# protocol-v2 (AAVE)

Benchmark Security Review Performed by Bug Hunter
27 May 2025

# About Bug Hunter

Bug Hunter is an automated code review tool for Solidity smart contracts, developed by Truscova GmbH. It uses proprietary algorithms and advanced code analysis techniques to identify potential vulnerabilities efficiently.

Bug Hunter is designed to support developers in the Ethereum ecosystem by delivering consistent, scalable assessments that align with best practices in Web 3.0 security. Reports are generated via the Bug Hunter tool and are accessible to clients through a secure dashboard web interface or relevant Git Repositories.

Bug Hunter is a product of Truscova, a security R&D company headquartered in Bremen, Germany. Founded in late 2022, Truscova specializes in formal verification, fuzzing, dynamic and static analysis, and security tooling. Its founding team includes researchers with 35+ books, several patents, and over 20,000 citations in software testing, verification, security, and machine learning.

To explore more about Bug Hunter, visit

- Bug Hunter Website
- Bug Hunter Documentation
- Example reports

To explore more about Truscova, visit

- Truscova Website
- Twitter
- LinkedIn

# Notices and Remarks

## Copyright and Distribution



This security review report was generated by using Truscova's Bug Hunter tool to benchmark its performance on selected Yield Farming platforms. It is made available through the Bug Hunter dashboard and relevant Git repositories. Users may choose to share or publish this report at their own discretion.

## Test Coverage Disclaimer

This security review was conducted exclusively by Bug Hunter, an automated code review tool for Solidity codebases. Bug Hunter uses a combination of proprietary algorithms and programmatic techniques to identify potential vulnerabilities in Solidity smart contracts.

The findings in this report should not be considered a complete or exhaustive list of all possible security issues in the reviewed codebase and is dependent on Bug Hunter's coverage of detectors rules. The complete list of detectors rules covered by Bug Hunter automated code review is available at: https://docs.bughunter.live/.

# Contents

# Executive Summary

## Engagement Overview and Scope

A codebase from project AAVE was reviewed for security vulnerabilities by Bug Hunter with the following details:

- GitHub Repository: https://github.com/aave/protocol-v2
- Commit hash: 12d97f9f13a3f04c206c6a72b93c23126b869572

Bug Hunter reviewed the following files:

- LendingPoolAddressesProvider.sol
- LendingPoolAddressesProviderRegistry.sol
- IAaveIncentivesController.sol
- IChainlinkAggregator.sol
- IERC20.sol
- IERC20Detailed.sol
- IExchangeAdapter.sol
- ILendingPool.sol
- ILendingPoolAddressesProvider.sol
- ILendingPoolAddressesProviderRegistry.sol
- ILendingRateOracle.sol
- IPriceOracle.sol
- IPriceOracleGetter.sol
- IReserveInterestRateStrategy.sol
- ISwapAdapter.sol
- IUniswapExchange.sol
- DefaultReserveInterestRateStrategy.sol
- LendingPool.sol
- LendingPoolCollateralManager.sol
- LendingPoolConfigurator.sol
- LendingPoolStorage.sol
- ReserveConfiguration.sol
- UserConfiguration.sol
- GenericLogic.sol
- ReserveLogic.sol
- ValidationLogic.sol
- Errors.sol
- Helpers.sol
- GenericLogic.sol
- ReserveLogic.sol
- ValidationLogic.sol
- MathUtils.sol
- PercentageMath.sol
- Context.sol
- IERC20DetailedBytes.sol

- AToken.sol
- IncentivizedERC20.sol
- StableDebtToken.sol
- VariableDebtToken.sol
- DebtTokenBase.sol
- IAToken.sol
- IScaledBalanceToken.sol
- IStableDebtToken.sol
- IVariableDebtToken.sol

## Summary of Findings

The uncovered vulnerabilities in codebase during the security review are summarized in the following table:

| Severity | Count |
|---|---|
| High | 0 |
| Medium | 4 |
| Low | 21 |

## Project Summary

## Contact Information

In case of support, contact us at support@bughunter.live

## Project Timeline

| Date | Event |
|---|---|
| 27.05.2025 | Project submitted |
| 27.05.2025 | Security review concluded |

# Detailed Findings

## 1. Incorrect Bitmask in LIQUIDATION_BONUS_MASK

| Severity | Medium |
|---|---|
| Finding ID | BH-protocol-v2-01 |
| Target | contracts/libraries/configuration/ReserveConfiguration.sol |
| Function name | - |

### Description

The bitmask defined in `LIQUIDATION_BONUS_MASK` is missing a leading 'F', resulting in an incorrect value of `0xFFFFFFF0000FFFFFFFF` instead of the intended `0xFFFFFFFF0000FFFFFFFF`. This causes the upper four bits of the field to be corrupted during operations on the liquidation bonus field. Consequently, critical parameters may be altered unintentionally, leading to potential misconfigurations or exploitation.

```
uint256 constant LIQUIDATION_BONUS_MASK = 0xFFFFFFF0000FFFFFFFF;
```

### Recommendation

Update the `LIQUIDATION_BONUS_MASK` constant to include the missing 'F' at the start of the mask, ensuring it accurately preserves all unrelated bits. Below is the corrected Solidity code

```
uint256 constant LIQUIDATION_BONUS_MASK = 0xFFFFFFFF0000FFFFFFFF;
```

## 2. Missing Bit-Length Validation in setLtv Allows Data Corruption

| | |
|---|---|
| Severity | Medium |
| Finding ID | BH-protocol-v2-02 |
| Target | contracts/libraries/configuration/ReserveConfiguration.sol |
| Function name | setLtv |

### Description

The `setLtv` function directly assigns the `ltv` parameter to a bit-packed storage field without validating that it fits within the intended bit length. The LTV (Loan-to-Value) field is expected to be 16 bits wide, meaning it should only accept values from 0 to 65535. Providing a larger value results in overflow into adjacent fields, such as the liquidation threshold, leading to unintended state corruption. This creates a high-severity risk as it could destabilize the risk parameters of the reserve.

```
function setLtv(ReserveConfiguration.Map memory self, uint256 ltv) internal pure {
    self.data = (self.data & LTV_MASK) | ltv;
  }
```

### Recommendation

Add validation to ensure the provided `ltv` value fits within the 16-bit field before modifying the storage. This prevents corruption of other fields and ensures proper configuration boundaries. Below is the patched Solidity code.

```
function setLtv(ReserveConfiguration.Map memory self, uint256 ltv) internal pure {
    require(ltv <= 65535, "Invalid LTV: exceeds 16 bits");
    self.data = (self.data & LTV_MASK) | ltv;
}
```

## 3. Uninitialized Variable 'availableLiquidity' Used in validateBorrow

| Severity | Medium |
|---|---|
| Finding ID | BH-protocol-v2-03 |
| Target | contracts/libraries/logic/ValidationLogic.sol |
| Function name | validateBorrow |

### Description

In the `validateBorrow` function, the variable `vars.availableLiquidity` is used in a calculation to determine the maximum stable loan size (`maxLoanSizeStable`) but is never initialized. Using an uninitialized storage or memory variable in Solidity results in a default value of zero, which can lead to logic errors. In this case, the uninitialized value will cause the `amount <= maxLoanSizeStable` check to always fail, effectively blocking valid stable rate borrowing attempts. This leads to unexpected behavior and may prevent users from interacting with the protocol as intended.

```
function validateBorrow(
    ReserveLogic.ReserveData storage reserve,
    address userAddress,
    uint256 amount,
    uint256 amountInETH,
    uint256 interestRateMode,
    uint256 maxStableLoanPercent,
    mapping(address => ReserveLogic.ReserveData) storage reservesData,
    UserConfiguration.Map storage userConfig,
    address[] calldata reserves,
    address oracle
  ) external view {
    ValidateBorrowLocalVars memory vars;

    (
      vars.isActive,
      vars.isFreezed,
      vars.borrowingEnabled,
      vars.stableRateBorrowingEnabled
    ) = reserve.configuration.getFlags();

    require(vars.isActive, Errors.NO_ACTIVE_RESERVE);
    require(!vars.isFreezed, Errors.NO_UNFREEZED_RESERVE);

    require(vars.borrowingEnabled, Errors.BORROWING_NOT_ENABLED);

    //validate interest rate mode
    require(
      uint256(ReserveLogic.InterestRateMode.VARIABLE) == interestRateMode ||
```

```
      uint256(ReserveLogic.InterestRateMode.STABLE) == interestRateMode,
      Errors.INVALID_INTEREST_RATE_MODE_SELECTED
    );

    (
      vars.userCollateralBalanceETH,
      vars.userBorrowBalanceETH,
      vars.currentLtv,
      vars.currentLiquidationThreshold,
      vars.healthFactor
    ) = GenericLogic.calculateUserAccountData(
      userAddress,
      reservesData,
      userConfig,
      reserves,
      oracle
    );

    require(vars.userCollateralBalanceETH > 0, Errors.COLLATERAL_BALANCE_IS_0);

    require(
      vars.healthFactor > GenericLogic.HEALTH_FACTOR_LIQUIDATION_THRESHOLD,
      Errors.HEALTH_FACTOR_LOWER_THAN_LIQUIDATION_THRESHOLD
    );

    //add the current already borrowed amount to the amount requested to calculate the
total collateral needed.
    vars.amountOfCollateralNeededETH =
vars.userBorrowBalanceETH.add(amountInETH).percentDiv(
      vars.currentLtv
    ); //LTV is calculated in percentage

    require(
      vars.amountOfCollateralNeededETH <= vars.userCollateralBalanceETH,
      Errors.COLLATERAL_CANNOT_COVER_NEW_BORROW
    );

    /**
     * Following conditions need to be met if the user is borrowing at a stable rate:
     * 1. Reserve must be enabled for stable rate borrowing
     * 2. Users cannot borrow from the reserve if their collateral is (mostly) the same currency
     *    they are borrowing, to prevent abuses.
     * 3. Users will be able to borrow only a relatively small, configurable amount of the total
     *    liquidity
     **/

    if (vars.rateMode == ReserveLogic.InterestRateMode.STABLE) {
```

```
    //check if the borrow mode is stable and if stable rate borrowing is enabled on this
reserve

    require(vars.stableRateBorrowingEnabled,
Errors.STABLE_BORROWING_NOT_ENABLED);

    require(
      !userConfig.isUsingAsCollateral(reserve.id) ||
        reserve.configuration.getLtv() == 0 ||
        amount > IERC20(reserve.aTokenAddress).balanceOf(userAddress),
      Errors.CALLATERAL_SAME_AS_BORROWING_CURRENCY
    );

    //calculate the max available loan size in stable rate mode as a percentage of the
    //available liquidity
    uint256 maxLoanSizeStable =
vars.availableLiquidity.percentMul(maxStableLoanPercent);

    require(amount <= maxLoanSizeStable,
Errors.AMOUNT_BIGGER_THAN_MAX_LOAN_SIZE_STABLE);
  }
 }
```

## Recommendation

Initialize `vars.availableLiquidity` before it is used. The value should be fetched from the reserve's current liquidity, typically by querying the reserve's aToken balance.

## 4. Incorrect Calculation of Previous Stable Debt in _mintToTreasury Causes Debt Accrual Underestimation

| Severity | Medium |
|---|---|
| Finding ID | BH-protocol-v2-04 |
| Target | contracts/libraries/logic/ReserveLogic.sol |
| Function name | _mintToTreasury |

### Description

The `_mintToTreasury` function incorrectly calculates `vars.previousStableDebt` using the compounded interest on `vars.principalStableDebt`, resulting in a value that is equal or close to `vars.currentStableDebt`. This leads to the incorrect assumption that no additional stable debt has accrued since the last update, thereby excluding stable debt growth from treasury minting calculations. As a result, the protocol underestimates total debt accrual and under-mints tokens to the treasury. This can lead to a financial imbalance and lost protocol revenue over time.

```
function _mintToTreasury(
    ReserveData storage reserve,
    address variableDebtToken,
    uint256 previousVariableBorrowIndex,
    uint256 newLiquidityIndex,
    uint256 newVariableBorrowIndex
  ) internal {
    MintToTreasuryLocalVars memory vars;

    vars.reserveFactor = reserve.configuration.getReserveFactor();

    if (vars.reserveFactor == 0) {
      return;
    }

    //fetching the last scaled total variable debt
    vars.scaledVariableDebt = IVariableDebtToken(variableDebtToken).scaledTotalSupply();

    //fetching the principal, total stable debt and the avg stable rate
    (
      vars.principalStableDebt,
      vars.currentStableDebt,
      vars.avgStableRate,
      vars.stableSupplyUpdatedTimestamp
    ) = IStableDebtToken(reserve.stableDebtTokenAddress).getSupplyData();

    //calculate the last principal variable debt
```

```
    vars.previousVariableDebt =
vars.scaledVariableDebt.rayMul(previousVariableBorrowIndex);

    //calculate the new total supply after accumulation of the index
    vars.currentVariableDebt = vars.scaledVariableDebt.rayMul(newVariableBorrowIndex);

    //calculate the stable debt until the last timestamp update
    vars.cumulatedStableInterest = MathUtils.calculateCompoundedInterest(
      vars.avgStableRate,
      vars.stableSupplyUpdatedTimestamp
    );

    vars.previousStableDebt =
vars.principalStableDebt.rayMul(vars.cumulatedStableInterest);

    //debt accrued is the sum of the current debt minus the sum of the debt at the last
update
    vars.totalDebtAccrued = vars
      .currentVariableDebt
      .add(vars.currentStableDebt)
      .sub(vars.previousVariableDebt)
      .sub(vars.previousStableDebt);

    vars.amountToMint = vars.totalDebtAccrued.percentMul(vars.reserveFactor);

    IAToken(reserve.aTokenAddress).mintToTreasury(vars.amountToMint,
newLiquidityIndex);
  }
```

## Recommendation

Track the actual previous stable debt at the last update or store and retrieve it from state if available. If previous stable debt is not tracked, consider revising the debt accrual logic to ensure accurate debt growth calculations. One possible workaround is to update protocol design to explicitly store historical debt data or timestamps at minting checkpoints.

## 5. Unoptimized Access of Storage Array in getAddressesProvidersList Increases Gas Costs

| Severity | Low |
|---|---|
| Finding ID | BH-protocol-v2-05 |
| Target | contracts/configuration/LendingPoolAddressesProviderRegistry.sol |
| Function name | getAddressesProvidersList |

### Description

The `getAddressesProvidersList` function accesses the storage array `addressesProvidersList` multiple times within a loop. Each access to a storage array element requires multiple SLOAD operations due to overflow and bounds checks, resulting in inefficient gas usage. Specifically, accessing `addressesProvidersList[i]` within the loop causes 2 SLOADs per iteration, and checking the length in the loop condition incurs another SLOAD, leading to 3 * N total SLOADs. This inefficiency can be improved by caching the storage array into memory.

```
function getAddressesProvidersList() external override view returns (address[] memory) {
    uint256 maxLength = addressesProvidersList.length;

    address[] memory activeProviders = new address[](maxLength);

    for (uint256 i = 0; i < addressesProvidersList.length; i++) {
      if (addressesProviders[addressesProvidersList[i]] > 0) {
        activeProviders[i] = addressesProvidersList[i];
      }
    }

    return activeProviders;
  }
```

### Recommendation

To optimize, cache the storage array `addressesProvidersList` into a memory variable before iterating over it. This reduces storage reads from O(N) to O(1) for the loop bound and O(N) for element access.

## 6. Incorrect Bitmask in STABLE_BORROWING_MASK May Affect Future Bit Field Usage

| Severity | Low |
|---|---|
| Finding ID | BH-protocol-v2-6 |
| Target | contracts/libraries/configuration/ReserveConfiguration.sol |
| Function name | - |

### Description

The bitmask `STABLE_BORROWING_MASK` is incorrectly defined as `0xFFFF07FFFFFFFFFFFFFF`. The correct value should be `0xFFFFF7FFFFFFFFFFFFFF`, which properly clears the bit at the correct offset. If those bits are repurposed in the future, the current mask could inadvertently overwrite them during bit manipulations, leading to subtle and hard-to-debug corruption of configuration data.

```
uint256 constant STABLE_BORROWING_MASK = 0xFFFF07FFFFFFFFFFFFFF;
```

### Recommendation

Update the bitmask to the correct value `0xFFFFF7FFFFFFFFFFFFFF` to ensure only the intended bit is masked.

## 7. Missing Zero-Amount Check in validateBorrow Allows Redundant or Risky Borrow Calls

| Severity | Low |
|---|---|
| Finding ID | BH-protocol-v2-7 |
| Target | contracts/libraries/logic/ValidationLogic.sol |
| Function name | validateBorrow |

### Description

The `validateBorrow` function does not validate whether the `amount` parameter is greater than zero. As a result, borrowers can call this function with a zero amount, which might pass all existing validations and proceed with unnecessary or unintended logic execution. While this might not cause direct fund loss, it can lead to wasted gas, pollute system metrics, and may open the door to edge case exploits in future versions or integrations that assume positive borrow amounts.

```
function validateBorrow(
    ReserveLogic.ReserveData storage reserve,
    address userAddress,
    uint256 amount,
    uint256 amountInETH,
    uint256 interestRateMode,
    uint256 maxStableLoanPercent,
    mapping(address => ReserveLogic.ReserveData) storage reservesData,
    UserConfiguration.Map storage userConfig,
    address[] calldata reserves,
    address oracle
  ) external view {
    ValidateBorrowLocalVars memory vars;

    (
      vars.isActive,
      vars.isFreezed,
      vars.borrowingEnabled,
      vars.stableRateBorrowingEnabled
    ) = reserve.configuration.getFlags();

    require(vars.isActive, Errors.NO_ACTIVE_RESERVE);
    require(!vars.isFreezed, Errors.NO_UNFREEZED_RESERVE);

    require(vars.borrowingEnabled, Errors.BORROWING_NOT_ENABLED);

    //validate interest rate mode
    require(
      uint256(ReserveLogic.InterestRateMode.VARIABLE) == interestRateMode ||
```

```
      uint256(ReserveLogic.InterestRateMode.STABLE) == interestRateMode,
      Errors.INVALID_INTEREST_RATE_MODE_SELECTED
    );

    (
      vars.userCollateralBalanceETH,
      vars.userBorrowBalanceETH,
      vars.currentLtv,
      vars.currentLiquidationThreshold,
      vars.healthFactor
    ) = GenericLogic.calculateUserAccountData(
      userAddress,
      reservesData,
      userConfig,
      reserves,
      oracle
    );

    require(vars.userCollateralBalanceETH > 0, Errors.COLLATERAL_BALANCE_IS_0);

    require(
      vars.healthFactor > GenericLogic.HEALTH_FACTOR_LIQUIDATION_THRESHOLD,
      Errors.HEALTH_FACTOR_LOWER_THAN_LIQUIDATION_THRESHOLD
    );

    //add the current already borrowed amount to the amount requested to calculate the
total collateral needed.
    vars.amountOfCollateralNeededETH =
vars.userBorrowBalanceETH.add(amountInETH).percentDiv(
      vars.currentLtv
    ); //LTV is calculated in percentage

    require(
      vars.amountOfCollateralNeededETH <= vars.userCollateralBalanceETH,
      Errors.COLLATERAL_CANNOT_COVER_NEW_BORROW
    );

    /**
     * Following conditions need to be met if the user is borrowing at a stable rate:
     * 1. Reserve must be enabled for stable rate borrowing
     * 2. Users cannot borrow from the reserve if their collateral is (mostly) the same currency
     *    they are borrowing, to prevent abuses.
     * 3. Users will be able to borrow only a relatively small, configurable amount of the total
     *    liquidity
     **/

    if (vars.rateMode == ReserveLogic.InterestRateMode.STABLE) {
```

```
    //check if the borrow mode is stable and if stable rate borrowing is enabled on this
reserve

    require(vars.stableRateBorrowingEnabled,
Errors.STABLE_BORROWING_NOT_ENABLED);

    require(
      !userConfig.isUsingAsCollateral(reserve.id) ||
        reserve.configuration.getLtv() == 0 ||
        amount > IERC20(reserve.aTokenAddress).balanceOf(userAddress),
      Errors.CALLATERAL_SAME_AS_BORROWING_CURRENCY
    );

    //calculate the max available loan size in stable rate mode as a percentage of the
    //available liquidity
    uint256 maxLoanSizeStable =
vars.availableLiquidity.percentMul(maxStableLoanPercent);

    require(amount <= maxLoanSizeStable,
Errors.AMOUNT_BIGGER_THAN_MAX_LOAN_SIZE_STABLE);
   }
  }
```

## Recommendation

Add an explicit check to ensure the borrow amount is greater than zero. This protects the function against redundant processing and future logical errors.

## 8. Lack of Proxy Mode Tracking in setAddress Enables Inconsistent Address Configuration

| Severity | Low |
|---|---|
| Finding ID | BH-protocol-v2-8 |
| Target | contracts/configuration/LendingPoolAddressesProvider.sol |
| Function name | setAddress |

### Description

The `setAddress` function does not track whether each `id` is associated with a proxied or non-proxied address. As a result, the contract allows an implicit and potentially unintended transition from a proxied address to a non-proxied address and vice versa. This may break logic or expectations in systems relying on upgradeable proxies. These inconsistencies can lead to deployment errors, lost upgrade paths, or unexpected runtime behavior.

```
function setAddress(
    bytes32 id,
    address newAddress,
    address implementationAddress
  ) external override onlyOwner {
    if (implementationAddress != address(0)) {
      _updateImpl(id, implementationAddress);
      emit AddressSet(id, implementationAddress, true);
    } else {
      _addresses[id] = newAddress;
      emit AddressSet(id, newAddress, false);
    }
  }
```

### Recommendation

Introduce explicit tracking of proxy usage per `id`, and make transitions between proxied and non-proxied modes deliberate and constrained. This can be achieved by maintaining a mapping that records the mode for each `id`, and enforcing rules around transitions.

## 9. Missing Validation Allows Registration of Lending Pool with Invalid ID

| Severity | Low |
|---|---|
| Finding ID | BH-protocol-v2-9 |
| Target | contracts/configuration/LendingPoolAddressesProviderRegistry.sol |
| Function name | registerAddressesProvider |

### Description

The `registerAddressesProvider` function does not validate that the `id` parameter is non-zero. In this system, an ID value of zero is reserved to indicate an inactive or absent lending pool. Allowing registration with an ID of 0 may result in ambiguities when querying or interpreting provider data, potentially leading to logic errors, misrouted interactions, or bypassed validations. This weakens the integrity of the lending pool registry and can introduce unexpected protocol behavior.

```
function registerAddressesProvider(address provider, uint256 id) external override
onlyOwner {
    _addressesProviders[provider] = id;
    _addToAddressesProvidersList(provider);
    emit AddressesProviderRegistered(provider);
  }
```

### Recommendation

Add a check to ensure that the `id` parameter is greater than zero before proceeding with registration. This will enforce the intended semantics of ID values and maintain clean separation between active and inactive lending pools.

## 10. Missing Active Reserve Check in validateWithdraw Allows Operations on Inactive Reserves

| Severity | Low |
|---|---|
| Finding ID | BH-protocol-v2-10 |
| Target | contracts/libraries/logic/ValidationLogic.sol |
| Function name | validateWithdraw |

### Description

The `validateWithdraw` function does not verify whether the specified reserve is active. Without this check, users can potentially perform withdrawal operation on inactive reserves. This behavior can lead to inconsistencies in reserve state management, allow unintended access to funds, or break protocol logic that assumes operations are only performed on active reserves. It also undermines risk controls that may rely on reserve deactivation as a safety mechanism.

```
function validateWithdraw(
    address reserveAddress,
    uint256 amount,
    uint256 userBalance,
    mapping(address => ReserveLogic.ReserveData) storage reservesData,
    UserConfiguration.Map storage userConfig,
    address[] calldata reserves,
    address oracle
  ) external view {
    require(amount > 0, Errors.AMOUNT_NOT_GREATER_THAN_0);

    require(amount <= userBalance, Errors.NOT_ENOUGH_AVAILABLE_USER_BALANCE);

    require(
      GenericLogic.balanceDecreaseAllowed(
        reserveAddress,
        msg.sender,
        userBalance,
        reservesData,
        userConfig,
        reserves,
        oracle
      ),
      Errors.TRANSFER_NOT_ALLOWED
    );
  }
```

### Recommendation

Include a validation step to ensure that the reserve is active before allowing any withdrawal operation.

## 11. Missing Active Reserve Check in rebalanceStableBorrowRate Allows Operation on Inactive Reserves

| Severity | Low |
|---|---|
| Finding ID | BH-protocol-v2-11 |
| Target | contracts/lendingpool/LendingPool.sol |
| Function name | rebalanceStableBorrowRate |

### Description

The `rebalanceStableBorrowRate` function does not validate whether the targeted reserve is active. This omission allows the function to proceed with operations such as burning and minting debt tokens and updating interest rates on reserves that may be administratively frozen or inactive. Operating on inactive reserves undermines protocol safety and may result in unintended or harmful behavior, such as modifying state in a reserve that is intended to be immutable due to emergency pauses or migration steps.

```
function rebalanceStableBorrowRate(address asset, address user) external override {

    _whenNotPaused();

    ReserveLogic.ReserveData storage reserve = _reserves[asset];

    IERC20 stableDebtToken = IERC20(reserve.stableDebtTokenAddress);
    IERC20 variableDebtToken = IERC20(reserve.variableDebtTokenAddress);
    address aTokenAddress = reserve.aTokenAddress;

    uint256 stableBorrowBalance = IERC20(stableDebtToken).balanceOf(user);

    //if the utilization rate is below 95%, no rebalances are needed
    uint256 totalBorrows =
stableDebtToken.totalSupply().add(variableDebtToken.totalSupply()).wadToRay();
    uint256 availableLiquidity = IERC20(asset).balanceOf(aTokenAddress).wadToRay();
    uint256 usageRatio = totalBorrows == 0
      ? 0
      : totalBorrows.rayDiv(availableLiquidity.add(totalBorrows));

    //if the liquidity rate is below REBALANCE_UP_THRESHOLD of the max variable APR at
95% usage,
    //then we allow rebalancing of the stable rate positions.

    uint256 currentLiquidityRate = reserve.currentLiquidityRate;
    uint256 maxVariableBorrowRate = IReserveInterestRateStrategy(
      reserve
        .interestRateStrategyAddress
    )
```

```
      .getMaxVariableBorrowRate();

    require(
      usageRatio >= REBALANCE_UP_USAGE_RATIO_THRESHOLD &&
      currentLiquidityRate <=
        maxVariableBorrowRate.percentMul(REBALANCE_UP_LIQUIDITY_RATE_THRESHOLD),
      Errors.INTEREST_RATE_REBALANCE_CONDITIONS_NOT_MET
    );

    reserve.updateState();

    IStableDebtToken(address(stableDebtToken)).burn(user, stableBorrowBalance);
    IStableDebtToken(address(stableDebtToken)).mint(user, stableBorrowBalance,
reserve.currentStableBorrowRate);

    reserve.updateInterestRates(asset, aTokenAddress, 0, 0);

    emit RebalanceStableBorrowRate(asset, user);

  }
```

## Recommendation

Add a validation step to ensure the reserve is active before allowing rebalancing of stable borrow rates.

## 12. Missing Active Reserve Check in flashLoan Enables Operations on Inactive Reserves

| Severity | Low |
|---|---|
| Finding ID | BH-protocol-v2-12 |
| Target | contracts/lendingpool/LendingPool.sol |
| Function name | flashLoan |

### Description

The `flashLoan` function does not verify whether the specified reserve is active before initiating a flash loan. As a result, the system allows flash loan operations on reserves that are administratively deactivated or frozen. This undermines reserve-level access control and opens the protocol to potential misuse, such as interacting with deprecated or under-maintenance assets. The absence of this validation compromises the intent behind reserve deactivation, which should block all core operations, including flash loans.

```
function flashLoan(
    address receiverAddress,
    address asset,
    uint256 amount,
    uint256 mode,
    bytes calldata params,
    uint16 referralCode
  ) external override {
    _whenNotPaused();
    ReserveLogic.ReserveData storage reserve = _reserves[asset];
    FlashLoanLocalVars memory vars;

    vars.aTokenAddress = reserve.aTokenAddress;

    vars.premium = amount.mul(FLASHLOAN_PREMIUM_TOTAL).div(10000);

    ValidationLogic.validateFlashloan(mode, vars.premium);

    ReserveLogic.InterestRateMode debtMode = ReserveLogic.InterestRateMode(mode);

    vars.receiver = IFlashLoanReceiver(receiverAddress);

    //transfer funds to the receiver
    IAToken(vars.aTokenAddress).transferUnderlyingTo(receiverAddress, amount);

    //execute action of the receiver
    vars.receiver.executeOperation(asset, amount, vars.premium, params);

    vars.amountPlusPremium = amount.add(vars.premium);
```

```
    if (debtMode == ReserveLogic.InterestRateMode.NONE) {
      IERC20(asset).safeTransferFrom(receiverAddress, vars.aTokenAddress,
vars.amountPlusPremium);

      reserve.updateState();
      reserve.cumulateToLiquidityIndex(IERC20(vars.aTokenAddress).totalSupply(),
vars.premium);
      reserve.updateInterestRates(asset, vars.aTokenAddress, vars.premium, 0);

      emit FlashLoan(receiverAddress, asset, amount, vars.premium, referralCode);
    } else {
     //if the user didn't choose to return the funds, the system checks if there
     //is enough collateral and eventually open a position
      _executeBorrow(
        ExecuteBorrowParams(
          asset,
          msg.sender,
          msg.sender,
          vars.amountPlusPremium,
          mode,
          vars.aTokenAddress,
          referralCode,
          false
        )
      );
    }
  }
```

## Recommendation

Introduce a validation step to ensure that the reserve is active before allowing a flash loan to proceed. This helps enforce safety mechanisms and aligns behavior with the expectations of reserve lifecycle management.

## 13. Collateral Presence Check Can Be Bypassed via aToken Transfers

| Severity | Low |
|---|---|
| Finding ID | BH-protocol-v2-13 |
| Target | contracts/libraries/logic/ValidationLogic.sol |
| Function name | validateBorrow |

### Description

The `require(vars.userCollateralBalanceETH > 0, Errors.COLLATERAL_BALANCE_IS_0);` line verifies that a user has deposited collateral. However, users can receive aTokens via transfers or protocol incentives, bypassing the original deposit validation. This can allow users to appear eligible for borrowing without going through required risk checks.

```
function validateBorrow(
    ReserveLogic.ReserveData storage reserve,
    address userAddress,
    uint256 amount,
    uint256 amountInETH,
    uint256 interestRateMode,
    uint256 maxStableLoanPercent,
    mapping(address => ReserveLogic.ReserveData) storage reservesData,
    UserConfiguration.Map storage userConfig,
    address[] calldata reserves,
    address oracle
  ) external view {
    ValidateBorrowLocalVars memory vars;

    (
      vars.isActive,
      vars.isFreezed,
      vars.borrowingEnabled,
      vars.stableRateBorrowingEnabled
    ) = reserve.configuration.getFlags();

    require(vars.isActive, Errors.NO_ACTIVE_RESERVE);
    require(!vars.isFreezed, Errors.NO_UNFREEZED_RESERVE);

    require(vars.borrowingEnabled, Errors.BORROWING_NOT_ENABLED);

    //validate interest rate mode
    require(
      uint256(ReserveLogic.InterestRateMode.VARIABLE) == interestRateMode ||
        uint256(ReserveLogic.InterestRateMode.STABLE) == interestRateMode,
      Errors.INVALID_INTEREST_RATE_MODE_SELECTED
    );
```

```
    (
      vars.userCollateralBalanceETH,
      vars.userBorrowBalanceETH,
      vars.currentLtv,
      vars.currentLiquidationThreshold,
      vars.healthFactor
    ) = GenericLogic.calculateUserAccountData(
      userAddress,
      reservesData,
      userConfig,
      reserves,
      oracle
    );

    require(vars.userCollateralBalanceETH > 0, Errors.COLLATERAL_BALANCE_IS_0);

    require(
      vars.healthFactor > GenericLogic.HEALTH_FACTOR_LIQUIDATION_THRESHOLD,
      Errors.HEALTH_FACTOR_LOWER_THAN_LIQUIDATION_THRESHOLD
    );

    //add the current already borrowed amount to the amount requested to calculate the
total collateral needed.
    vars.amountOfCollateralNeededETH =
vars.userBorrowBalanceETH.add(amountInETH).percentDiv(
      vars.currentLtv
    ); //LTV is calculated in percentage

    require(
      vars.amountOfCollateralNeededETH <= vars.userCollateralBalanceETH,
      Errors.COLLATERAL_CANNOT_COVER_NEW_BORROW
    );

    /**
     * Following conditions need to be met if the user is borrowing at a stable rate:
     * 1. Reserve must be enabled for stable rate borrowing
     * 2. Users cannot borrow from the reserve if their collateral is (mostly) the same currency
     *    they are borrowing, to prevent abuses.
     * 3. Users will be able to borrow only a relatively small, configurable amount of the total
     *    liquidity
     **/

    if (vars.rateMode == ReserveLogic.InterestRateMode.STABLE) {
      //check if the borrow mode is stable and if stable rate borrowing is enabled on this
reserve
```

```
    require(vars.stableRateBorrowingEnabled,
Errors.STABLE_BORROWING_NOT_ENABLED);

    require(
      !userConfig.isUsingAsCollateral(reserve.id) ||
        reserve.configuration.getLtv() == 0 ||
        amount > IERC20(reserve.aTokenAddress).balanceOf(userAddress),
      Errors.CALLATERAL_SAME_AS_BORROWING_CURRENCY
    );

    //calculate the max available loan size in stable rate mode as a percentage of the
    //available liquidity
    uint256 maxLoanSizeStable =
vars.availableLiquidity.percentMul(maxStableLoanPercent);

    require(amount <= maxLoanSizeStable,
Errors.AMOUNT_BIGGER_THAN_MAX_LOAN_SIZE_STABLE);
  }
 }
```

## Recommendation

Track the source of collateral (e.g., deposited vs transferred) or revalidate borrow eligibility when a user's aToken balance increases. Consider restricting aToken transfers or validating collateral status during `borrow()` execution.

## 14. Health Factor Check Can Be Circumvented via External Collateral Top-up",

| Severity | Low |
|---|---|
| Finding ID | BH-protocol-v2-14 |
| Target | contracts/libraries/logic/ValidationLogic.sol |
| Function name | validateBorrow |

### Description

The `require(vars.healthFactor > GenericLogic.HEALTH_FACTOR_LIQUIDATION_THRESHOLD, Errors.HEALTH_FACTOR_LOWER_THAN_LIQUIDATION_THRESHOLD);` check ensures user safety before borrowing. However, a user can raise their health factor by receiving aTokens via transfers without triggering this revalidation, potentially making previously ineligible borrowers eligible.

```
function validateBorrow(
    ReserveLogic.ReserveData storage reserve,
    address userAddress,
    uint256 amount,
    uint256 amountInETH,
    uint256 interestRateMode,
    uint256 maxStableLoanPercent,
    mapping(address => ReserveLogic.ReserveData) storage reservesData,
    UserConfiguration.Map storage userConfig,
    address[] calldata reserves,
    address oracle
  ) external view {
    ValidateBorrowLocalVars memory vars;

    (
      vars.isActive,
      vars.isFreezed,
      vars.borrowingEnabled,
      vars.stableRateBorrowingEnabled
    ) = reserve.configuration.getFlags();

    require(vars.isActive, Errors.NO_ACTIVE_RESERVE);
    require(!vars.isFreezed, Errors.NO_UNFREEZED_RESERVE);

    require(vars.borrowingEnabled, Errors.BORROWING_NOT_ENABLED);

    //validate interest rate mode
    require(
      uint256(ReserveLogic.InterestRateMode.VARIABLE) == interestRateMode ||
        uint256(ReserveLogic.InterestRateMode.STABLE) == interestRateMode,
```

```
      Errors.INVALID_INTEREST_RATE_MODE_SELECTED
    );

    (
      vars.userCollateralBalanceETH,
      vars.userBorrowBalanceETH,
      vars.currentLtv,
      vars.currentLiquidationThreshold,
      vars.healthFactor
    ) = GenericLogic.calculateUserAccountData(
      userAddress,
      reservesData,
      userConfig,
      reserves,
      oracle
    );

    require(vars.userCollateralBalanceETH > 0, Errors.COLLATERAL_BALANCE_IS_0);

    require(
      vars.healthFactor > GenericLogic.HEALTH_FACTOR_LIQUIDATION_THRESHOLD,
      Errors.HEALTH_FACTOR_LOWER_THAN_LIQUIDATION_THRESHOLD
    );

    //add the current already borrowed amount to the amount requested to calculate the
total collateral needed.
    vars.amountOfCollateralNeededETH =
vars.userBorrowBalanceETH.add(amountInETH).percentDiv(
      vars.currentLtv
    ); //LTV is calculated in percentage

    require(
      vars.amountOfCollateralNeededETH <= vars.userCollateralBalanceETH,
      Errors.COLLATERAL_CANNOT_COVER_NEW_BORROW
    );

    /**
     * Following conditions need to be met if the user is borrowing at a stable rate:
     * 1. Reserve must be enabled for stable rate borrowing
     * 2. Users cannot borrow from the reserve if their collateral is (mostly) the same currency
     *    they are borrowing, to prevent abuses.
     * 3. Users will be able to borrow only a relatively small, configurable amount of the total
     *    liquidity
     **/

    if (vars.rateMode == ReserveLogic.InterestRateMode.STABLE) {
      //check if the borrow mode is stable and if stable rate borrowing is enabled on this
reserve
```

```
    require(vars.stableRateBorrowingEnabled,
Errors.STABLE_BORROWING_NOT_ENABLED);

    require(
      !userConfig.isUsingAsCollateral(reserve.id) ||
        reserve.configuration.getLtv() == 0 ||
        amount > IERC20(reserve.aTokenAddress).balanceOf(userAddress),
      Errors.CALLATERAL_SAME_AS_BORROWING_CURRENCY
    );

    //calculate the max available loan size in stable rate mode as a percentage of the
    //available liquidity
    uint256 maxLoanSizeStable =
vars.availableLiquidity.percentMul(maxStableLoanPercent);

    require(amount <= maxLoanSizeStable,
Errors.AMOUNT_BIGGER_THAN_MAX_LOAN_SIZE_STABLE);
  }
 }
```

## Recommendation

Implement health factor revalidation upon key balance-affecting events (e.g., aToken transfers, incentive rewards) or restrict aToken transfers to trusted flows. This ensures borrow permission reflects actual risk posture.

## 15. Borrow Limit Check Based on LTV Can Be Bypassed via Unvalidated aToken Balance Increases

| Severity | Low |
|---|---|
| Finding ID | BH-protocol-v2-15 |
| Target | contracts/libraries/logic/ValidationLogic.sol |
| Function name | validateBorrow |

### Description

The calculation `amountOfCollateralNeededETH <= userCollateralBalanceETH` is used to ensure borrowing stays within the allowed LTV ratio. However, if a user's aToken balance is increased through transfers, rewards, or third-party actions, this check can be bypassed, enabling borrowing beyond safe limits.

```
function validateBorrow(
    ReserveLogic.ReserveData storage reserve,
    address userAddress,
    uint256 amount,
    uint256 amountInETH,
    uint256 interestRateMode,
    uint256 maxStableLoanPercent,
    mapping(address => ReserveLogic.ReserveData) storage reservesData,
    UserConfiguration.Map storage userConfig,
    address[] calldata reserves,
    address oracle
  ) external view {
    ValidateBorrowLocalVars memory vars;

    (
      vars.isActive,
      vars.isFreezed,
      vars.borrowingEnabled,
      vars.stableRateBorrowingEnabled
    ) = reserve.configuration.getFlags();

    require(vars.isActive, Errors.NO_ACTIVE_RESERVE);
    require(!vars.isFreezed, Errors.NO_UNFREEZED_RESERVE);

    require(vars.borrowingEnabled, Errors.BORROWING_NOT_ENABLED);

    //validate interest rate mode
    require(
      uint256(ReserveLogic.InterestRateMode.VARIABLE) == interestRateMode ||
        uint256(ReserveLogic.InterestRateMode.STABLE) == interestRateMode,
      Errors.INVALID_INTEREST_RATE_MODE_SELECTED
```

```
    );

    (
      vars.userCollateralBalanceETH,
      vars.userBorrowBalanceETH,
      vars.currentLtv,
      vars.currentLiquidationThreshold,
      vars.healthFactor
    ) = GenericLogic.calculateUserAccountData(
      userAddress,
      reservesData,
      userConfig,
      reserves,
      oracle
    );

    require(vars.userCollateralBalanceETH > 0, Errors.COLLATERAL_BALANCE_IS_0);

    require(
      vars.healthFactor > GenericLogic.HEALTH_FACTOR_LIQUIDATION_THRESHOLD,
      Errors.HEALTH_FACTOR_LOWER_THAN_LIQUIDATION_THRESHOLD
    );

    //add the current already borrowed amount to the amount requested to calculate the
total collateral needed.
    vars.amountOfCollateralNeededETH =
vars.userBorrowBalanceETH.add(amountInETH).percentDiv(
      vars.currentLtv
    ); //LTV is calculated in percentage

    require(
      vars.amountOfCollateralNeededETH <= vars.userCollateralBalanceETH,
      Errors.COLLATERAL_CANNOT_COVER_NEW_BORROW
    );

    /**
     * Following conditions need to be met if the user is borrowing at a stable rate:
     * 1. Reserve must be enabled for stable rate borrowing
     * 2. Users cannot borrow from the reserve if their collateral is (mostly) the same currency
     *    they are borrowing, to prevent abuses.
     * 3. Users will be able to borrow only a relatively small, configurable amount of the total
     *    liquidity
     **/

    if (vars.rateMode == ReserveLogic.InterestRateMode.STABLE) {
      //check if the borrow mode is stable and if stable rate borrowing is enabled on this
reserve
```

```
    require(vars.stableRateBorrowingEnabled,
Errors.STABLE_BORROWING_NOT_ENABLED);

    require(
      !userConfig.isUsingAsCollateral(reserve.id) ||
        reserve.configuration.getLtv() == 0 ||
        amount > IERC20(reserve.aTokenAddress).balanceOf(userAddress),
      Errors.CALLATERAL_SAME_AS_BORROWING_CURRENCY
    );

    //calculate the max available loan size in stable rate mode as a percentage of the
    //available liquidity
    uint256 maxLoanSizeStable =
vars.availableLiquidity.percentMul(maxStableLoanPercent);

    require(amount <= maxLoanSizeStable,
Errors.AMOUNT_BIGGER_THAN_MAX_LOAN_SIZE_STABLE);
  }
 }
```

## Recommendation

Ensure collateral increases are validated through protocol-controlled deposits. Consider splitting collateral balances into 'validated' and 'external' categories, or restrict aToken transfers and reward flows to trusted accounts only.

## 16. Stable Borrow Collateral Matching Check Can Be Bypassed with Token Transfers

| Severity | Low |
|---|---|
| Finding ID | BH-protocol-v2-16 |
| Target | contracts/libraries/logic/ValidationLogic.sol |
| Function name | validateBorrow |

### Description

The check preventing users from borrowing in the same asset as their collateral (`!userConfig.isUsingAsCollateral(...) || amount > balanceOf(user)`) is intended to prevent rate manipulation. However, users can receive aTokens from other accounts and bypass this logic, satisfying the balance condition without a real deposit.

```
function validateBorrow(
    ReserveLogic.ReserveData storage reserve,
    address userAddress,
    uint256 amount,
    uint256 amountInETH,
    uint256 interestRateMode,
    uint256 maxStableLoanPercent,
    mapping(address => ReserveLogic.ReserveData) storage reservesData,
    UserConfiguration.Map storage userConfig,
    address[] calldata reserves,
    address oracle
  ) external view {
    ValidateBorrowLocalVars memory vars;

    (
      vars.isActive,
      vars.isFreezed,
      vars.borrowingEnabled,
      vars.stableRateBorrowingEnabled
    ) = reserve.configuration.getFlags();

    require(vars.isActive, Errors.NO_ACTIVE_RESERVE);
    require(!vars.isFreezed, Errors.NO_UNFREEZED_RESERVE);

    require(vars.borrowingEnabled, Errors.BORROWING_NOT_ENABLED);

    //validate interest rate mode
    require(
      uint256(ReserveLogic.InterestRateMode.VARIABLE) == interestRateMode ||
        uint256(ReserveLogic.InterestRateMode.STABLE) == interestRateMode,
      Errors.INVALID_INTEREST_RATE_MODE_SELECTED
```

```
    );

    (
      vars.userCollateralBalanceETH,
      vars.userBorrowBalanceETH,
      vars.currentLtv,
      vars.currentLiquidationThreshold,
      vars.healthFactor
    ) = GenericLogic.calculateUserAccountData(
      userAddress,
      reservesData,
      userConfig,
      reserves,
      oracle
    );

    require(vars.userCollateralBalanceETH > 0, Errors.COLLATERAL_BALANCE_IS_0);

    require(
      vars.healthFactor > GenericLogic.HEALTH_FACTOR_LIQUIDATION_THRESHOLD,
      Errors.HEALTH_FACTOR_LOWER_THAN_LIQUIDATION_THRESHOLD
    );

    //add the current already borrowed amount to the amount requested to calculate the
total collateral needed.
    vars.amountOfCollateralNeededETH =
vars.userBorrowBalanceETH.add(amountInETH).percentDiv(
      vars.currentLtv
    ); //LTV is calculated in percentage

    require(
      vars.amountOfCollateralNeededETH <= vars.userCollateralBalanceETH,
      Errors.COLLATERAL_CANNOT_COVER_NEW_BORROW
    );

    /**
     * Following conditions need to be met if the user is borrowing at a stable rate:
     * 1. Reserve must be enabled for stable rate borrowing
     * 2. Users cannot borrow from the reserve if their collateral is (mostly) the same currency
     *    they are borrowing, to prevent abuses.
     * 3. Users will be able to borrow only a relatively small, configurable amount of the total
     *    liquidity
     **/

    if (vars.rateMode == ReserveLogic.InterestRateMode.STABLE) {
      //check if the borrow mode is stable and if stable rate borrowing is enabled on this
reserve
```

```
    require(vars.stableRateBorrowingEnabled,
Errors.STABLE_BORROWING_NOT_ENABLED);

    require(
      !userConfig.isUsingAsCollateral(reserve.id) ||
        reserve.configuration.getLtv() == 0 ||
        amount > IERC20(reserve.aTokenAddress).balanceOf(userAddress),
      Errors.CALLATERAL_SAME_AS_BORROWING_CURRENCY
    );

    //calculate the max available loan size in stable rate mode as a percentage of the
    //available liquidity
    uint256 maxLoanSizeStable =
vars.availableLiquidity.percentMul(maxStableLoanPercent);

    require(amount <= maxLoanSizeStable,
Errors.AMOUNT_BIGGER_THAN_MAX_LOAN_SIZE_STABLE);
  }
 }
```

## Recommendation

Track how collateral was obtained (deposited vs transferred), and enforce stricter logic around stable borrowing. Restrict aToken transfers or include validation of deposit origin before allowing stable borrow flows.

## 17. Collateral Abuse Check in validateSwapRateMode Can Be Bypassed via aToken Transfers

| Severity | Low |
|---|---|
| Finding ID | BH-protocol-v2-17 |
| Target | contracts/libraries/logic/ValidationLogic.sol |
| Function name | validateSwapRateMode |

### Description

The `validateSwapRateMode` function includes a check designed to prevent users from manipulating interest rates by borrowing under a variable rate, supplying excessive collateral, and then switching to a stable rate. The check relies on comparing the user's total debt to their aToken balance: `stableBorrowBalance + variableBorrowBalance > aToken.balanceOf(msg.sender)`. However, users can receive aTokens via transfers or rewards from other sources, inflating their balance without genuine collateral deposit. This bypasses the intent of the check and allows interest rate manipulation through external aToken top-ups.

```
function validateSwapRateMode(
    ReserveLogic.ReserveData storage reserve,
    UserConfiguration.Map storage userConfig,
    uint256 stableBorrowBalance,
    uint256 variableBorrowBalance,
    ReserveLogic.InterestRateMode currentRateMode
  ) external view {
    (bool isActive, bool isFreezed, , bool stableRateEnabled) =
reserve.configuration.getFlags();

    require(isActive, Errors.NO_ACTIVE_RESERVE);
    require(!isFreezed, Errors.NO_UNFREEZED_RESERVE);

    if (currentRateMode == ReserveLogic.InterestRateMode.STABLE) {
      require(stableBorrowBalance > 0, Errors.NO_STABLE_RATE_LOAN_IN_RESERVE);
    } else if (currentRateMode == ReserveLogic.InterestRateMode.VARIABLE) {
      require(variableBorrowBalance > 0, Errors.NO_VARIABLE_RATE_LOAN_IN_RESERVE);
      /**
       * user wants to swap to stable, before swapping we need to ensure that
       * 1. stable borrow rate is enabled on the reserve
       * 2. user is not trying to abuse the reserve by depositing
       * more collateral than he is borrowing, artificially lowering
       * the interest rate, borrowing at variable, and switching to stable
       **/
      require(stableRateEnabled, Errors.STABLE_BORROWING_NOT_ENABLED);

      require(
```

```
      !userConfig.isUsingAsCollateral(reserve.id) ||
        reserve.configuration.getLtv() == 0 ||
        stableBorrowBalance.add(variableBorrowBalance) >
        IERC20(reserve.aTokenAddress).balanceOf(msg.sender),
      Errors.CALLATERAL_SAME_AS_BORROWING_CURRENCY
    );
  } else {
    revert(Errors.INVALID_INTEREST_RATE_MODE_SELECTED);
  }
}
```

## Recommendation

Restrict or track the source of aToken balances used for collateral so that only explicitly deposited amounts are considered in this check. Alternatively, maintain a separate record of 'validated collateral' that excludes externally transferred aTokens. If this is not feasible, re-evaluate the logic or add conservative thresholds to reduce manipulation surface.

## 18. Missing Minimum Holding Duration Enables Collateral Manipulation

| Severity | Medium |
|---|---|
| Finding ID | BH-protocol-v2-18 |
| Target | contracts/libraries/logic/ GenericLogic.sol |
| Function name | balanceDecreaseAllowed |

### Description

The `balanceDecreaseAllowed` function checks if a user's collateral can be decreased without compromising the health factor but lacks any enforcement of a minimum holding duration between deposit and withdrawal. This allows users to deposit collateral and withdraw it in the next block, potentially manipulating the oracle price or health factor to game the system. This defeats the intended protective mechanism and exposes the protocol to price manipulation and under-collateralized borrowing attacks.

```
function balanceDecreaseAllowed(
    address asset,
    address user,
    uint256 amount,
    mapping(address => ReserveLogic.ReserveData) storage reservesData,
    UserConfiguration.Map calldata userConfig,
    address[] calldata reserves,
    address oracle
  ) external view returns (bool) {
    if (
      !userConfig.isBorrowingAny() ||
      !userConfig.isUsingAsCollateral(reservesData[asset].id)
    ) {
      return true;
    }

    balanceDecreaseAllowedLocalVars memory vars;

    (vars.ltv, , , vars.decimals) = reservesData[asset].configuration.getParams();

    if (vars.ltv == 0) {
      return true; //if reserve is not used as collateral, no reasons to block the transfer
    }

    (
      vars.collateralBalanceETH,
      vars.borrowBalanceETH,
      ,
      vars.currentLiquidationThreshold,
```

```
    ) = calculateUserAccountData(user, reservesData, userConfig, reserves, oracle);

    if (vars.borrowBalanceETH == 0) {
      return true; //no borrows - no reasons to block the transfer
    }

    vars.amountToDecreaseETH =
IPriceOracleGetter(oracle).getAssetPrice(asset).mul(amount).div(
      10**vars.decimals
    );

    vars.collateralBalancefterDecrease =
vars.collateralBalanceETH.sub(vars.amountToDecreaseETH);

    //if there is a borrow, there can't be 0 collateral
    if (vars.collateralBalancefterDecrease == 0) {
      return false;
    }

    vars.liquidationThresholdAfterDecrease = vars
      .collateralBalanceETH
      .mul(vars.currentLiquidationThreshold)
      .sub(vars.amountToDecreaseETH.mul(vars.reserveLiquidationThreshold))
      .div(vars.collateralBalancefterDecrease);

    uint256 healthFactorAfterDecrease = calculateHealthFactorFromBalances(
      vars.collateralBalancefterDecrease,
      vars.borrowBalanceETH,
      vars.liquidationThresholdAfterDecrease
    );

    return healthFactorAfterDecrease >
GenericLogic.HEALTH_FACTOR_LIQUIDATION_THRESHOLD;
  }
```

## Recommendation

Introduce a minimum holding duration by tracking the timestamp of the last deposit per user and asset. Reject withdrawals or collateral decreases if the holding period has not elapsed. Below is a suggested patch.

## 19. Unoptimized Loop in _addToAddressesProvidersList Causes Excessive SLOAD Operations

| Severity | Low |
|---|---|
| Finding ID | BH-protocol-v2-19 |
| Target | contracts/configuration/LendingPoolAddressesProviderRegistry.sol |
| Function name | _addToAddressesProvidersList |

### Description

The `_addToAddressesProvidersList` function performs a linear search to check if the `provider` already exists in the `addressesProvidersList` array. However, `addressesProvidersList.length` is accessed repeatedly in the loop condition, which incurs multiple storage reads, since it's a storage variable. In tight loops over storage arrays, this results in higher gas costs, especially as the list grows.

```
function _addToAddressesProvidersList(address provider) internal {
    for (uint256 i = 0; i < addressesProvidersList.length; i++) {
      if (addressesProvidersList[i] == provider) {
        return;
      }
    }

    addressesProvidersList.push(provider);
  }
```

### Recommendation

Cache the length of the `addressesProvidersList` array in a local memory variable before the loop. This reduces redundant storage access and improves gas efficiency.

## 20. Unoptimized reserveAlreadyAdded Flag Calculation in _addReserveToList

| Severity | Low |
|---|---|
| Finding ID | BH-protocol-v2-20 |
| Target | contracts/lendingpool/LendingPool.sol |
| Function name | _ addReserveToList |

### Description

The `_addReserveToList` function performs a linear search through `_reservesList` to determine if an asset is already added, using a `reserveAlreadyAdded` boolean flag. However, the current implementation continues iterating even after the asset is found, which is inefficient. It wastes gas by scanning the rest of the list unnecessarily after a match is detected.

```
function _addReserveToList(address asset) internal {
    bool reserveAlreadyAdded = false;
    require(_reservesList.length < MAX_NUMBER_RESERVES,
Errors.NO_MORE_RESERVES_ALLOWED);
    for (uint256 i = 0; i < _reservesList.length; i++)
      if (_reservesList[i] == asset) {
        reserveAlreadyAdded = true;
      }
    if (!reserveAlreadyAdded) {
      _reserves[asset].id = uint8(_reservesList.length);
      _reservesList.push(asset);
    }
  }
```

### Recommendation

Return early as soon as the asset is found in `_reservesList` to save gas. This avoids unnecessary loop iterations and makes the logic cleaner.

## 21. Incorrect Constraint on Liquidation Threshold and Bonus in configureReserveAsCollateral

| Severity | Low |
|---|---|
| Finding ID | BH-protocol-v2-21 |
| Target | contracts/lendingpool/LendingPoolConfigurator.sol |
| Function name | _configureReserveAsCollateral |

### Description

The `configureReserveAsCollateral` function includes a flawed validation constraint: `require(liquidationThreshold + absoluteBonus <= 100%)`. This logic incorrectly assumes that liquidationThreshold and liquidationBonus are linearly related and additive. The current check unnecessarily restricts valid and safe configurations, potentially harming protocol flexibility and risk tuning.

```
function configureReserveAsCollateral(
    address asset,
    uint256 ltv,
    uint256 liquidationThreshold,
    uint256 liquidationBonus
  ) external onlyPoolAdmin {
    ReserveConfiguration.Map memory currentConfig = pool.getConfiguration(asset);

    //validation of the parameters: the LTV can
    //only be lower or equal than the liquidation threshold
    //(otherwise a loan against the asset would cause instantaneous liquidation)
    require(ltv <= liquidationThreshold, Errors.LPC_INVALID_CONFIGURATION);

    if (liquidationThreshold != 0) {
      //liquidation bonus must be bigger than 100.00%, otherwise the liquidator would
receive less
      //collateral than needed to cover the debt.
      uint256 absoluteBonus =
liquidationBonus.sub(PercentageMath.PERCENTAGE_FACTOR,
Errors.LPC_INVALID_CONFIGURATION);
      require(absoluteBonus > 0, Errors.LPC_INVALID_CONFIGURATION);

      //we also need to require that the liq threshold is lower or equal than the liquidation
bonus, to ensure that
      //there is always enough margin for liquidators to receive the bonus.
      require(liquidationThreshold.add(absoluteBonus) <=
PercentageMath.PERCENTAGE_FACTOR, Errors.LPC_INVALID_CONFIGURATION);

    } else {
      require(liquidationBonus == 0, Errors.LPC_INVALID_CONFIGURATION);
```

```
    //if the liquidation threshold is being set to 0,
    // the reserve is being disabled as collateral. To do so,
    //we need to ensure no liquidity is deposited
    _checkNoLiquidity(asset);
  }

  currentConfig.setLtv(ltv);
  currentConfig.setLiquidationThreshold(liquidationThreshold);
  currentConfig.setLiquidationBonus(liquidationBonus);

  pool.setConfiguration(asset, currentConfig.data);

  emit CollateralConfigurationChanged(asset, ltv, liquidationThreshold, liquidationBonus);
 }
```

## Recommendation

Remove or revise the constraint to reflect the inverse relationship between `liquidationThreshold` and `liquidationBonus`. Instead of enforcing a summed cap, simulate a worst-case liquidation scenario (e.g., a user just at threshold) to ensure the seized collateral covers the debt plus bonus. Alternatively, introduce dynamic limits based on reserve-specific risk models.

## 22. Incorrect Constraint on Liquidation Threshold and Bonus in configureReserveAsCollateral

| Severity | Low |
|---|---|
| Finding ID | BH-protocol-v2-22 |
| Target | contracts/lendingpool/LendingPoolConfigurator.sol |
| Function name | _configureReserveAsCollateral |

### Description

The `configureReserveAsCollateral` function includes a flawed validation constraint: `require(liquidationThreshold + absoluteBonus <= 100%)`. This logic incorrectly assumes that liquidationThreshold and liquidationBonus are linearly related and additive. The current check unnecessarily restricts valid and safe configurations, potentially harming protocol flexibility and risk tuning.

```
function configureReserveAsCollateral(
    address asset,
    uint256 ltv,
    uint256 liquidationThreshold,
    uint256 liquidationBonus
  ) external onlyPoolAdmin {
    ReserveConfiguration.Map memory currentConfig = pool.getConfiguration(asset);

    //validation of the parameters: the LTV can
    //only be lower or equal than the liquidation threshold
    //(otherwise a loan against the asset would cause instantaneous liquidation)
    require(ltv <= liquidationThreshold, Errors.LPC_INVALID_CONFIGURATION);

    if (liquidationThreshold != 0) {
      //liquidation bonus must be bigger than 100.00%, otherwise the liquidator would
receive less
      //collateral than needed to cover the debt.
      uint256 absoluteBonus =
liquidationBonus.sub(PercentageMath.PERCENTAGE_FACTOR,
Errors.LPC_INVALID_CONFIGURATION);
      require(absoluteBonus > 0, Errors.LPC_INVALID_CONFIGURATION);

      //we also need to require that the liq threshold is lower or equal than the liquidation
bonus, to ensure that
      //there is always enough margin for liquidators to receive the bonus.
      require(liquidationThreshold.add(absoluteBonus) <=
PercentageMath.PERCENTAGE_FACTOR, Errors.LPC_INVALID_CONFIGURATION);

    } else {
      require(liquidationBonus == 0, Errors.LPC_INVALID_CONFIGURATION);
```

```
    //if the liquidation threshold is being set to 0,
    // the reserve is being disabled as collateral. To do so,
    //we need to ensure no liquidity is deposited
    _checkNoLiquidity(asset);
  }

  currentConfig.setLtv(ltv);
  currentConfig.setLiquidationThreshold(liquidationThreshold);
  currentConfig.setLiquidationBonus(liquidationBonus);

  pool.setConfiguration(asset, currentConfig.data);

  emit CollateralConfigurationChanged(asset, ltv, liquidationThreshold, liquidationBonus);
 }
```

## Recommendation

Remove or revise the constraint to reflect the inverse relationship between `liquidationThreshold` and `liquidationBonus`. Instead of enforcing a summed cap, simulate a worst-case liquidation scenario (e.g., a user just at threshold) to ensure the seized collateral covers the debt plus bonus. Alternatively, introduce dynamic limits based on reserve-specific risk models.

## 23. Potential Reentrancy in liquidationCall Due to Unsafe External Calls

| Severity | Low |
|---|---|
| Finding ID | BH-protocol-v2-23 |
| Target | contracts/protocol/lendingpool/LendingPoolCollateralManager.sol |
| Function name | _liquidationCall |

### Description

The `liquidationCall` function performs multiple external calls to user-controlled contracts or untrusted tokens (e.g., aToken, debtToken, and ERC20s) before finalizing all internal state updates. Specifically, calls like `safeTransferFrom`, `transferOnLiquidation`, and `burn` may trigger fallback or hook logic in malicious token contracts. Without a reentrancy guard, this can allow attackers to exploit intermediate protocol states or perform unauthorized recursive operations.

```
function liquidationCall(
    address collateralAsset,
    address debtAsset,
    address user,
    uint256 debtToCover,
    bool receiveAToken
  ) external override returns (uint256, string memory) {
    DataTypes.ReserveData storage collateralReserve = _reserves[collateralAsset];
    DataTypes.ReserveData storage debtReserve = _reserves[debtAsset];
    DataTypes.UserConfigurationMap storage userConfig = _usersConfig[user];

    LiquidationCallLocalVars memory vars;

    (, , , , vars.healthFactor) = GenericLogic.calculateUserAccountData(
      user,
      _reserves,
      userConfig,
      _reservesList,
      _reservesCount,
      _addressesProvider.getPriceOracle()
    );

    (vars.userStableDebt, vars.userVariableDebt) = Helpers.getUserCurrentDebt(user,
debtReserve);

    (vars.errorCode, vars.errorMsg) = ValidationLogic.validateLiquidationCall(
      collateralReserve,
      debtReserve,
      userConfig,
      vars.healthFactor,
```

```
      vars.userStableDebt,
      vars.userVariableDebt
    );

    if (Errors.CollateralManagerErrors(vars.errorCode) !=
Errors.CollateralManagerErrors.NO_ERROR) {
      return (vars.errorCode, vars.errorMsg);
    }

    vars.collateralAtoken = IAToken(collateralReserve.aTokenAddress);

    vars.userCollateralBalance = vars.collateralAtoken.balanceOf(user);

    vars.maxLiquidatableDebt = vars.userStableDebt.add(vars.userVariableDebt).percentMul(
      LIQUIDATION_CLOSE_FACTOR_PERCENT
    );

    vars.actualDebtToLiquidate = debtToCover > vars.maxLiquidatableDebt
      ? vars.maxLiquidatableDebt
      : debtToCover;

    (
      vars.maxCollateralToLiquidate,
      vars.debtAmountNeeded
    ) = _calculateAvailableCollateralToLiquidate(
      collateralReserve,
      debtReserve,
      collateralAsset,
      debtAsset,
      vars.actualDebtToLiquidate,
      vars.userCollateralBalance
    );

    // If debtAmountNeeded < actualDebtToLiquidate, there isn't enough
    // collateral to cover the actual amount that is being liquidated, hence we liquidate
    // a smaller amount

    if (vars.debtAmountNeeded < vars.actualDebtToLiquidate) {
      vars.actualDebtToLiquidate = vars.debtAmountNeeded;
    }

    // If the liquidator reclaims the underlying asset, we make sure there is enough available
liquidity in the
    // collateral reserve
    if (!receiveAToken) {
      uint256 currentAvailableCollateral =
        IERC20(collateralAsset).balanceOf(address(vars.collateralAtoken));
      if (currentAvailableCollateral < vars.maxCollateralToLiquidate) {
```

```
      return (
        uint256(Errors.CollateralManagerErrors.NOT_ENOUGH_LIQUIDITY),
        Errors.LPCM_NOT_ENOUGH_LIQUIDITY_TO_LIQUIDATE
      );
    }
  }

  debtReserve.updateState();

  if (vars.userVariableDebt >= vars.actualDebtToLiquidate) {
    IVariableDebtToken(debtReserve.variableDebtTokenAddress).burn(
      user,
      vars.actualDebtToLiquidate,
      debtReserve.variableBorrowIndex
    );
  } else {
    // If the user doesn't have variable debt, no need to try to burn variable debt tokens
    if (vars.userVariableDebt > 0) {
      IVariableDebtToken(debtReserve.variableDebtTokenAddress).burn(
        user,
        vars.userVariableDebt,
        debtReserve.variableBorrowIndex
      );
    }
    IStableDebtToken(debtReserve.stableDebtTokenAddress).burn(
      user,
      vars.actualDebtToLiquidate.sub(vars.userVariableDebt)
    );
  }

  debtReserve.updateInterestRates(
    debtAsset,
    debtReserve.aTokenAddress,
    vars.actualDebtToLiquidate,
    0
  );

  if (receiveAToken) {
    vars.liquidatorPreviousATokenBalance =
IERC20(vars.collateralAtoken).balanceOf(msg.sender);
    vars.collateralAtoken.transferOnLiquidation(user, msg.sender,
vars.maxCollateralToLiquidate);

    if (vars.liquidatorPreviousATokenBalance == 0) {
      DataTypes.UserConfigurationMap storage liquidatorConfig =
_usersConfig[msg.sender];
      liquidatorConfig.setUsingAsCollateral(collateralReserve.id, true);
      emit ReserveUsedAsCollateralEnabled(collateralAsset, msg.sender);
```

```
    }
  } else {
    collateralReserve.updateState();
    collateralReserve.updateInterestRates(
      collateralAsset,
      address(vars.collateralAtoken),
      0,
      vars.maxCollateralToLiquidate
    );

    // Burn the equivalent amount of aToken, sending the underlying to the liquidator
    vars.collateralAtoken.burn(
      user,
      msg.sender,
      vars.maxCollateralToLiquidate,
      collateralReserve.liquidityIndex
    );
  }

  // If the collateral being liquidated is equal to the user balance,
  // we set the currency as not being used as collateral anymore
  if (vars.maxCollateralToLiquidate == vars.userCollateralBalance) {
    userConfig.setUsingAsCollateral(collateralReserve.id, false);
    emit ReserveUsedAsCollateralDisabled(collateralAsset, user);
  }

  // Transfers the debt asset being repaid to the aToken, where the liquidity is kept
  IERC20(debtAsset).safeTransferFrom(
    msg.sender,
    debtReserve.aTokenAddress,
    vars.actualDebtToLiquidate
  );

  emit LiquidationCall(
    collateralAsset,
    debtAsset,
    user,
    vars.actualDebtToLiquidate,
    vars.maxCollateralToLiquidate,
    msg.sender,
    receiveAToken
  );

  return (uint256(Errors.CollateralManagerErrors.NO_ERROR), Errors.LPCM_NO_ERRORS);
}
```

## Recommendation

Protect the `liquidationCall` function using a `nonReentrant` modifier (from OpenZeppelin's ReentrancyGuard). Alternatively, rigorously enforce checks-effects-interactions order, but this is more error-prone.

## 24. Unchecked Return Value from ERC20 Transfer in transferOnLiquidation

| Severity | Low |
|---|---|
| Finding ID | BH-protocol-v2-24 |
| Target | contracts/tokenization/AToken.sol |
| Function name | _ transferOnLiquidation |

### Description

The `transferOnLiquidation` function calls `_transfer(from, to, value, false)` without checking for the success of the token transfer. If `_transfer` internally invokes `ERC20.transfer()` or `transferFrom()` on a non-standard token (e.g., USDT) that does not revert on failure but returns `false`, and this return value is not handled, the contract may incorrectly assume the transfer succeeded. This can result in lost funds or protocol state inconsistency during liquidations

```
function transferOnLiquidation(
    address from,
    address to,
    uint256 value
  ) external override onlyLendingPool {
    //being a normal transfer, the Transfer() and BalanceTransfer() are emitted
    //so no need to emit a specific event here
    _transfer(from, to, value, false);
  }
```

### Recommendation

Ensure that all internal transfer functions either revert on failure or explicitly check the return value of token transfer operations. If `_transfer` wraps external calls, use OpenZeppelin's `SafeERC20.safeTransfer` to enforce success, or add an explicit `require(success, \"Transfer failed\")` statement.

## 25. Potential Reentrancy in withdraw Due to External Call to aToken.burn() Before State Update

| Severity | Low |
|---|---|
| Finding ID | BH-protocol-v2-25 |
| Target | contracts/lendingpool/LendingPool.sol |
| Function name | withdraw |

### Description

The `withdraw` function calls an external contract's `burn()` method on the aToken contract before completing internal state changes and emitting the `Withdraw` event. If the aToken contract is compromised or allows callback execution (e.g., via ERC777-style hooks), this opens the protocol to reentrancy attacks. A malicious actor could re-enter the withdraw function and manipulate balances or configurations before state updates complete, leading to double-withdrawals or protocol inconsistency.

```
function withdraw(address asset, uint256 amount) external override {
  _whenNotPaused();
  ReserveLogic.ReserveData storage reserve = _reserves[asset];

  address aToken = reserve.aTokenAddress;

  uint256 userBalance = IAToken(aToken).balanceOf(msg.sender);

  uint256 amountToWithdraw = amount;

  //if amount is equal to uint(-1), the user wants to redeem everything
  if (amount == type(uint256).max) {
    amountToWithdraw = userBalance;
  }

  ValidationLogic.validateWithdraw(
    asset,
    amountToWithdraw,
    userBalance,
    _reserves,
    _usersConfig[msg.sender],
    _reservesList,
    _addressesProvider.getPriceOracle()
  );

  reserve.updateState();

  reserve.updateInterestRates(asset, aToken, 0, amountToWithdraw);
```

```
    if (amountToWithdraw == userBalance) {
      _usersConfig[msg.sender].setUsingAsCollateral(reserve.id, false);
    }

    IAToken(aToken).burn(msg.sender, msg.sender, amountToWithdraw,
reserve.liquidityIndex);

    emit Withdraw(asset, msg.sender, amount);
  }
```

## Recommendation

Apply a reentrancy guard to the function (`nonReentrant`) or move all internal state changes and event emissions to occur *before* any external calls.