable(_principalBalance.add(_balanceIncrease));
            }
        } else {
            // previous borrow was variable
            if (_newBorrowRateMode == uint256(CoreLibrary.InterestRateMode.FIXED)) {
                reserve.decreaseTotalBorrowsVariable(_principalBalance);

reserve.increaseTotalBorrowsFixedAndUpdateAverageRate(_principalBalance.add(_balanceIncreas
e), _fixedRate);
            } else if (_newBorrowRateMode ==
uint256(CoreLibrary.InterestRateMode.VARIABLE)) {
                //switching the whole amount borrowed to variable
                reserve.increaseTotalBorrowsVariable(_balanceIncrease);
            }
        }
    }
```

Figure 2.1: `updateReserveTotalBorrowsByRateMode` function in LendingPoolCore.sol

This function should only be called by the `LendingPool` contract; however, it currently can be called by any user.

Exploit Scenario

An attacker can directly manipulate the `LendingPoolCore` state variables, using `updateReserveTotalBorrowsByRateMode` to exploit the `LendingPool` contract.

Recommendation

Short term, use the `onlyLendingPool` modifier with the `updateReserveTotalBorrowsByRateMode` function to restrict the access to that code.

Long term, add proper unit tests to ensure that an unauthorized user cannot successfully call any state-modifying function in the `LendingPoolCore` contract.

3. Lack of access control for LendingPoolAddressesProvider and NetworkMetadataProvider

Severity: Medium Difficulty: Low
Type: Access Controls Finding ID: TOB-DLP-03
Target: LendingPoolAddressesProvider.sol, NetworkMetadataProvider.sol

Description
Both LendingPoolAddressesProvider and NetworkMetadataProvider contracts lack access controls for operations that set address or uint256. This allows an attacker to change the behavior of a given AToken, LendingPool, LendingPoolCore or LendingPoolConfigurator by changing relevant addresses or uint values.

This issue seems to be known due to "// TODO: add access control rules under DAO" comments accompanying the code of the affected functions. However, this access control scheme was not yet implemented at the time of the assessment.

Exploit Scenario
An attacker changes LendingPoolCore address of LendingPool contract's address provider (LendingPoolAddressesProvider) to another contract they control. This contract can, for example, revert all calls made to it, causing LendingPool contract's operations to fail.

Recommendation
Short term, protect setter operations from being called by unexpected addresses. The setter functions that need to be protected are in LendingPoolAddressesProvider, NetworkMetadataProvider, and their base contracts (AddressStorage and UintStorage). This can be done using an Ownable contract and setting the owner to a multi-signature wallet, which will avoid a single point of failure, in case of a private-key compromise.

Long term, implement a governance system to agree on changes.

4. Lack of access control for FeeProvider setter functions

Severity: High	Difficulty: Low
Type: Access Controls	Finding ID: TOB-DLP-04
Target: `FeeProvider.sol`	

Description

The `setFeesCollectionAddress` and `setLoanOriginationFeePercentage` functions in `FeeProvider` contract can be called by anyone. This allows an attacker to modify the loan origination fee percentage and the address to which the fees are sent.

Exploit Scenario

An attacker changes the `feesCollectionAddress` of a `FeeProvider` used by a `LendingPool` by calling the `setFeesCollectionAddress` function. As a result, all collected fees are now sent to their address, instead of the original one.

Recommendation

Short term, protect setter operations from being called by unexpected addresses. The setter functions that needs to be protected are `setFeesCollectionAddress` and `setLoanOriginationFeePercentage` in `FeeProvider` contract.

This can be done using an `Ownable` contract and setting the owner to a multi-signature wallet, which will avoid a single point of failure, in case of a private-key compromise.

Long term, implement a governance system to agree on changes.

5. borrow/repay calls race condition can be exploited to repay more than expected

Severity: High
Type: Timing
Target: `LendingPool.sol`

Difficulty: High
Finding ID: TOB-DLP-05

Description

A borrower can front-run a repay with another borrow, forcing the user to repay more than expected.

The repay function (Figure 5.1) is used to return the funds borrowed, even on behalf of other users.

```
  function repay(address _reserve, uint256 _amount, address _onBehalfOf)
external payable nonReentrant {

        uint256 compoundedBorrowBalance =
core.getUserCompoundedBorrowBalance(_reserve, _onBehalfOf);
        uint256 principalBalance =
core.getUserPrincipalBorrowBalance(_reserve, _onBehalfOf);
        uint256 originationFee = core.getUserOriginationFee(_reserve,
_onBehalfOf);

        require(compoundedBorrowBalance > 0, "The user does not have any
borrow pending");

        require(
            compoundedBorrowBalance.add(originationFee) >= _amount ||
_amount == UINT_MAX_VALUE,
            "User is trying to repay more than the total amount of the
loan"
        );
        …
```

Figure 5.1: Header of the repay function in LendingPool.sol

This function requires specifying an amount to repay. If this amount is `uint(-1)`, it will repay the current loan. Otherwise, it will check whether the amount to pay is equal or less than the loan. If it is equal to the loan, it is possible for an attacker to make another user to repay more than expected by front-running the `repay` with a `borrow`.

Exploit Scenario

Alice wants to repay Bob's loan for 1000 ERC20 tokens. She calls `repay` using `uint(-1)` as the amount. Bob sees the unconfirmed transaction and front-runs it to borrow another

1000 ERC20 tokens. As a result, `LendingPool` will take 2000 ERC20 tokens from Alice to repay, instead of the 1000 ERC20 tokens as Alice intended.

Recommendation
Short term, one possible mitigation is to instruct the users to document this behavior to make sure users are aware of this risk.

Long term, consider requesting the user input an upper bound of the price that he is willing to repay. Revert if the load is higher than expected. Alternatively, remove the feature to repay the entire loan using `uint(-1).`

6. repay calls can be blocked by front-running them with another repay call

Severity: Low
Type: Timing
Target: `LendingPool.sol`

Difficulty: High
Finding ID: TOB-DLP-06

Description

An attacker can make a repay call to revert by front-running it with another repay, forcing the user to redo the transaction.

The repay function (Figure 6.1) is used to return the funds borrowed, even on behalf of other users.

```
 function repay(address _reserve, uint256 _amount, address _onBehalfOf)
external payable nonReentrant {

        uint256 compoundedBorrowBalance =
core.getUserCompoundedBorrowBalance(_reserve, _onBehalfOf);
        uint256 principalBalance =
core.getUserPrincipalBorrowBalance(_reserve, _onBehalfOf);
        uint256 originationFee = core.getUserOriginationFee(_reserve,
_onBehalfOf);

        require(compoundedBorrowBalance > 0, "The user does not have any
borrow pending");

        require(
            compoundedBorrowBalance.add(originationFee) >= _amount ||
_amount == UINT_MAX_VALUE,
            "User is trying to repay more than the total amount of the
loan"
        );
        …
```

Figure 6.1: Header of the repay function in LendingPool.sol

This function requires specifying an amount to repay. If this amount is `uint(-1)`, it will repay the current loan. Otherwise, it will check whether the amount to pay is equal or less than the loan. If it is less than the loan, it is possible for an attacker to block the transaction by front-running the `repay` with another one.

Exploit Scenario

Alice wants to repay her 10 ETH loan. She calls `repay` with the total amount. Bob sees the unconfirmed transaction and front-runs it with another `repay` on her behalf, but using the minimal amount (e.g. 1 WEI). As a result, Alice's transaction reverts, since her loan is less than 10 ETH. She now needs to redo the repay.

Recommendation
Short term, as a simpler alternative, document this behavior, advising users to carefully approve a tight amount of tokens for the LendingPool.

Long term, consider monitoring the blockchain using the Repay events to catch this possible attack.

7. repay does not validate if the user borrows ERC20 tokens, but pays using ether

Severity: Low
Type: Data Validation
Target: `LendingPool.sol`

Difficulty: High
Finding ID: TOB-DLP-07

Description
A user sending ETH to repay ERC20 tokens will lose the ETH sent.

The `repay` function (Figure 7.1) can be used to repay ETH or ERC20 tokens. The call to the `transferToReserve` function (Figure 7.2) is responsible to validate the amount of ether/tokens.

```
  function repay(address _reserve, uint256 _amount, address _onBehalfOf)
external payable nonReentrant {

        uint256 compoundedBorrowBalance =
core.getUserCompoundedBorrowBalance(_reserve, _onBehalfOf);
        uint256 principalBalance =
core.getUserPrincipalBorrowBalance(_reserve, _onBehalfOf);
        uint256 originationFee = core.getUserOriginationFee(_reserve,
_onBehalfOf);

        require(compoundedBorrowBalance > 0, "The user does not have any
borrow pending");

        require(
            compoundedBorrowBalance.add(originationFee) >= _amount ||
_amount == UINT_MAX_VALUE,
            "User is trying to repay more than the total amount of the
loan"
        );
        …
        core.transferToReserve.value(msg.value)(_reserve, _onBehalfOf,
paybackAmountMinusFees);

        emit Repay(_reserve, _onBehalfOf, _amount);
  }
```

Figure 7.1: Part of the repay function in LendingPool.sol. The call to the transferToReserve function has been highlighted with a red color.

However, this function does not check if the user tries to repay ERC20 tokens using ETH.

```
    function transferToReserve(address _reserve, address _user, uint256
_amount) external payable onlyLendingPool {
        CoreLibrary.ReserveData storage reserve = reserves[_reserve];

        require(
            reserve.usageAsCollateralEnabled || reserve.borrowingEnabled,
            "The reserve isn't enabled for borrowing or as collateral"
        );

        if (_reserve != ethereumAddress)
            ERC20(_reserve).safeTransferFrom(_user, address(this),
_amount);
        else require(msg.value == _amount, "The amount and the value sent
to deposit do not match");
    }
```

Figure 7.2: transferToReserve in LendingPoolCore.sol.

The function will take the ERC20 tokens and accept the ETH, without reporting any error.

Exploit Scenario

Alice mistakenly tries to pay her ERC20 tokens loan, using ETH. As a result, the `LendingPool` contract will take her ERC20 tokens, and she will lose her ETH.

Recommendation

Short term, disallow sending ETH if the user has an ERC20 token loan.

Long term, add proper unit tests to ensure that the `LendingPool` contract cannot successfully take ETH and ERC20 tokens from a user at the same time.

8. redeem does not properly validate parameters and allows drainage of funds

Severity: High
Type: Data Validation
Target: `LendingPool.sol`

Difficulty: Low
Finding ID: TOB-DLP-08

Description

An attacker could use the `redeem` function with a specially crafted `AToken` contract to steal all the available liquidity.

The `redeem` function (Figure 8.1) allows users to get back their funds, given the address of a `aToken` contract and the amount to redeem.

```
function redeem(address _aToken, uint256 _aTokenAmount) external nonReentrant {

        AToken aToken = AToken(_aToken);
        address reserve = aToken.getUnderlyingAssetAddress();

        uint256 underlyingAmountToRedeem = 0;
        uint256 aTokenBalance = aToken.balanceOf(msg.sender);
        require(
            aTokenBalance >= _aTokenAmount && _aTokenAmount > 0,
            "The aToken user balance is not enough to cover the amount to redeem or the
amount to redeem is 0"
        );

       //calculate underlying to redeem
        underlyingAmountToRedeem = aToken.aTokenAmountToUnderlyingAmount(_aTokenAmount);
        aToken.burnOnRedeem(msg.sender, _aTokenAmount);

        uint256 currentAvailableLiquidity = core.getReserveAvailableLiquidity(reserve);
        require(currentAvailableLiquidity >= underlyingAmountToRedeem, "There is not enough
liquidity available to redeem");

        //compound liquidity and variable borrow interests
        core.updateReserveCumulativeIndexes(reserve);

        (,,,,,uint256 healthFactor) = dataProvider.calculateUserGlobalData(msg.sender);

        //at this point, the withdraw should not trigger the liquidation
        require(
            healthFactor > dataProvider.getHealthFactorLiquidationThreshold(),
            "Redeem would bring the loan in a liquidation state thus cannot be accepted"
        );

        /**
        @dev update reserve data
         */

        core.decreaseReserveTotalLiquidity(reserve, underlyingAmountToRedeem);
        core.updateReserveInterestRates(reserve);
        core.setReserveLastUpdate(reserve);

        core.transferToUser(reserve, msg.sender, underlyingAmountToRedeem);
        emit Redeem(_aToken, msg.sender, _aTokenAmount, underlyingAmountToRedeem);

}
```

Figure 8.1: `redeem` function in LendingPool.sol

However, since it does not check the validity of the supplied `AToken` contract, it allows any user to control the amount of liquidity to redeem in the `underlyingAmountToRedeem` variable. bec

Exploit Scenario

An attacker can create a specially crafted `AToken` contract to redeem any user funds, given there is enough liquidity. Such contract just return arbitrary amounts when the `balanceOf` and the `aTokenAmountToUnderlyingAmount` functions are called, in order to perform the exploit.

Recommendation

Short term, properly validate the provided `_aToken` address using the corresponding reserve.

Long term, add proper unit tests to simulate this attack, in order to test that current `redeem` implementation is not vulnerable.

9. updateReserveTotalBorrowsByRateMode does not properly validate InterestRateMode inputs

Severity: High
Difficulty: Low
Type: Data Validation
Finding ID: TOB-DLP-09
Target: `LendingPool.sol`

Description

The interest rate mode parameter is not properly validated during the call to `borrow` and can leave the contract in an inconsistent state.

The `borrow` function allows borrowers to specify an `interestRateMode` input, which could be either `FIXED` (0) or `VARIABLE` (1).

```
    function borrow(address _reserve, uint256 _amount, uint256
_interestRateMode, uint16 _referralCode) external nonReentrant {
        BorrowLocalVars memory vars;

        //check that the amount is available in the reserve
        require(
            core.getReserveAvailableLiquidity(_reserve) >= _amount,
            "There is not enough liquidity available in the reserve"
        );

        //check if the borrow mode is fixed and if fixed rate borrowing is
enabled on this reserve
        require(
            _interestRateMode ==
uint256(CoreLibrary.InterestRateMode.VARIABLE) ||
                core.getReserveIsFixedBorrowRateEnabled(_reserve),
            "Fixed borrows rate are not enabled on this reserve"
        );
    …
}
```

Figure 9.1: Header of the borrow function in LendingPool.sol

This function calls `updateReserveTotalBorrowsByRateMode` to update the state variables related to borrows.

```
    function updateReserveTotalBorrowsByRateMode(
        address _reserve,
        address _user,
        uint256 _principalBalance,
        uint256 _balanceIncrease,
        uint256 _newBorrowRateMode,
        uint256 _fixedRate
    ) external onlyLendingPool {
        CoreLibrary.InterestRateMode currentRateMode =
getUserCurrentBorrowRateMode(_reserve, _user);
        CoreLibrary.ReserveData storage reserve = reserves[_reserve];


        if (currentRateMode == CoreLibrary.InterestRateMode.FIXED) {
            // there was no previous borrow or previous borrow was fixed
            CoreLibrary.UserReserveData storage user =
usersReserveData[_user][_reserve];

reserve.decreaseTotalBorrowsFixedAndUpdateAverageRate(_principalBalance,
user.fixedBorrowRate);

            if (_newBorrowRateMode ==
uint256(CoreLibrary.InterestRateMode.FIXED)) {

reserve.increaseTotalBorrowsFixedAndUpdateAverageRate(_principalBalance.ad
d(_balanceIncrease), _fixedRate);
            }  else if (_newBorrowRateMode ==
uint256(CoreLibrary.InterestRateMode.VARIABLE)) {
                //switching the whole amount borrowed to variable

reserve.increaseTotalBorrowsVariable(_principalBalance.add(_balanceIncreas
e));
            }
        } else {
            // previous borrow was variable
            if (_newBorrowRateMode ==
uint256(CoreLibrary.InterestRateMode.FIXED)) {
                reserve.decreaseTotalBorrowsVariable(_principalBalance);

reserve.increaseTotalBorrowsFixedAndUpdateAverageRate(_principalBalance.ad
d(_balanceIncrease), _fixedRate);
            } else if (_newBorrowRateMode ==
uint256(CoreLibrary.InterestRateMode.VARIABLE)) {
                //switching the whole amount borrowed to variable
                reserve.increaseTotalBorrowsVariable(_balanceIncrease);
            }
        }
    }
```

Figure 9.2: updateReserveTotalBorrowsByRateMode function in LendingPoolCore.sol

However, this function does not properly validate the borrow-rate mode used. Therefore, if an invalid input is used, the code will skip important code which modifies the state variables.

Exploit Scenario
An attacker borrows funds, using an invalid borrow-rate mode to force the contract to transition to an invalid state. As a result, the contract must be patched and re-deployed.

Recommendation
Short term, properly validate the `_interestRateMode` input to allow only valid modes.

Long term, add proper unit tests to simulate this attack and test that the current `borrow` implementation is not vulnerable.

10. DefaultReserveInterestRateStrategy parameters are not validated and can block the Lending Pool

Severity: Low
Type: Data Validation
Target: `DefaultReserveInterestRateStrategy.sol`

Difficulty: High
Finding ID: TOB-DLP-10

Description
The lack of parameter validation in `DefaultReserveInterestRateStrategy` can block the Lending Pool, if invalid values are used.

The `DefaultReserveInterestRateStrategy` contract requires a number of parameters to be deployed:

```
contract DefaultReserveInterestRateStrategy is
IReserveInterestRateStrategy {

    using WadRayMath for uint256;
    using SafeMath for uint256;

    uint256 constant FIXED_RATE_INCREASE_THRESHOLD = (10 ** 27)/2; // half
ray

    LendingPoolCore core;
    ILendingRateOracle lendingRateOracle;

    uint256 baseVariableBorrowRate;
    uint256 variableBorrowRateScaling;
    uint256 fixedBorrowRateScaling;
    uint256 borrowToLiquidityRateDelta;
    address reserve;

    constructor(
        address _reserve,
        LendingPoolAddressesProvider _provider,
        uint256 _baseVariableBorrowRate,
        uint256 _variableBorrowRateScaling,
        uint256 _fixedBorrowRateScaling,
        uint256 _borrowToLiquidityRateDelta) public {

        core = LendingPoolCore(_provider.getLendingPoolCore());
        lendingRateOracle =
ILendingRateOracle(_provider.getLendingRateOracle());
        baseVariableBorrowRate = _baseVariableBorrowRate;
        variableBorrowRateScaling = _variableBorrowRateScaling;
        fixedBorrowRateScaling = _fixedBorrowRateScaling;
        borrowToLiquidityRateDelta = _borrowToLiquidityRateDelta;
        reserve = _reserve;
    }
    ...
```

Figure 10.1: Header of the `DefaultReserveInterestRateStrategy` contract in DefaultReserveInterestRateStrategy.sol

These parameters are copied into the state variables and used in several important functions, including `calculateInterestRates`:

```
    function calculateInterestRates(
        address _reserve,
        uint256 _utilizationRate,
        uint256 _totalBorrowsFixed,
        uint256 _totalBorrowsVariable,
        uint256 _averageFixedBorrowRate
        ) external view returns (uint256 currentLiquidityRate, uint256
currentFixedBorrowRate, uint256 currentVariableBorrowRate) {

        currentFixedBorrowRate =
lendingRateOracle.getMarketBorrowRate(_reserve);

        if(_utilizationRate > FIXED_RATE_INCREASE_THRESHOLD){
            currentFixedBorrowRate =
currentFixedBorrowRate.add(fixedBorrowRateScaling.rayMul(_utilizationRate.
sub(FIXED_RATE_INCREASE_THRESHOLD)));
        }

        currentVariableBorrowRate =
_utilizationRate.rayMul(variableBorrowRateScaling).add(
             baseVariableBorrowRate
        );

        currentLiquidityRate = getOverallBorrowRateInternal(
            _totalBorrowsFixed,
            _totalBorrowsVariable,
            currentVariableBorrowRate,
            _averageFixedBorrowRate).rayMul(_utilizationRate);

    }
```

Figure 10.2: calculateInterestRates function in DefaultReserveInterestRateStrategy.sol

However the parameters are not validated and cannot be changed by anyone (not even the owner of the contract), if they need to be adjusted. If the parameters use some invalid values, the call to `calculateInterestRates` could revert, potentially blocking the LendingPool contract.

Exploit Scenario

The `DefaultReserveInterestRateStrategy` is deployed using some invalid parameters, causing unexpected reverts in calculateInterestRates. As a result, the important functions in the Lending Pool such as `borrow`, `deposit`, `redeem` and `repay` will revert, forcing the Lending Pool to be patched and re-deployed.

Recommendation

Short term, properly define input ranges and validate the `DefaultReserveInterestRateStrategy` contract parameters on-chain.

Long term, add proper unit tests to ensure that `DefaultReserveInterestRateStrategy` deployment will fail with invalid parameters. Consider using Echidna and Manticore to test the constructor implementation and make sure it will only accept valid parameters.

11. There is no way to redeem all available tokens when Lending Pool parameters are close to the redemption limit

Severity: Medium
Type: Data Validation
Target: `LendingPool.sol`

Difficulty: Low
Finding ID: TOB-DLP-11

Description
If the available liquidity or health metric are close to the acceptable limits, the redeem function provides no way to specify to redeem the maximum amount of available tokens, forcing the user to guess this amount.

Users can get back their deposited funds using the `redeem` function:

```
   function redeem(address _aToken, uint256 _aTokenAmount) external
nonReentrant {

        AToken aToken = AToken(_aToken);
        address reserve = aToken.getUnderlyingAssetAddress();

        uint256 underlyingAmountToRedeem = 0;
        uint256 aTokenBalance = aToken.balanceOf(msg.sender);
        require(
            aTokenBalance >= _aTokenAmount && _aTokenAmount > 0,
            "The aToken user balance is not enough to cover the amount to
redeem or the amount to redeem is 0"
        );

        //calculate underlying to redeem
        underlyingAmountToRedeem =
aToken.aTokenAmountToUnderlyingAmount(_aTokenAmount);
        aToken.burnOnRedeem(msg.sender, _aTokenAmount);

        uint256 currentAvailableLiquidity =
core.getReserveAvailableLiquidity(reserve);
        require(currentAvailableLiquidity >= underlyingAmountToRedeem,
"There is not enough liquidity available to redeem");

        //compound liquidity and variable borrow interests
        core.updateReserveCumulativeIndexes(reserve);

        (,,,,,uint256 healthFactor) =
dataProvider.calculateUserGlobalData(msg.sender);

        //at this point, the withdraw should not trigger the liquidation
        require(
            healthFactor >
dataProvider.getHealthFactorLiquidationThreshold(),
            "Redeem would bring the loan in a liquidation state thus
cannot be accepted"
        );

        /**
        @dev update reserve data
         */

        core.decreaseReserveTotalLiquidity(reserve,
underlyingAmountToRedeem);
        core.updateReserveInterestRates(reserve);
        core.setReserveLastUpdate(reserve);

        core.transferToUser(reserve, msg.sender,
underlyingAmountToRedeem);

        emit Redeem(_aToken, msg.sender, _aTokenAmount,
underlyingAmountToRedeem);

    }
```

Figure 11.1: redeem function in LendingPool.sol

However, redemption is restricted: it should have enough liquidity available and the health liquidation factor should be above a certain threshold. When a user tries to redeem all their tokens while the Lending Pool parameters are close to the redemption limit, it is very difficult for the user to determine how many tokens can be redeemed. There is no option to specify to redeem as much as possible, as in the `repay` function. This can be a significant obstacle for the user trying to redeem their tokens.

Exploit Scenario

Alice has a large amount of tokens. During a bank run, she tries to redeem all of them. She repeatedly fails to do so, since she specifies too many tokens to redeem. As a result, Alice is forced to redeem her tokens in small batches, which costs more gas.

Recommendation

Short term, properly implement a mechanism to redeem all the available tokens, given the current liquidity and health liquidation factor. As an alternative, document this behavior to warn users about this constraint.

Long term, consider using Echidna and Manticore to test that users can redeem the maximum amount of their token using `redeem`.

12. Race condition on LendingPool allows attacker to redeem their token first

Severity: Medium
Type: Timing
Target: `LendingPool.sol`

Difficulty: High
Finding ID: TOB-DLP-12

Description

A race condition on the LendingPool contract allows an attacker to front-run a large token redemption during a bank run so he can redeem his own tokens first.

User can get back their deposited funds using the `redeem` function:

```
   function redeem(address _aToken, uint256 _aTokenAmount) external
nonReentrant {

        AToken aToken = AToken(_aToken);
        address reserve = aToken.getUnderlyingAssetAddress();

        uint256 underlyingAmountToRedeem = 0;
        uint256 aTokenBalance = aToken.balanceOf(msg.sender);
        require(
            aTokenBalance >= _aTokenAmount && _aTokenAmount > 0,
            "The aToken user balance is not enough to cover the amount to
redeem or the amount to redeem is 0"
        );

        //calculate underlying to redeem
        underlyingAmountToRedeem =
aToken.aTokenAmountToUnderlyingAmount(_aTokenAmount);
        aToken.burnOnRedeem(msg.sender, _aTokenAmount);

        uint256 currentAvailableLiquidity =
core.getReserveAvailableLiquidity(reserve);
        require(currentAvailableLiquidity >= underlyingAmountToRedeem,
"There is not enough liquidity available to redeem");

        //compound liquidity and variable borrow interests
        core.updateReserveCumulativeIndexes(reserve);

        (,,,,,uint256 healthFactor) =
dataProvider.calculateUserGlobalData(msg.sender);

        //at this point, the withdraw should not trigger the liquidation
        require(
            healthFactor >
dataProvider.getHealthFactorLiquidationThreshold(),
            "Redeem would bring the loan in a liquidation state thus
cannot be accepted"
        );

        /**
        @dev update reserve data
         */

        core.decreaseReserveTotalLiquidity(reserve,
underlyingAmountToRedeem);
        core.updateReserveInterestRates(reserve);
        core.setReserveLastUpdate(reserve);

        core.transferToUser(reserve, msg.sender,
underlyingAmountToRedeem);

        emit Redeem(_aToken, msg.sender, _aTokenAmount,
underlyingAmountToRedeem);

    }
```

Figure 12.1: redeem function in LendingPool.sol

In Ethereum, all transactions appear on the network before being accepted. Users can see upcoming token redemptions. As a result, an attacker can front-run large token redemptions to make sure his own tokens will be redeemed.

Exploit Scenario
Alice has a large amount of tokens. During a bank run, she tries to redeem all of them. Bob sees the unconfirmed redeem transaction and front-runs her `redeem` call with his own tokens, using a higher gas-price.

Alice's transaction reverts if the available liquidity or the health liquidity factor are too low, since Bob's redemption will be performed first. As a result, Alice's `redeem` call reverts, and she cannot get her funds back.

Recommendation
Short term, document this behavior to make sure users are aware of redemption front-runs that can be exploited by others. On large redemptions, consider:

- Warning off-chain about this behavior
- Recommending an increase to the gas price or splitting the transaction into several smaller ones to avoid detection by potential attackers

Long term, carefully consider the unpredictable nature of Ethereum transactions, and design contracts to not depend on the order of transactions.

13. Insufficient validation of reserve address in some LendingPoolCore functions

Severity: Informational
Type: Data Validation
Target: `LendingPoolCore.sol`

Difficulty: Low
Finding ID: TOB-DLP-13

Description

Several functions within the `LendingPoolCore` contract operate on structs stored in mappings. The `_reserve` and `_user` function parameters determine which particular struct is modified. These functions currently do not validate that the struct being accessed has been initialized. As a result, a Lending Pool contract may unintentionally access uninitialized data, and then read or modify it instead of the intended data.

```
 function increaseUserOriginationFee(address _reserve, address _user, uint256 _amount)
external onlyLendingPool {
      CoreLibrary.UserReserveData storage user = usersReserveData[_user][_reserve];
      user.originationFee = user.originationFee.add(_amount);
 }

 function decreaseUserOriginationFee(address _reserve, address _user, uint256 _amount)
external onlyLendingPool {
      CoreLibrary.UserReserveData storage user = usersReserveData[_user][_reserve];
      user.originationFee = user.originationFee.sub(_amount);
 }
```

Figure 13.1: Example functions missing validation checks.

```
 function updateUserFixedBorrowRate(address _reserve, address _user) external onlyLendingPool
{
     CoreLibrary.ReserveData storage reserve = reserves[_reserve];
     CoreLibrary.UserReserveData storage user = usersReserveData[_user][_reserve];
     user.fixedBorrowRate = getReserveCurrentFixedBorrowRate(_reserve);
 }

 function resetUserFixedBorrowRate(address _reserve, address _user) external onlyLendingPool {
     CoreLibrary.UserReserveData storage user = usersReserveData[_user][_reserve];
     user.fixedBorrowRate = 0;
 }
```

Figure 13.2: Further examples of functions which do not validate their input data.

Exploit Scenario

The Aave developers decide to extend the Lending Pool contract, using the `LendingPoolCore` functions. However, they do not properly check whether the reserve address and the returned data from `LendingPoolCore` are initialized. As a result, the use of the new code can potentially produce unexpected results.

Recommendations

Short term, add missing validation checks to the relevant functions in `LendingPoolCore`.

Long term, always ensure data has been initialized before accessing it.

14. Users should check that test mode is disabled before interacting with the Lending Pool

Severity: Informational
Type: Access Controls
Target: `LendingPoolCore.sol`

Difficulty: High
Finding ID: TOB-DLP-14

Description

Users should always check that test mode in the LendingPoolCore is disabled before interacting with the contracts.

The LendingPoolCore contract defines a special mode called "test mode," which allows any user to modify the current block number using `forwardTime` function:

```
    /**
     * @notice test function to move forward time, only if testMode is
enabled
     */

     function isTestModeEnabled() external view returns(bool) {
         return testModeEnabled;
     }

     function forwardTime(uint256 newBlockNumber) external {

         require(testModeEnabled == true, "This function can be called only
in test mode");
         testBlockNumber = newBlockNumber;
     }


     function getLastBlockNumber() internal view returns(uint256) {
         return testModeEnabled ? testBlockNumber : block.number;
     }
```

Figure 14.1: testMode-related function in LendingPoolCore.sol

Any user interacting with the Lending Pool should check that test mode is disabled before using it. This creates a trust issue, since the deployed contract includes this code.

Exploit Scenario

Alice is considering depositing or borrowing funds in the Lending Pool. She found the `LendingPoolCore` includes some functions related to a "test mode," which allows changing important information like the last block number. Unless this is properly explained to the users, and they check that test mode is disabled, many will choose to not interact with the Lending Pool. As a result, Alice distrusts the Aave developers and decides not to interact with their smart contracts.

Recommendation
Short term, remove the test mode, or refactor it to a separate contract that uses inheritance to expose this special function.

Long term, review all capabilities in smart contracts to make sure they don’t include any debugging code that could be used as a backdoor.

15. swapBorrowRateMode does not check if reserve allows fixed interest rates

Severity: High
Type: Access Controls
Target: LendingPool.sol

Difficulty: Low
Finding ID: TOB-DLP-15

Description
The swapBorrowRateMode allows changing the interest rate mode of any reserve to fixed, even if it is disallowed by the reserve itself.

The borrow function allows specifying an interestRateMode input, which could be either FIXED (0) or VARIABLE (1). Only some reserves allow the fixed interest rate mode.
The swapBorrowRateMode function allows changing from one mode to the other:

```
function swapBorrowRateMode(address _reserve) external nonReentrant {

        CoreLibrary.InterestRateMode currentRateMode =
core.getUserCurrentBorrowRateMode(_reserve, msg.sender);
        uint256 compoundedBorrowBalance =
core.getUserCompoundedBorrowBalance(_reserve, msg.sender);
        require(compoundedBorrowBalance > 0, "User does not have a borrow
in progress on this reserve");

        uint256 currentPrincipalBorrowBalance =
core.getUserPrincipalBorrowBalance(_reserve, msg.sender);
        uint256 balanceIncrease =
compoundedBorrowBalance.sub(currentPrincipalBorrowBalance);

        //compounding reserve indexes
        core.updateReserveCumulativeIndexes(_reserve);
        core.updateUserLastVariableBorrowCumulativeIndex(_reserve,
msg.sender);

        if(currentRateMode == CoreLibrary.InterestRateMode.FIXED){

            uint256 userFixedRate =
core.getUserCurrentFixedBorrowRate(_reserve, msg.sender);

            //switch to variable

core.decreaseReserveTotalBorrowsFixedAndUpdateAverageRate(_reserve,
currentPrincipalBorrowBalance, userFixedRate); //decreasing fixed from old
principal balance
            core.increaseReserveTotalBorrowsVariable(_reserve,
compoundedBorrowBalance); //increase variable borrows
```

```
            core.resetUserFixedBorrowRate(_reserve, msg.sender);
        }
        else {
            //switch to fixed
            uint256 currentFixedRate =
core.getReserveCurrentFixedBorrowRate(_reserve);
            core.decreaseReserveTotalBorrowsVariable(_reserve,
currentPrincipalBorrowBalance);

core.increaseReserveTotalBorrowsFixedAndUpdateAverageRate(_reserve,
compoundedBorrowBalance, currentFixedRate);
            core.updateUserFixedBorrowRate(_reserve,msg.sender);
        }

        core.increaseReserveTotalLiquidity(_reserve, balanceIncrease);
        core.updateReserveInterestRates(_reserve);


        //compounding cumulated interest
        core.increaseUserPrincipalBorrowBalance(_reserve, msg.sender,
balanceIncrease);
        core.setReserveLastUpdate(_reserve);
        core.setUserLastUpdate(_reserve, msg.sender);

        emit Swap(_reserve, msg.sender);
    }
```

Figure 15.1: `swapBorrowRateMode` *function in LendingPool.sol*

However, this function does not check if the reserve allows using the `FIXED` interest rate mode. Therefore, a user can change the mode of her loans by calling this function, without any restriction.

Exploit Scenario

Bob wants to borrow some funds using the fixed interest mode from a particular reserve. However, this reserve is restricted to variable mode. Since he cannot directly borrow using this interest mode, he uses borrows with variable interest rates. Immediately after, he calls `swapBorrowRateMode`. As a result, he obtains a loan with a fixed interest rate, despite it not being allowed.

Recommendation

Short term, properly implement a check in `swapBorrowRateMode` to verify if a reserve can be used in fixed or variable mode.

Long term, consider using Echidna and Manticore to test that users cannot change the interest mode of their loans, if the reserve has restrictions.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important
Medium	Individual user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal

	implications for client
High	Large numbers of users, very bad for client’s reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details or must discover other weaknesses in order to exploit this issue

B. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

Version pragmas

- **Use consistent version pragmas across all sources, preferably based on the version used during development (0.5.6 - as specified in truffle-config.js).** The following version pragmas can be observed across the source code:
  - Migrations.sol#1 declares pragma solidity>=0.4.21<0.6.0
  - tokenization/AToken.sol#1 declares pragma solidity^0.5.2
  - All the other sources use pragma solidity^0.5.0.

AToken

- **Remove the commented code in this contract.** It will increase the readability, especially during a code review.

CoreLibrary

- **Correct the typo in the `geNormalizedIncome` function name**. It will make the name easier to understand.

LendingPool

- **Mark the `uint16 _referral` parameters in `Deposit` and `Borrow` events as indexed.** This will allow external tooling to use bloom filters to quickly determine inclusion in a given block/transaction.
- **In the `repay` function, move the related `newPrincipalBalance` logic into the "`borrowRateMode == CoreLibrary.InterestRateMode.FIXED`" if block.** This will decrease the gas spent for processing repays with `borrowRateMode` set to `VARIABLE`, as shown in Figure B.1.

```
   function repay(address _reserve, uint256 _amount, address _onBehalfOf) external
payable nonReentrant {
        // (...)

        if (borrowRateMode == CoreLibrary.InterestRateMode.FIXED) {
            // (...) ← move the newPrincipalBalance, its check and
resetUserFixedBorrowRate call here
        } else {
            // (...)
        }

        uint256 newPrincipalBalance = core.getUserPrincipalBorrowBalance(_reserve,
_onBehalfOf);
```

```
        if (newPrincipalBalance == 0 && borrowRateMode ==
CoreLibrary.InterestRateMode.FIXED) {
            core.resetUserFixedBorrowRate(_reserve, _onBehalfOf);
        }
        // (...) - the newPrincipalBalance is not used further
    }
```

Figure B.1: The repay function.
The code highlighted in red can be moved to the block highlighted in green.

LendingPoolCore

- **Remove the unused *reserve* local variable in `updateUserFixedBorrowRate` function.** The unnecessary local variable increases gas cost. This fix has been proposed in the [#2 merge request to the Aave Protocol repository](#).

NetworkMetadataProvider

- **Mark `NetworkMetadataProvider` contract as the one that implements INetworkMetadataProvider interface.** This allows external tools to use this information to enhance code analysis or code navigation in IDEs.

FeeProvider

- **The originationFeePercentage lacks documentation comment.** It is unclear how the default value was figured out, or calculated, or that it is stored as WAD.
- **The `originationFeePercentage`'s default value is `2500000000000000`, and this literal is not readable, due to too many digits.** It is hard to immediately see that this value is 25 * 10^{14} or 0.0025 * 10^{18} or, in other words, that it is 0.0025% written in WAD. This literal should be written as `25 * WAD / (10**4)` and moved to a constant to avoid doing calculations in a function and spending additional gas.

C. Property testing using Echidna

Trail of Bits used Echidna, our property-based testing framework, to find logic errors in the Solidity components of the Aave Protocol.

In this appendix, we present a simple example on how to use Echidna to build a custom testing harness for specific components of the Aave Protocol. We define a simple invariant for the `FeeProvider` and perform a random sequence of smart contract transactions in an attempt to cause anomalous behavior and break it.

Figure C.1 shows the Solidity source code used to test that the calculation of the fee in `FeeProvider` will be always larger than 0. We also restricted the borrow amount used to calculate the fee to avoid unrealistic scenarios where the value borrowed is 0 or extremely large. We used the `crytic_positiveFee` function as an invariant to check that the value returned by `calculateLoanOriginationFee` is positive. An example of how to run this test with Echidna appears in Figure C.2.

```
pragma solidity ^0.5.0;

import "../fees/FeeProvider.sol";

contract CryticInterface {
    address internal crytic_owner = address(0x41414141);
    address internal crytic_user = address(0x42424242);
    address internal crytic_attacker = address(0x43434343);
}

contract FeeProviderTest is CryticInterface, FeeProvider {

    uint amount = 100000000;
    constructor() FeeProvider(address(0x0)) public {}

    function setAmount(uint _amount) public {
        amount = _amount;
    }

    function crytic_positiveFee() public returns (bool) {
        if (amount == 0 || amount > 2 ** 64)
          return true;
        if (originationFeePercentage != 2500000000000000)
          return true;

        return this.calculateLoanOriginationFee(msg.sender, amount) > 0;
    }
}
```

Figure C.1: cryticFeeProvider.sol test

```
$ echidna-test . FeeProviderTest --config contracts/crytic/cryticFeeProvider.yaml
...
Analyzing contract: contracts/crytic/cryticFeeProvider.sol:FeeProviderTest
crytic_positiveFee: failed!💥
  Call sequence:
    setAmount(64)
...
```

Figure C.2: An example run of Echidna with the `cryticFeeProvider.sol` *test, including test results.*

As a result of running this test, Echidna finds that using small amounts (e.g., 64) could result in zero-fee loans.

D. Whitepaper Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance the whitepaper and code readability and may prevent the introduction of vulnerabilities in the future.

Throughout

- Several variables defined in the whitepaper do not specify if they have a multiplier (e.g., WAD/RAY or none) while others do. Consider if it would be helpful to do this for all variables.

1.8 Utilization rate

- This is expressed as a fraction with `Lt` (total liquidity) in the denominator. The formula definition does not describe how `Lt = 0` should be handled to avoid attempting division by zero. `getReserveUtilizationRate` in CoreLibrary.sol handles this as a special case, returning 0 in the event of no total liquidity.

1.12 Current fixed borrow rate

- The whitepaper refers to an unknown paragraph for details of how `Rf` should be calculated. The codebase determines this value in `calculateInterestRates` in DefaultReserveInterestRateStrategy.sol by first taking a base value from an oracle, and then scaling it if utilization is sufficiently high.

1.14 Overall borrow rate

- This is expressed as a fraction with `Bt` (total borrows) in the denominator. The formula definition does not describe how `Bt = 0` should be handled to avoid attempting division by zero. `getOverallBorrowRateInternal` in DefaultReserveInterestRateStrategy.sol handles this as a special case, returning 0 in the event of no total borrows.

1.15 Current Liquidity Rate

- Note: “Borrow/Liquidity Rate delta” and “Current Liquidity Rate” are both numbered 1.15
- The whitepaper definition includes subtraction of the Borrow/Liquidity Rate delta. However, the calculation performed during calls to `calculateInterestRates` in DefaultReserveInterestRateStrategy.sol appears to omit this term and only perform the multiplication.

1.21 Compounded borrow balance

- The Compounded borrow balance denominator cannot be 0, however the code checks for that because uninitialized `ReserveData` is read in the `getUserCompoundedBorrowBalance` function when a nonexistent reserve is used:

```
function getUserCompoundedBorrowBalance(address _reserve, address
_user) public view returns (uint256) {
        CoreLibrary.UserReserveData storage user =
usersReserveData[_user][_reserve];
        CoreLibrary.ReserveData storage reserve =
reserves[_reserve];

        return
            CoreLibrary.getCompoundedBorrowBalance(
                user,
                reserve,
                blocksPerYear,
                getLastBlockNumber()

            );
}
```

Then, the uninitialized data is detected when checking for a zero `principalBorrowBalance`:

```
function getCompoundedBorrowBalance(
        CoreLibrary.UserReserveData storage _self,
        CoreLibrary.ReserveData storage _reserve,
        uint256 _blocksPerYear,
        uint256 _currentBlock
    ) internal view returns (uint256) {
        if (_self.principalBorrowBalance == 0) return 0;
        …
}
```

Avoiding the use of uninitialized data will simplify the code, reduce the gas cost and be easier to review in future audits.

E. Fix Log

From September 23, 2019 to September 25, 2019, Trail of Bits reviewed fixes for issues identified in this report. The Aave team addressed or accepted the risk of all discovered issues in their codebase as a result of the assessment. Of those issues, nine were remediated and six were risk-accepted; we reviewed each of the fixes to understand the strength of correctness of the proposed remediation. After verifying the fixes for the issues identified in the report, Trail of Bits also reviewed the new liquidation call functionality and identified no new security issues.

ID	Title	Severity	Status
01	Solidity compiler optimizations can be dangerous	Undetermined	Risk Accepted
02	Lack of access controls in updateReserveTotalBorrowsByRateMode	High	Fixed
03	Lack of access control for LendingPoolAddressesProvider and NetworkMetadataProvider	Medium	Risk Accepted
04	Lack of access control for FeeProvider setter functions	High	Fixed
05	borrow/repay calls race condition can be exploited to repay more than expected	High	Fixed
06	repay calls can be blocked by front-running them with another repay call	Low	Risk Accepted
07	repay does not validate if the user borrows ERC20 tokens, but pays using ether	Low	Fixed
08	redeem does not properly validate parameters and allows drainage of funds	High	Fixed
09	updateReserveTotalBorrowsByRateMode does not properly validate InterestRateMode inputs	High	Fixed
10	DefaultReserveInterestRateStrategy parameters are not validated and can block the Lending Pool	Low	Risk Accepted
11	There is no way to redeem all available tokens when Lending Pool parameters are close to the redemption limit	Medium	Risk Accepted

12	Race condition on LendingPool allows attacker to redeem their token first	Medium	Risk Accepted
13	Insufficient validation of reserve address in some `LendingPoolCore` functions	Informational	Fixed
14	Users should check that test mode is disabled before interacting with the Lending Pool	Informational	Fixed
15	`swapBorrowRateMode` does not check if reserve allows fixed interest rates	High	Fixed

Detailed Fix Log

Finding 1: Solidity compiler optimizations can be dangerous
Risk accepted. Aave indicated that compiler optimizations provide substantial gas savings to users and these savings currently outweigh the risk of optimizer bugs.
https://gitlab.com/aave-tech/dlp/contracts/issues/1

Finding 2: Lack of access controls in updateReserveTotalBorrowsByRateMode
Fixed. Aave added the missing `onlyLendingPool` modifier to the affected function.
https://gitlab.com/aave-tech/dlp/contracts/commit/1ee87735b657229f1d57887fbc8fa4ce8e615156

Finding 3: Lack of access control for `LendingPoolAddressesProvider` and `NetworkMetadataProvider`
Risk accepted. The Aave team acknowledged this issue and plans to use DAOstack to add the missing access controls before release. However, as this governance system was not in place at the time of this assessment or retest, the issue remains open.
https://gitlab.com/aave-tech/dlp/contracts/issues/3

Finding 4: Lack of access control for `FeeProvider` setter functions
Fixed. The Aave team added the suggested `onlyOwner` modifiers to the affected functions.
https://gitlab.com/aave-tech/dlp/contracts/merge_requests/6

Finding 5: `borrow`/`repay` calls race condition can be exploited to repay more than expected
Fixed. The Aave team added a check to require users repaying on behalf of another user to specify the amount to repay instead of using the `UINT_MAX` shortcut.
https://gitlab.com/aave-tech/dlp/contracts/merge_requests/21

Finding 6: `repay` calls can be blocked by front-running them with another `repay` call
Risk accepted. The Aave team indicated that they will provide documentation about this issue and offer support to affected users.
https://gitlab.com/aave-tech/dlp/contracts/issues/6

Finding 7: `repay` does not validate if the user borrowed ERC20 tokens, but pays using ether
Fixed. The Aave team added a check to ensure that no ETH is being sent when repaying an ERC20 loan.
https://gitlab.com/aave-tech/dlp/contracts/merge_requests/15

Finding 8: `redeem` does not properly validate parameters and allows drainage of funds

Fixed. The Aave team restructured the affected function to move some of the logic to the AToken contract and restrict access to the companion function in the LendingPool to the AToken itself.
https://gitlab.com/aave-tech/dlp/contracts/merge_requests/7

Finding 9: updateReserveTotalBorrowsByRateMode does not properly validate InterestRateMode inputs
Fixed. The Aave team added a require statement that checks that the input InterestRateMode is within the correct range.
https://gitlab.com/aave-tech/dlp/contracts/merge_requests/16

Finding 10: DefaultReserveInterestRateStrategy parameters are not validated and can block the Lending Pool
Risk accepted. The Aave team indicated they plan to manually validate new InterestRateStrategy contract parameters after they are deployed, but before enabling support for them.
https://gitlab.com/aave-tech/dlp/contracts/issues/15

Finding 11: There is no way to redeem all available tokens when Lending Pool parameters are close to the redemption limit
Risk accepted. The Aave team indicated that this is expected behavior and that they will provide documentation to that effect to users.
https://gitlab.com/aave-tech/dlp/contracts/issues/16

Finding 12: Race condition on LendingPool allows attacker to redeem their token first
Risk accepted. The Aave team has indicated that they will document this risk for users and assess mitigations for this type of issue going forward.
https://gitlab.com/aave-tech/dlp/contracts/issues/17

Finding 13: Insufficient validation of reserve address in some LendingPoolCore functions
Fixed. The Aave team added a new modifier, onlyActiveReserve, to ensure functions are only executed against reserve data that has been initialized. For gas usage reasons, this modifier was not added to the functions highlighted in the finding description; however, all paths to these functions are otherwise covered by this modifier.
https://gitlab.com/aave-tech/dlp/contracts/issues/18

Finding 14: Users should check that test mode is disabled before interacting with the Lending Pool
Fixed. The Aave team removed the test mode functionality from the codebase.
https://gitlab.com/aave-tech/dlp/contracts/merge_requests/13/

Finding 15: swapBorrowRateMode does not check if reserve allows fixed interest rates

Fixed. The Aave team added a check to ensure the reserve allows fixed borrowing rates when a user attempts to swap their interest rate mode.
https://gitlab.com/aave-tech/dlp/contracts/merge_requests/5

Detailed Issue Discussion

Responses from Aave for risk-accepted and unfixed issues are included as quotes below.

Finding 1: Solidity compiler optimizations can be dangerous

> *We believe that right now, the gas cost savings for the users (which we estimated in roughly 30-40%) outweigh the risk of possible unknown bugs. We will continue monitoring the evolution of Solidity compilers to ensure there are no security concerns.*

Finding 6: `repay` calls can be blocked by front-running them with another `repay` call

> *We will make sure users understand this behavior, and provide support in case of attack. No changes will be applied to the protocol.*

Finding 10: `DefaultReserveInterestRateStrategy` parameters are not validated and can block the Lending Pool

> *The way this will work, InterestRateStrategy contracts will be changed upon governance vote in case specific conditions arise (e.g., sudden market changes). In this case, new `InterestRateStrategy` contracts will be preemptively audited and deployed, and made public before the governance vote. In case an incorrect version is deployed, the reserve can be paused and a new `InterestRateStrategy` contract can be deployed that fixes the issue.*

Finding 11: There is no way to redeem all available tokens when Lending Pool parameters are close to the redemption limit

> *We believe it is a behavior that is intrinsic to the protocol specification. Even real-life banking entities might incur in it in case of a bank run, so it's not dependent on the specific implementation. We also provide tokenization functions to allow users, to a certain degree, to use* `aTokens` *instead of the underlying asset if necessity arises. We will make sure to properly document this behavior.*

Finding 12: Race condition on LendingPool allows attacker to redeem their token first

> *We believe that, given the transactional nature of the ethereum blockchain, how the current transaction execution is handled, and how the protocol works, there is no effective solution to this. We will make sure to inform users about this behavior and keep following innovations at the blockchain level that might help alleviate this issue in the future.*