Armors Labs

Vinci Protocol

Smart Contract Audit

Vinci Protocol Audit Summary

Project name : Vinci Protocol Contract

Project address: None

Code URL : https://github.com/VinciProtocol/vinci-protocol

Commit : 185cf3b9310d5324de8107689b083c550c175a25

Project target : Vinci Protocol Contract Audit

Blockchain : Ethereum

Test result : PASSED

Audit Info

Audit NO : 0X202204010026

Audit Team : Armors Labs

Audit Proofreading: https://armors.io/#project-cases

Vinci Protocol Audit

The Vinci Protocol team asked us to review and audit their Vinci Protocol contract. We looked at the code and now publish our results.

Here is our assessment and recommendations, in order of importance.

Document information

Name	Auditor	Version	Date
Vinci Protocol Audit	Rock, Sophia, Rushairer, Rico, David, Alice	1.0.0	2022-04-01

Audit results

Note that as of the date of publishing, the above review reflects the current understanding of known security patterns as they relate to the Vinci Protocol contract. The above should not be construed as investment advice.

Based on the widely recognized security status of the current underlying blockchain and smart contract, this audit report is valid for 3 months from the date of output.

Disclaimer

Armors Labs Reports is not and should not be regarded as an "approval" or "disapproval" of any particular project or team. These reports are not and should not be regarded as indicators of the economy or value of any "product" or "asset" created by any team. Armors do not cover testing or auditing the integration with external contract or services (such as Unicrypt, Uniswap, PancakeSwap etc'…)

Audited target file

file	md5
GenericLogic.sol	0c6166db02e11365154ef35a83a5f765
LendingPool.sol	15fc192a4a4776744adc13a330f65631
UserConfiguration.sol	6880974748b4bb09ae62b855f742f4dd
LendingPoolCollateralManager.sol	7e609dd3b521cf15f79b62ab22f7adc1
NToken.sol	6f223702cf36656ef8e449cfb98d3599
LendingPoolStorage.sol	f5cfd07f7c989b3b4756c5a4d7d0dbfe
NFTVaultLogic.sol	afe44266c3fac4f8f8486dc935976a2a
LendingPoolConfigurator.sol	6b9b14b6ee42a41c28bfa68f7c53f5f1
WrappedERC1155.sol	c433819e4c59eb3f8ea0e49dd922fb90
ValidationLogic.sol	0673d86bfd8bc216a0d19e90c5d3630b
NFTVaultConfiguration.sol	c8fb0d08ac2f4fab139d8f50dbdd6464

Vulnerability analysis

Vulnerability distribution

vulnerability level	number
Critical severity	0
High severity	0
Medium severity	0
Low severity	0

Summary of audit results

Vulnerability	status
Re-Entrancy	safe

Vulnerability	status
Arithmetic Over/Under Flows	safe
Unexpected Blockchain Currency	safe
Delegatecall	safe
Default Visibilities	safe
Entropy Illusion	safe
External Contract Referencing	safe
Short Address/Parameter Attack	safe
Unchecked CALL Return Values	safe
Race Conditions / Front Running	safe
Denial Of Service (DOS)	safe
Block Timestamp Manipulation	safe
Constructors with Care	safe
Unintialised Storage Pointers	safe
Floating Points and Numerical Precision	safe
tx.origin Authentication	safe
Permission restrictions	safe

Contract file

GenericLogic.sol

```
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.8.11;
pragma abicoder v2;

import {IERC20} from '../../../dependencies/openzeppelin/contracts/IERC20.sol';
import {IERC1155Stat} from '../../../interfaces/IERC1155Stat.sol';
import {ReserveLogic} from './ReserveLogic.sol';
import {ReserveConfiguration} from '../configuration/ReserveConfiguration.sol';
import {NFTVaultLogic} from './NFTVaultLogic.sol';
import {NFTVaultConfiguration} from '../configuration/NFTVaultConfiguration.sol';
import {UserConfiguration} from '../configuration/UserConfiguration.sol';
import {WadRayMath} from '../math/WadRayMath.sol';
import {PercentageMath} from '../math/PercentageMath.sol';
import {IPriceOracleGetter} from '../../../interfaces/IPriceOracleGetter.sol';
import {DataTypes} from '../types/DataTypes.sol';

/**
 * @title GenericLogic library
 * @author Aave
 * @title Implements protocol-level logic to calculate and validate the state of a user
 */
library GenericLogic {
  using ReserveLogic for DataTypes.ReserveData;
  using NFTVaultLogic for DataTypes.NFTVaultData;
  using WadRayMath for uint256;
  using PercentageMath for uint256;
  using ReserveConfiguration for DataTypes.ReserveConfigurationMap;
  using NFTVaultConfiguration for DataTypes.NFTVaultConfigurationMap;
  using UserConfiguration for DataTypes.UserConfigurationMap;

  uint256 public constant HEALTH_FACTOR_LIQUIDATION_THRESHOLD = 1 ether;

  struct balanceDecreaseAllowedLocalVars {
    uint256 liquidationThreshold;
    uint256 totalCollateralInETH;
    uint256 totalDebtInETH;
    uint256 avgLiquidationThreshold;
    uint256 amountToDecreaseInETH;
    uint256 collateralBalanceAfterDecrease;
    uint256 liquidationThresholdAfterDecrease;
    uint256 healthFactorAfterDecrease;
    bool reserveUsageAsCollateralEnabled;
  }

  /**
   * @dev Checks if a specific balance decrease is allowed
   * (i.e. doesn't bring the user borrow position health factor under HEALTH_FACTOR_LIQUIDATION_THRES
   * @param asset The address of the underlying asset of the reserve
   * @param user The address of the user
   * @param amount The amount to decrease
   * @param reserves The data of all the reserves
   * @param nftVaults The data of all the vaults
   * @param userConfig The user configuration
   * @param oracle The address of the oracle contract
   * @return true if the decrease of the balance is allowed
   **/
  function balanceDecreaseAllowed(
    address asset,
    address user,
    uint256 amount,
    DataTypes.PoolReservesData storage reserves,
    DataTypes.PoolNFTVaultsData storage nftVaults,
    DataTypes.UserConfigurationMap calldata userConfig,
    address oracle
  ) external view returns (bool) {
    if (!userConfig.isBorrowingAny()) {
      return true;
    }

    balanceDecreaseAllowedLocalVars memory vars;

    (, vars.liquidationThreshold, ) = nftVaults.data[asset]
      .configuration
      .getParams();

    if (vars.liquidationThreshold == 0) {
      return true;
    }

    (
      vars.totalCollateralInETH,
      vars.totalDebtInETH,
      ,
      vars.avgLiquidationThreshold,

    ) = calculateUserAccountData(user, reserves, nftVaults, userConfig, oracle);

    if (vars.totalDebtInETH == 0) {
      return true;
    }
    vars.amountToDecreaseInETH = IPriceOracleGetter(oracle).getAssetPrice(asset) * amount;

    vars.collateralBalanceAfterDecrease = vars.totalCollateralInETH - vars.amountToDecreaseInETH;

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

    vars.liquidationThresholdAfterDecrease = (vars.totalCollateralInETH * vars.avgLiquidationThreshol
      - vars.amountToDecreaseInETH * vars.liquidationThreshold)
      / vars.collateralBalanceAfterDecrease;

    uint256 healthFactorAfterDecrease =
      calculateHealthFactorFromBalances(
        vars.collateralBalanceAfterDecrease,
        vars.totalDebtInETH,
        vars.liquidationThresholdAfterDecrease
      );

    return healthFactorAfterDecrease >= GenericLogic.HEALTH_FACTOR_LIQUIDATION_THRESHOLD;
  }

  struct CalculateUserAccountDataVars {
    uint256 reserveUnitPrice;
    uint256 tokenUnit;
    uint256 compoundedLiquidityBalance;
    uint256 compoundedBorrowBalance;
    uint256 decimals;
    uint256 ltv;
    uint256 liquidationThreshold;
    uint256 i;
    uint256 healthFactor;
    uint256 totalCollateralInETH;
    uint256 totalDebtInETH;
    uint256 avgLtv;
    uint256 avgLiquidationThreshold;
    uint256 reservesLength;
    bool healthFactorBelowThreshold;
    address currentReserveAddress;
    bool usageAsCollateralEnabled;
    bool userUsesReserveAsCollateral;
  }

  /**
   * @dev Calculates the user data across the reserves.
   * this includes the total liquidity/collateral/borrow balances in ETH,
   * the average Loan To Value, the average Liquidation Ratio, and the Health factor.
   * @param user The address of the user
   * @param reserves data of all the reserves
   * @param userConfig The configuration of the user
   * @param reserves The list of the available reserves
   * @param oracle The price oracle address
   * @return The total collateral and total debt of the user in ETH, the avg ltv, liquidation thresho
   **/
  function calculateUserAccountData(
    address user,
    DataTypes.PoolReservesData storage reserves,
    DataTypes.PoolNFTVaultsData storage nftVaults,
    DataTypes.UserConfigurationMap memory userConfig,
    address oracle
  )
    internal
    view
    returns (
      uint256,
      uint256,
      uint256,
      uint256,
      uint256
    )
  {
    CalculateUserAccountDataVars memory vars;

    if (userConfig.isEmpty()) {
      return (0, 0, 0, 0, type(uint256).max);
    }
    for (vars.i = 0; vars.i < reserves.count; vars.i++) {
      if (!userConfig.isUsingAsCollateralOrBorrowing(vars.i)) {
        continue;
      }

      vars.currentReserveAddress = reserves.list[vars.i];
      DataTypes.ReserveData storage currentReserve = reserves.data[vars.currentReserveAddress];

      (vars.ltv, vars.liquidationThreshold, , vars.decimals, ) = currentReserve
        .configuration
        .getParams();

      vars.tokenUnit = 10**vars.decimals;
      vars.reserveUnitPrice = IPriceOracleGetter(oracle).getAssetPrice(vars.currentReserveAddress);

      if (vars.liquidationThreshold != 0 && userConfig.isUsingAsCollateral(vars.i)) {
        vars.compoundedLiquidityBalance = IERC20(currentReserve.vTokenAddress).balanceOf(user);

        uint256 liquidityBalanceETH =
          vars.reserveUnitPrice * vars.compoundedLiquidityBalance / vars.tokenUnit;

        vars.totalCollateralInETH = vars.totalCollateralInETH + liquidityBalanceETH;

        vars.avgLtv = vars.avgLtv + liquidityBalanceETH * vars.ltv;
        vars.avgLiquidationThreshold = vars.avgLiquidationThreshold
          + liquidityBalanceETH * vars.liquidationThreshold;
      }

      if (userConfig.isBorrowing(vars.i)) {
        vars.compoundedBorrowBalance = IERC20(currentReserve.variableDebtTokenAddress).balanceOf(user

        vars.totalDebtInETH = vars.totalDebtInETH
          + vars.reserveUnitPrice * vars.compoundedBorrowBalance / vars.tokenUnit;
      }
    }

    for (vars.i = 0; vars.i < nftVaults.count; vars.i++) {
      if (!userConfig.isUsingNFTVaultAsCollateral(vars.i)) {
        continue;
      }

      vars.currentReserveAddress = nftVaults.list[vars.i];
      DataTypes.NFTVaultData storage currentVault = nftVaults.data[vars.currentReserveAddress];

      (vars.ltv, vars.liquidationThreshold, ) = currentVault
        .configuration
        .getParams();

      vars.reserveUnitPrice = IPriceOracleGetter(oracle).getAssetPrice(vars.currentReserveAddress);

      if (vars.liquidationThreshold != 0 && userConfig.isUsingNFTVaultAsCollateral(vars.i)) {
        vars.compoundedLiquidityBalance = IERC1155Stat(currentVault.nTokenAddress).balanceOf(user);

        uint256 liquidityBalanceETH =
          vars.reserveUnitPrice * vars.compoundedLiquidityBalance;
        vars.totalCollateralInETH = vars.totalCollateralInETH + liquidityBalanceETH;

        vars.avgLtv = vars.avgLtv + liquidityBalanceETH * vars.ltv;
        vars.avgLiquidationThreshold = vars.avgLiquidationThreshold
          + liquidityBalanceETH * vars.liquidationThreshold;
      }
    }

    vars.avgLtv = vars.totalCollateralInETH > 0 ? vars.avgLtv / vars.totalCollateralInETH : 0;
    vars.avgLiquidationThreshold = vars.totalCollateralInETH > 0
      ? vars.avgLiquidationThreshold / vars.totalCollateralInETH
      : 0;

    vars.healthFactor = calculateHealthFactorFromBalances(
      vars.totalCollateralInETH,
      vars.totalDebtInETH,
      vars.avgLiquidationThreshold
    );
    return (
      vars.totalCollateralInETH,
      vars.totalDebtInETH,
      vars.avgLtv,
      vars.avgLiquidationThreshold,
      vars.healthFactor
    );
  }

  /**
   * @dev Calculates the health factor from the corresponding balances
   * @param totalCollateralInETH The total collateral in ETH
   * @param totalDebtInETH The total debt in ETH
   * @param liquidationThreshold The avg liquidation threshold
   * @return The health factor calculated from the balances provided
   **/
  function calculateHealthFactorFromBalances(
    uint256 totalCollateralInETH,
    uint256 totalDebtInETH,
    uint256 liquidationThreshold
  ) internal pure returns (uint256) {
    if (totalDebtInETH == 0) return type(uint256).max;

    return (totalCollateralInETH.percentMul(liquidationThreshold)).wadDiv(totalDebtInETH);
  }

  /**
   * @dev Calculates the equivalent amount in ETH that an user can borrow, depending on the available
   * average Loan To Value
   * @param totalCollateralInETH The total collateral in ETH
   * @param totalDebtInETH The total borrow balance
   * @param ltv The average loan to value
   * @return the amount available to borrow in ETH for the user
   **/

  function calculateAvailableBorrowsETH(
    uint256 totalCollateralInETH,
    uint256 totalDebtInETH,
    uint256 ltv
  ) internal pure returns (uint256) {
    uint256 availableBorrowsETH = totalCollateralInETH.percentMul(ltv);

    if (availableBorrowsETH < totalDebtInETH) {
      return 0;
    }

    availableBorrowsETH = availableBorrowsETH - totalDebtInETH;
    return availableBorrowsETH;
  }
}
```

LendingPool.sol

```
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.8.11;
pragma abicoder v2;

import {IERC20} from '../../dependencies/openzeppelin/contracts/IERC20.sol';
import {GPv2SafeERC20} from '../../dependencies/gnosis/contracts/GPv2SafeERC20.sol';
import {IERC721} from '../../dependencies/openzeppelin/contracts/IERC721.sol';
import {SafeERC721} from '../libraries/helpers/SafeERC721.sol';
import {Address} from '../../dependencies/openzeppelin/contracts/Address.sol';
import {ILendingPoolAddressesProvider} from '../../interfaces/ILendingPoolAddressesProvider.sol';
import {IVToken} from '../../interfaces/IVToken.sol';
import {INToken} from '../../interfaces/INToken.sol';
import {IERC1155Stat} from '../../interfaces/IERC1155Stat.sol';
import {IVariableDebtToken} from '../../interfaces/IVariableDebtToken.sol';
import {IPriceOracleGetter} from '../../interfaces/IPriceOracleGetter.sol';
import {INFTXEligibility} from '../../interfaces/INFTXEligibility.sol';
//import {IStableDebtToken} from '../../interfaces/IStableDebtToken.sol';
import {ILendingPool} from '../../interfaces/ILendingPool.sol';
import {VersionedInitializable} from '../libraries/aave-upgradeability/VersionedInitializable.sol';
import {Helpers} from '../libraries/helpers/Helpers.sol';
import {Errors} from '../libraries/helpers/Errors.sol';
import {WadRayMath} from '../libraries/math/WadRayMath.sol';
import {PercentageMath} from '../libraries/math/PercentageMath.sol';
import {ReserveLogic} from '../libraries/logic/ReserveLogic.sol';
import {NFTVaultLogic} from '../libraries/logic/NFTVaultLogic.sol';
import {GenericLogic} from '../libraries/logic/GenericLogic.sol';
import {ValidationLogic} from '../libraries/logic/ValidationLogic.sol';
import {ReserveConfiguration} from '../libraries/configuration/ReserveConfiguration.sol';
import {NFTVaultConfiguration} from '../libraries/configuration/NFTVaultConfiguration.sol';
import {UserConfiguration} from '../libraries/configuration/UserConfiguration.sol';
import {DataTypes} from '../libraries/types/DataTypes.sol';
import {LendingPoolStorage} from './LendingPoolStorage.sol';

/**
 * @title LendingPool contract
 * @dev Main point of interaction with a Vinci protocol's market
 * - Users can:
 *   # Deposit
 *   # Withdraw
 *   # Deposit NFT
 *   # Withdraw NFT
 *   # Borrow
 *   # Repay
 *   # Enable/disable their NFTs as collateral
 *   # Liquidate positions
 * - To be covered by a proxy contract, owned by the LendingPoolAddressesProvider of the specific mar
 * - All admin functions are callable by the LendingPoolConfigurator contract defined also in the
 *   LendingPoolAddressesProvider
 * @author Aave
 * @author Vinci
 **/
contract LendingPool is VersionedInitializable, ILendingPool, LendingPoolStorage {
  using WadRayMath for uint256;
  using PercentageMath for uint256;
  using GPv2SafeERC20 for IERC20;
  using SafeERC721 for IERC721;
  using ReserveLogic for DataTypes.ReserveData;
  using ReserveConfiguration for DataTypes.ReserveConfigurationMap;
  using NFTVaultLogic for DataTypes.NFTVaultData;
  using NFTVaultConfiguration for DataTypes.NFTVaultConfigurationMap;
  using UserConfiguration for DataTypes.UserConfigurationMap;

  uint256 public constant LENDINGPOOL_REVISION = 0x1;

  modifier whenNotPaused() {
    _whenNotPaused();
    _;
  }

  modifier onlyLendingPoolConfigurator() {
    _onlyLendingPoolConfigurator();
    _;
  }

  function _whenNotPaused() internal view {
    require(!_paused, Errors.LP_IS_PAUSED);
  }

  function _onlyLendingPoolConfigurator() internal view {
    require(
      _addressesProvider.getLendingPoolConfigurator() == msg.sender,
      Errors.LP_CALLER_NOT_LENDING_POOL_CONFIGURATOR
    );
  }

  function getRevision() internal pure override returns (uint256) {
    return LENDINGPOOL_REVISION;
  }

  /**
   * @dev Function is invoked by the proxy contract when the LendingPool contract is added to the
   * LendingPoolAddressesProvider of the market.
   * - Caching the address of the LendingPoolAddressesProvider in order to reduce gas consumption
   *   on subsequent operations
   * @param provider The address of the LendingPoolAddressesProvider
   **/
  function initialize(ILendingPoolAddressesProvider provider) public initializer {
    _addressesProvider = provider;
    _maxStableRateBorrowSizePercent = 2500;
    _flashLoanPremiumTotal = 9;
    _maxNumberOfReserves = 128;
    _maxNumberOfNFTVaults = 256;
  }

  /**
   * @dev Deposits an `amount` of underlying asset into the reserve, receiving in return overlying vT
   * - E.g. User deposits 100 USDC and gets in return 100 aUSDC
   * @param asset The address of the underlying asset to deposit
   * @param amount The amount to be deposited
   * @param onBehalfOf The address that will receive the vTokens, same as msg.sender if the user
   *   wants to receive them on his own wallet, or a different address if the beneficiary of vTokens
   *   is a different wallet
   **/
  function deposit(
    address asset,
    uint256 amount,
    address onBehalfOf,
    uint16 referralCode
  ) external override whenNotPaused {
    DataTypes.ReserveData storage reserve = _reserves.data[asset];

    ValidationLogic.validateDeposit(reserve, amount);
```

```
    address vToken = reserve.vTokenAddress;

    reserve.updateState();
    reserve.updateInterestRates(asset, vToken, amount, 0);

    IERC20(asset).safeTransferFrom(msg.sender, vToken, amount);

    IVToken(vToken).mint(onBehalfOf, amount, reserve.liquidityIndex);

    emit Deposit(asset, msg.sender, onBehalfOf, amount);
  }

  /**
   * @dev Withdraws an `amount` of underlying asset from the reserve, burning the equivalent aTokens
   * E.g. User has 100 vUSDC, calls withdraw() and receives 100 USDC, burning the 100 vUSDC
   * @param asset The address of the underlying asset to withdraw
   * @param amount The underlying amount to be withdrawn
   *   - Send the value type(uint256).max in order to withdraw the whole vToken balance
   * @param to Address that will receive the underlying, same as msg.sender if the user
   *   wants to receive it on his own wallet, or a different address if the beneficiary is a
   *   different wallet
   * @return The final amount withdrawn
   **/
  function withdraw(
    address asset,
    uint256 amount,
    address to
  ) external override whenNotPaused returns (uint256) {
    DataTypes.ReserveData storage reserve = _reserves.data[asset];

    address vToken = reserve.vTokenAddress;

    uint256 userBalance = IVToken(vToken).balanceOf(msg.sender);

    uint256 amountToWithdraw = amount;

    if (amount == type(uint256).max) {
      amountToWithdraw = userBalance;
    }

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

    reserve.updateState();

    reserve.updateInterestRates(asset, vToken, 0, amountToWithdraw);

    IVToken(vToken).burn(msg.sender, to, amountToWithdraw, reserve.liquidityIndex);

    emit Withdraw(asset, msg.sender, to, amountToWithdraw);

    return amountToWithdraw;
  }

  /**
   * @dev Deposits NFTs with given `tokenIds` and `amounts` into the vault, receiving in return overl
   * E.g. User deposits an MAYC with tokenid 1234 and gets in return 1 vnMAYC with tokenid 1234
   * @param nft The address of the underlying asset to deposit
   * @param tokenIds The array of token ids to deposit
   * @param amounts The array of amounts to deposit.
   * - Must be the same length with `tokenIds`.
   * - All elements must be 1 for ERC721 NFT.
   * @param onBehalfOf The address that will receive the vnTokens, same as msg.sender if the user
   *   wants to receive them on his own wallet, or a different address if the beneficiary of vnTokens
   *   is a different wallet
   **/
  function depositNFT(
    address nft,
    uint256[] calldata tokenIds,
    uint256[] calldata amounts,
    address onBehalfOf,
    uint16 referralCode
  ) external override whenNotPaused {
    require(tokenIds.length == amounts.length, Errors.LP_TOKEN_AND_AMOUNT_LENGTH_NOT_MATCH);
    DataTypes.NFTVaultData storage vault = _nftVaults.data[nft];

    ValidationLogic.validateDepositNFT(
      vault,
      tokenIds,
      amounts
    );

    address nToken = vault.nTokenAddress;
    for(uint256 i = 0; i < tokenIds.length; ++i){
      IERC721(nft).safeTransferFrom(msg.sender, nToken, tokenIds[i]);
      bool isFirstDeposit = INToken(nToken).mint(onBehalfOf, tokenIds[i], 1);
      if (isFirstDeposit) {
        _usersConfig[onBehalfOf].setUsingNFTVaultAsCollateral(vault.id, true);
        emit NFTVaultUsedAsCollateralEnabled(nft, onBehalfOf);
      }
    }

    emit DepositNFT(nft, msg.sender, onBehalfOf, tokenIds, amounts);

  }

  /**
   * @dev Withdraws underlying NFTs with given `tokenIds` and `amounts` from the vault, burning the e
   * E.g. User has vnMAYC with token id 1234, calls withdraw() and receives the MAYC with token id 12
   * @param nft The address of the underlying NFT to withdraw
   * @param tokenIds The array of token ids to withdraw
   * @param amounts The array of amounts to withdraw.
   * - Must be the same length with `tokenIds`.
   * - All elements must be 1 for ERC721 NFT.
   * @param to Address that will receive the underlying, same as msg.sender if the user
   *   wants to receive it on his own wallet, or a different address if the beneficiary is a
   *   different wallet
   * @return The final amount withdrawn
   **/
  function withdrawNFT(
    address nft,
    uint256[] calldata tokenIds,
    uint256[] calldata amounts,
    address to
  ) external override whenNotPaused returns (uint256[] memory) {
    require(tokenIds.length == amounts.length, Errors.LP_TOKEN_AND_AMOUNT_LENGTH_NOT_MATCH);
    DataTypes.NFTVaultData storage vault = _nftVaults.data[nft];

    address nToken = vault.nTokenAddress;

    uint256[] memory userBalances = IERC1155Stat(nToken).balanceOfBatch(msg.sender, tokenIds);

    uint256[] memory amountsToWithdraw = amounts;

    uint256 amountToWithdraw = 0;
    for(uint256 i = 0; i < tokenIds.length; ++i){
      if (amounts[i] == type(uint256).max) {
        amountsToWithdraw[i] = userBalances[i];
      }
      amountToWithdraw = amountToWithdraw + amountsToWithdraw[i];
    }

    ValidationLogic.validateWithdrawNFT(
      nft,
      tokenIds,
      amountsToWithdraw,
      userBalances,
      _reserves,
      _nftVaults,
      _usersConfig[msg.sender],
      _addressesProvider.getPriceOracle()
    );

    if (amountToWithdraw == IERC1155Stat(nToken).balanceOf(msg.sender)) {
      _usersConfig[msg.sender].setUsingNFTVaultAsCollateral(vault.id, false);
      emit NFTVaultUsedAsCollateralDisabled(nft, msg.sender);
    }

    INToken(nToken).burnBatch(msg.sender, to, tokenIds, amountsToWithdraw);

    INFTXEligibility(vault.nftEligibility).afterRedeemHook(tokenIds);
    emit WithdrawNFT(nft, msg.sender, to, tokenIds, amountsToWithdraw);

    return amountsToWithdraw;
  }

  /**
   * @dev Allows users to borrow a specific `amount` of the reserve underlying asset, provided that t
   * already deposited enough collateral, or he was given enough allowance by a credit delegator on t
   * corresponding debt token (VariableDebtToken)
   * - E.g. User borrows 100 USDC passing as `onBehalfOf` his own address, receiving the 100 USDC in
   *   and 100 variable debt tokens.
   * @param asset The address of the underlying asset to borrow
   * @param amount The amount to be borrowed
   * @param interestRateMode The interest rate mode at which the user wants to borrow. Unused current
   * @param referralCode Code used to register the integrator originating the operation, for potentia
   *   0 if the action is executed directly by the user, without any middle-man
   * @param onBehalfOf Address of the user who will receive the debt. Should be the address of the bo
   * calling the function if he wants to borrow against his own collateral, or the address of the cre
   * if he has been given credit delegation allowance
   **/
  function borrow(
    address asset,
    uint256 amount,
    uint256 interestRateMode,
    uint16 referralCode,
    address onBehalfOf
  ) external override whenNotPaused {
    DataTypes.ReserveData storage reserve = _reserves.data[asset];

    _executeBorrow(
      ExecuteBorrowParams(
        asset,
        msg.sender,
        onBehalfOf,
        amount,
        interestRateMode,
        reserve.vTokenAddress,
        referralCode,
        true
      )
    );
  }

  /**
   * @notice Repays a borrowed `amount` on a specific reserve, burning the equivalent debt tokens own
   * - E.g. User repays 100 USDC, burning 100 variable debt tokens of the `onBehalfOf` address
   * @param asset The address of the borrowed underlying asset previously borrowed
   * @param amount The amount to repay
   * - Send the value type(uint256).max in order to repay the whole debt for `asset` on the specific
   * @param rateMode The interest rate mode at of the debt the user wants to repay. Unused Currently,
   * @param onBehalfOf Address of the user who will get his debt reduced/removed. Should be the addre
   * user calling the function if he wants to reduce/remove his own debt, or the address of any other
   * other borrower whose debt should be removed
   * @return The final amount repaid
   **/
  function repay(
    address asset,
    uint256 amount,
    uint256 rateMode,
    address onBehalfOf
  ) external override whenNotPaused returns (uint256) {
    DataTypes.ReserveData storage reserve = _reserves.data[asset];

    (, uint256 variableDebt) = Helpers.getUserCurrentDebt(onBehalfOf, reserve);

    DataTypes.InterestRateMode interestRateMode = DataTypes.InterestRateMode(rateMode);

    ValidationLogic.validateRepay(
      reserve,
      amount,
      interestRateMode,
      onBehalfOf,
      0,
      variableDebt
    );

    uint256 paybackAmount = variableDebt;

    if (amount < paybackAmount) {
      paybackAmount = amount;
    }

    reserve.updateState();

    /*if (interestRateMode == DataTypes.InterestRateMode.STABLE) {
      IStableDebtToken(reserve.stableDebtTokenAddress).burn(onBehalfOf, paybackAmount);
    } else {*/
      IVariableDebtToken(reserve.variableDebtTokenAddress).burn(
        onBehalfOf,
        paybackAmount,
        reserve.variableBorrowIndex
      );
    //}

    address vToken = reserve.vTokenAddress;
    reserve.updateInterestRates(asset, vToken, paybackAmount, 0);

    if (variableDebt - paybackAmount == 0) {
      _usersConfig[onBehalfOf].setBorrowing(reserve.id, false);
    }

    IERC20(asset).safeTransferFrom(msg.sender, vToken, paybackAmount);

    IVToken(vToken).handleRepayment(msg.sender, paybackAmount);

    emit Repay(asset, msg.sender, paybackAmount);
    return paybackAmount;
  }

  struct NFTLiquidationCallParameters {
    address collateralAsset;
    address debtAsset;
    address user;
    uint256[] tokenIds;
    uint256[] amounts;
    bool receiveNToken;
  }

  /**
   * @dev Function to liquidate a non-healthy position collateral-wise, with Health Factor below 1
   * - The caller (liquidator) chooses `tokenIds` and `amounts` of the `collateralAsset` NFTs of the
   *   user getting liquidated, and pays the corrensponding discounted `debtAsset` price
   *   to cover the debt of the user getting liquidated.
   * - If there is any `debtAsset` left after repayment of the debt, it will be converted to vToken
   *   and transferred to the user getting liquidated.
   * @param collateralAsset The address of the underlying NFT used as collateral, to receive as resul
   * @param debtAsset The address of the underlying borrowed asset to be repaid with the liquidation
   * @param user The address of the borrower getting liquidated
   * @param tokenIds The array of token ids of the NFTs that the liquidator wants to receive
   * - Starting from the front, only the portion that covers 50% of the debt of the user get liquidat
   * @param amounts The array of ammounts of the NFTs that the liquidator wants to receive
   * - Must be the same length with `tokenIds`
   * @param receiveNToken `true` if the liquidators wants to receive the collateral nTokens, `false`
   * to receive the underlying collateral NFT directly
   **/
  function nftLiquidationCall(
    address collateralAsset,
    address debtAsset,
    address user,
    uint256[] calldata tokenIds,
    uint256[] calldata amounts,
    bool receiveNToken
  ) external override whenNotPaused {
    address collateralManager = _addressesProvider.getLendingPoolCollateralManager();
    NFTLiquidationCallParameters memory params;
    params.collateralAsset = collateralAsset;
    params.debtAsset = debtAsset;
    params.user = user;
    params.tokenIds = tokenIds;
    params.amounts = amounts;
    params.receiveNToken = receiveNToken;
    //solium-disable-next-line
    (bool success, bytes memory result) =
      collateralManager.delegatecall(
        abi.encodeWithSignature(
          'nftLiquidationCall((address,address,address,uint256[],uint256[],bool))',
          params
        )
      );

    require(success, Errors.LP_LIQUIDATION_CALL_FAILED);

    (uint256 returnCode, string memory returnMessage) = abi.decode(result, (uint256, string));

    require(returnCode == 0, string(abi.encodePacked(returnMessage)));
  }

  /**
   * @dev Returns the state and configuration of the reserve
   * @param asset The address of the underlying asset of the reserve
   * @return The state of the reserve
   **/
  function getReserveData(address asset)
    external
    view
    override
    returns (DataTypes.ReserveData memory)
  {
    return _reserves.data[asset];
  }

  /**
   * @dev Returns the state and configuration of the vault
   * @param asset The address of the underlying NFT of the vault
   * @return The state of the vault
   **/
  function getNFTVaultData(address asset)
    external
    view
    override
    returns (DataTypes.NFTVaultData memory)
  {
    return _nftVaults.data[asset];
  }

  /**
   * @dev Returns the user account data across all the reserves
   * @param user The address of the user
   * @return totalCollateralETH the total collateral in ETH of the user
   * @return totalDebtETH the total debt in ETH of the user
   * @return availableBorrowsETH the borrowing power left of the user
   * @return currentLiquidationThreshold the liquidation threshold of the user
   * @return ltv the loan to value of the user
   * @return healthFactor the current health factor of the user
   **/
  function getUserAccountData(address user)
    external
    view
    override
    returns (
      uint256 totalCollateralETH,
      uint256 totalDebtETH,
      uint256 availableBorrowsETH,
      uint256 currentLiquidationThreshold,
      uint256 ltv,
      uint256 healthFactor
    )
  {
    (
      totalCollateralETH,
      totalDebtETH,
      ltv,
      currentLiquidationThreshold,
      healthFactor
    ) = GenericLogic.calculateUserAccountData(
      user,
      _reserves,
      _nftVaults,
      _usersConfig[user],
      _addressesProvider.getPriceOracle()
    );

    availableBorrowsETH = GenericLogic.calculateAvailableBorrowsETH(
      totalCollateralETH,
      totalDebtETH,
      ltv
    );
  }

  /**
   * @dev Returns the configuration of the reserve
   * @param asset The address of the underlying asset of the reserve
   * @return The configuration of the reserve
   **/
  function getConfiguration(address asset)
    external
    view
    override
    returns (DataTypes.ReserveConfigurationMap memory)
  {
    return _reserves.data[asset].configuration;
  }

  /**
   * @dev Returns the configuration of the vault
   * @param asset The address of the underlying nft of the vault
   * @return The configuration of the vault
   **/
  function getNFTVaultConfiguration(address asset)
    external
    view
    override
    returns (DataTypes.NFTVaultConfigurationMap memory)
  {
    return _nftVaults.data[asset].configuration;
  }

  /**
   * @dev Returns the configuration of the user across all the reserves
   * @param user The user address
   * @return The configuration of the user
   **/
  function getUserConfiguration(address user)
    external
    view
    override
    returns (DataTypes.UserConfigurationMap memory)
  {
    return _usersConfig[user];
  }

  /**
   * @dev Returns the normalized income per unit of asset
   * @param asset The address of the underlying asset of the reserve
   * @return The reserve's normalized income
   */
  function getReserveNormalizedIncome(address asset)
    external
    view
    virtual
    override
    returns (uint256)
  {
    return _reserves.data[asset].getNormalizedIncome();
  }

  /**
   * @dev Returns the normalized variable debt per unit of asset
   * @param asset The address of the underlying asset of the reserve
   * @return The reserve normalized variable debt
   */
  function getReserveNormalizedVariableDebt(address asset)
    external
    view
    override
    returns (uint256)
  {
    return _reserves.data[asset].getNormalizedDebt();
  }

  /**
   * @dev Returns if the LendingPool is paused
   */
  function paused() external view override returns (bool) {
    return _paused;
  }

  /**
   * @dev Returns the list of the initialized reserves
   **/
  function getReservesList() external view override returns (address[] memory) {
    address[] memory _activeReserves = new address[](_reserves.count);

    for (uint256 i = 0; i < _reserves.count; i++) {
      _activeReserves[i] = _reserves.list[i];
    }
    return _activeReserves;
  }

  /**
   * @dev Returns the list of the initialized vaults
   **/
  function getNFTVaultsList() external view override returns (address[] memory) {
    address[] memory _activeVaults = new address[](_nftVaults.count);

    for (uint256 i = 0; i < _nftVaults.count; i++) {
      _activeVaults[i] = _nftVaults.list[i];
    }
    return _activeVaults;
  }

  /**
   * @dev Returns the cached LendingPoolAddressesProvider connected to this contract
   **/
  function getAddressesProvider() external view override returns (ILendingPoolAddressesProvider) {
    return _addressesProvider;
  }

  /**
   * @dev Returns the percentage of available liquidity that can be borrowed at once at stable rate
   */
  function MAX_STABLE_RATE_BORROW_SIZE_PERCENT() public view returns (uint256) {
    return _maxStableRateBorrowSizePercent;
  }

  /**
   * @dev Returns the fee on flash loans
   */
  function FLASHLOAN_PREMIUM_TOTAL() public view returns (uint256) {
    return _flashLoanPremiumTotal;
  }

  /**
   * @dev Returns the maximum number of reserves supported to be listed in this LendingPool
   */
  function MAX_NUMBER_RESERVES() public view returns (uint256) {
    return _maxNumberOfReserves;
  }
```

```
/**
 * @dev Validates and finalizes a vToken transfer
 * - Only callable by the overlying vToken of the `asset`
 * @param asset The address of the underlying asset of the vToken
 * @param from The user from which the vTokens are transferred
 * @param to The user receiving the vTokens
 * @param amount The amount being transferred/withdrawn
 * @param balanceFromBefore The vToken balance of the `from` user before the transfer
 * @param balanceToBefore The vToken balance of the `to` user before the transfer
 */
function finalizeTransfer(
  address asset,
  address from,
  address to,
  uint256 amount,
  uint256 balanceFromBefore,
  uint256 balanceToBefore
) external override whenNotPaused {
  require(msg.sender == _reserves.data[asset].vTokenAddress, Errors.LP_CALLER_MUST_BE_AN_VTOKEN);

  ValidationLogic.validateTransfer(
    from,
    _reserves,
    _nftVaults,
    _usersConfig[from],
    _addressesProvider.getPriceOracle()
  );
}

/**
 * @dev Validates and finalizes a nToken transfer
 * - Only callable by the overlying nToken of the `asset`
 * @param asset The address of the underlying NFT of the nToken
 * @param from The user from which the nTokens are transferred
 * @param to The user receiving the nTokens
 * @param tokenId The token id of the NFT being transferred/withdrawn
 * @param amount The amount of the NFT with `tokenId` being transferred/withdrawn
 * @param balanceFromBefore The vToken balance of the `from` user before the transfer
 * @param balanceToBefore The vToken balance of the `to` user before the transfer
 */
function finalizeNFTSingleTransfer(
  address asset,
  address from,
  address to,
  uint256 tokenId,
  uint256 amount,
  uint256 balanceFromBefore,
  uint256 balanceToBefore
) external override whenNotPaused {
  require(msg.sender == _nftVaults.data[asset].nTokenAddress, Errors.LP_CALLER_MUST_BE_AN_VTOKEN);

  ValidationLogic.validateTransfer(
    from,
    _reserves,
    _nftVaults,
    _usersConfig[from],
    _addressesProvider.getPriceOracle()
  );

  uint256 vaultId = _nftVaults.data[asset].id;

  if (from != to) {
    if (balanceFromBefore -amount == 0) {
      DataTypes.UserConfigurationMap storage fromConfig = _usersConfig[from];
      fromConfig.setUsingNFTVaultAsCollateral(vaultId, false);
      emit NFTVaultUsedAsCollateralDisabled(asset, from);
    }

    if (balanceToBefore == 0 && amount != 0) {
      DataTypes.UserConfigurationMap storage toConfig = _usersConfig[to];
      toConfig.setUsingNFTVaultAsCollateral(vaultId, true);
      emit NFTVaultUsedAsCollateralEnabled(asset, to);
    }
  }

}

/**
 * @dev Validates and finalizes a batch nToken transfer
 * - Only callable by the overlying nToken of the `asset`
 * @param asset The address of the underlying NFT of the nTokens
 * @param from The user from which the nTokens are transferred
 * @param to The user receiving the vTokens
 * @param tokenIds The array of token ids of the NFTs being transferred/withdrawn
 * @param amounts The array of amounts of the NFTs being transferred/withdrawn
 * - Must be the same length with `tokenIds`
 * @param balanceFromBefore The vToken balance of the `from` user before the transfer
 * @param balanceToBefore The vToken balance of the `to` user before the transfer
 */
function finalizeNFTBatchTransfer(
  address asset,
  address from,
  address to,
  uint256[] calldata tokenIds,
  uint256[] calldata amounts,
  uint256 balanceFromBefore,
  uint256 balanceToBefore
) external override whenNotPaused {
  require(msg.sender == _nftVaults.data[asset].nTokenAddress, Errors.LP_CALLER_MUST_BE_AN_VTOKEN);

  ValidationLogic.validateTransfer(
    from,
    _reserves,
    _nftVaults,
    _usersConfig[from],
    _addressesProvider.getPriceOracle()
  );

  uint256 vaultId = _nftVaults.data[asset].id;
  uint256 amount = 0;
  for(uint256 i = 0; i < amounts.length; ++i){
    amount = amount + amounts[i];
  }

  if (from != to) {
    if (balanceFromBefore - amount == 0) {
      DataTypes.UserConfigurationMap storage fromConfig = _usersConfig[from];
      fromConfig.setUsingNFTVaultAsCollateral(vaultId, false);
      emit NFTVaultUsedAsCollateralDisabled(asset, from);
    }

    if (balanceToBefore == 0 && amount != 0) {
      DataTypes.UserConfigurationMap storage toConfig = _usersConfig[to];
      toConfig.setUsingNFTVaultAsCollateral(vaultId, true);
      emit NFTVaultUsedAsCollateralEnabled(asset, to);
    }
  }

}

/**
 * @dev Initializes a reserve, activating it, assigning a vToken and debt tokens and an
```

```
   * interest rate strategy
   * - Only callable by the LendingPoolConfigurator contract
   * @param asset The address of the underlying asset of the reserve
   * @param vTokenAddress The address of the vToken that will be assigned to the reserve
   * @param stableDebtAddress The address of the StableDebtToken that will be assigned to the reserve
   * @param vTokenAddress The address of the VariableDebtToken that will be assigned to the reserve
   * @param interestRateStrategyAddress The address of the interest rate strategy contract
   **/
  function initReserve(
    address asset,
    address vTokenAddress,
    address stableDebtAddress,
    address variableDebtAddress,
    address interestRateStrategyAddress
  ) external override onlyLendingPoolConfigurator {
    require(Address.isContract(asset), Errors.LP_NOT_CONTRACT);
    _reserves.data[asset].init(
      vTokenAddress,
      stableDebtAddress,
      variableDebtAddress,
      interestRateStrategyAddress
    );
    _addReserveToList(asset);
  }

  /**
   * @dev Initializes a vault, activating it, assigning a nToken and an eligibility checker
   * - Only callable by the LendingPoolConfigurator contract
   * @param nft The address of the underlying NFT of the vault
   * @param nTokenAddress The address of the nToken that will be assigned to the vault
   * @param nftEligibility The address of the NFTXEligibility contract that will be used for checking
   **/
  function initNFTVault(
    address nft,
    address nTokenAddress,
    address nftEligibility
  ) external override onlyLendingPoolConfigurator {
    require(Address.isContract(nft), Errors.LP_NOT_CONTRACT);
    _nftVaults.data[nft].init(
      nTokenAddress,
      nftEligibility
    );
    _addNFTVaultToList(nft);
  }

  /**
   * @dev Updates the address of the interest rate strategy contract
   * - Only callable by the LendingPoolConfigurator contract
   * @param asset The address of the underlying asset of the reserve
   * @param rateStrategyAddress The address of the interest rate strategy contract
   **/
  function setReserveInterestRateStrategyAddress(address asset, address rateStrategyAddress)
    external
    override
    onlyLendingPoolConfigurator
  {
    _reserves.data[asset].interestRateStrategyAddress = rateStrategyAddress;
  }

  /**
   * @dev Sets the configuration bitmap of the reserve as a whole
   * - Only callable by the LendingPoolConfigurator contract
   * @param asset The address of the underlying asset of the reserve
   * @param configuration The new configuration bitmap
   **/
  function setConfiguration(address asset, uint256 configuration)
```

```
    external
    override
    onlyLendingPoolConfigurator
  {
    _reserves.data[asset].configuration.data = configuration;
  }

  /**
   * @dev Sets the configuration bitmap of the vault as a whole
   * - Only callable by the LendingPoolConfigurator contract
   * @param vault The address of the underlying NFT of the vault
   * @param configuration The new configuration bitmap
   **/
  function setNFTVaultConfiguration(address vault, uint256 configuration)
   external
   override
   onlyLendingPoolConfigurator
  {
    _nftVaults.data[vault].configuration.data = configuration;
  }

  /**
   * @dev Set the _pause state of a reserve
   * - Only callable by the LendingPoolConfigurator contract
   * @param val `true` to pause the reserve, `false` to un-pause it
   */
  function setPause(bool val) external override onlyLendingPoolConfigurator {
    _paused = val;
    if (_paused) {
      emit Paused();
    } else {
      emit Unpaused();
    }
  }

  struct ExecuteBorrowParams {
    address asset;
    address user;
    address onBehalfOf;
    uint256 amount;
    uint256 interestRateMode;
    address vTokenAddress;
    uint16 referralCode;
    bool releaseUnderlying;
  }

  function _executeBorrow(ExecuteBorrowParams memory vars) internal {
    DataTypes.ReserveData storage reserve = _reserves.data[vars.asset];
    DataTypes.UserConfigurationMap storage userConfig = _usersConfig[vars.onBehalfOf];

    address oracle = _addressesProvider.getPriceOracle();

    uint256 amountInETH =
      IPriceOracleGetter(oracle).getAssetPrice(vars.asset) * vars.amount
        / (10**reserve.configuration.getDecimals());

    ValidationLogic.validateBorrow(
      vars.asset,
      reserve,
      vars.onBehalfOf,
      vars.amount,
      amountInETH,
      vars.interestRateMode,
      0, //_maxStableRateBorrowSizePercent,
      _reserves,
      _nftVaults,
      userConfig,
      oracle
    );

    reserve.updateState();

    //uint256 currentStableRate = 0;

    bool isFirstBorrowing = false;
    /*if (DataTypes.InterestRateMode(vars.interestRateMode) == DataTypes.InterestRateMode.STABLE) {
      currentStableRate = reserve.currentStableBorrowRate;

      isFirstBorrowing = IStableDebtToken(reserve.stableDebtTokenAddress).mint(
        vars.user,
        vars.onBehalfOf,
        vars.amount,
        currentStableRate
      );
    } else {*/
      isFirstBorrowing = IVariableDebtToken(reserve.variableDebtTokenAddress).mint(
        vars.user,
        vars.onBehalfOf,
        vars.amount,
        reserve.variableBorrowIndex
      );
    //}

    if (isFirstBorrowing) {
      userConfig.setBorrowing(reserve.id, true);
    }

    reserve.updateInterestRates(
      vars.asset,
      vars.vTokenAddress,
      0,
      vars.releaseUnderlying ? vars.amount : 0
    );

    if (vars.releaseUnderlying) {
      IVToken(vars.vTokenAddress).transferUnderlyingTo(vars.user, vars.amount);
    }

    emit Borrow(
      vars.asset,
      vars.user,
      vars.amount,
      vars.interestRateMode,
      //DataTypes.InterestRateMode(vars.interestRateMode) == DataTypes.InterestRateMode.STABLE
      //  ? currentStableRate
      reserve.currentVariableBorrowRate
    );
  }

  function _addReserveToList(address asset) internal {
    uint256 reservesCount = _reserves.count;

    require(reservesCount < _maxNumberOfReserves, Errors.LP_NO_MORE_RESERVES_ALLOWED);

    bool reserveAlreadyAdded = _reserves.data[asset].id != 0 || _reserves.list[0] == asset;

    if (!reserveAlreadyAdded) {
      _reserves.data[asset].id = uint8(reservesCount);
      _reserves.list[reservesCount] = asset;

      _reserves.count = reservesCount + 1;
    }
  }

  function _addNFTVaultToList(address nft) internal {
    uint256 nftVaultsCount = _nftVaults.count;

    require(nftVaultsCount < _maxNumberOfNFTVaults, Errors.LP_NO_MORE_RESERVES_ALLOWED);

    bool vaultAlreadyAdded = _nftVaults.data[nft].id != 0 || _nftVaults.list[0] == nft;

    if (!vaultAlreadyAdded) {
      _nftVaults.data[nft].id = uint8(nftVaultsCount);
      _nftVaults.list[nftVaultsCount] = nft;

      _nftVaults.count = nftVaultsCount + 1;
    }
  }
}
```

UserConfiguration.sol

```
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.8.11;

import {Errors} from '../helpers/Errors.sol';
import {DataTypes} from '../types/DataTypes.sol';

/**
 * @title UserConfiguration library
 * @author Aave
 * @notice Implements the bitmap logic to handle the user configuration
 */
library UserConfiguration {
  uint256 internal constant BORROWING_MASK =
    0x5555555555555555555555555555555555555555555555555555555555555555;

  /**
   * @dev Sets if the user is borrowing the reserve identified by reserveIndex
   * @param self The configuration object
   * @param reserveIndex The index of the reserve in the bitmap
   * @param borrowing True if the user is borrowing the reserve, false otherwise
   **/
  function setBorrowing(
    DataTypes.UserConfigurationMap storage self,
    uint256 reserveIndex,
    bool borrowing
  ) internal {
    require(reserveIndex < 128, Errors.UL_INVALID_INDEX);
    self.data =
      (self.data & ~(1 << (reserveIndex * 2))) |
      (uint256(borrowing ? 1 : 0) << (reserveIndex * 2));
  }

  /**
   * @dev Sets if the user is using as collateral the reserve identified by reserveIndex
   * @param self The configuration object
   * @param reserveIndex The index of the reserve in the bitmap
   * @param usingAsCollateral True if the user is usin the reserve as collateral, false otherwise
   **/
  function setUsingAsCollateral(
    DataTypes.UserConfigurationMap storage self,
    uint256 reserveIndex,
    bool usingAsCollateral
  ) internal {
    require(reserveIndex < 128, Errors.UL_INVALID_INDEX);
    self.data =
      (self.data & ~(1 << (reserveIndex * 2 + 1))) |
      (uint256(usingAsCollateral ? 1 : 0) << (reserveIndex * 2 + 1));
  }

  /**
   * @dev Sets if the user is using as collateral the nft vault identified by vaultIndex
   * @param self The configuration object
   * @param vaultIndex The index of the nft vault in the bitmap
   * @param usingAsCollateral True if the user is usin the nft vault as collateral, false otherwise
   **/
  function setUsingNFTVaultAsCollateral(
    DataTypes.UserConfigurationMap storage self,
    uint256 vaultIndex,
    bool usingAsCollateral
  ) internal {
    require(vaultIndex < 256, Errors.UL_INVALID_INDEX);
    self.nData =
      (self.nData & ~(1 << vaultIndex)) |
      (uint256(usingAsCollateral ? 1 : 0) << vaultIndex);
  }

  /**
   * @dev Used to validate if a user has been using the reserve for borrowing or as collateral
   * @param self The configuration object
   * @param reserveIndex The index of the reserve in the bitmap
   * @return True if the user has been using a reserve for borrowing or as collateral, false otherwis
   **/
  function isUsingAsCollateralOrBorrowing(
    DataTypes.UserConfigurationMap memory self,
    uint256 reserveIndex
  ) internal pure returns (bool) {
    require(reserveIndex < 128, Errors.UL_INVALID_INDEX);
    return (self.data >> (reserveIndex * 2)) & 3 != 0;
  }

  /**
   * @dev Used to validate if a user has been using the reserve for borrowing
   * @param self The configuration object
   * @param reserveIndex The index of the reserve in the bitmap
   * @return True if the user has been using a reserve for borrowing, false otherwise
   **/
  function isBorrowing(DataTypes.UserConfigurationMap memory self, uint256 reserveIndex)
    internal
    pure
    returns (bool)
  {
    require(reserveIndex < 128, Errors.UL_INVALID_INDEX);
    return (self.data >> (reserveIndex * 2)) & 1 != 0;
  }

  /**
   * @dev Used to validate if a user has been using the reserve as collateral
   * @param self The configuration object
   * @param reserveIndex The index of the reserve in the bitmap
   * @return True if the user has been using a reserve as collateral, false otherwise
   **/
  function isUsingAsCollateral(DataTypes.UserConfigurationMap memory self, uint256 reserveIndex)
    internal
    pure
    returns (bool)
  {
    require(reserveIndex < 128, Errors.UL_INVALID_INDEX);
    return (self.data >> (reserveIndex * 2 + 1)) & 1 != 0;
  }
  /**
   * @dev Used to validate if a user has been using the nft vault as collateral
   * @param self The configuration object
   * @param vaultIndex The index of the nft vault in the bitmap
   * @return True if the user has been using a nft vault as collateral, false otherwise
   **/
  function isUsingNFTVaultAsCollateral(DataTypes.UserConfigurationMap memory self, uint256 vaultIndex
    internal
    pure
    returns (bool)
  {
    require(vaultIndex < 256, Errors.UL_INVALID_INDEX);
    return (self.nData >> vaultIndex) & 1 != 0;
  }

  /**
   * @dev Used to validate if a user has been borrowing from any reserve
   * @param self The configuration object
   * @return True if the user has been borrowing any reserve, false otherwise
   **/
  function isBorrowingAny(DataTypes.UserConfigurationMap memory self) internal pure returns (bool) {
    return self.data & BORROWING_MASK != 0;
  }

  /**
   * @dev Used to validate if a user has not been using any reserve
   * @param self The configuration object
   * @return True if the user has been borrowing any reserve, false otherwise
   **/
  function isEmpty(DataTypes.UserConfigurationMap memory self) internal pure returns (bool) {
    return self.data == 0 && self.nData == 0;
  }
}
```

LendingPoolCollateralManager.sol

```
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.8.11;
pragma abicoder v2;

import {IERC20} from '../../dependencies/openzeppelin/contracts//IERC20.sol';
import {IVToken} from '../../interfaces/IVToken.sol';
import {INToken} from '../../interfaces/INToken.sol';
import {IERC1155Stat} from '../../interfaces/IERC1155Stat.sol';
//import {IStableDebtToken} from '../../interfaces/IStableDebtToken.sol';
import {IVariableDebtToken} from '../../interfaces/IVariableDebtToken.sol';
import {IPriceOracleGetter} from '../../interfaces/IPriceOracleGetter.sol';
import {ILendingPoolCollateralManager} from '../../interfaces/ILendingPoolCollateralManager.sol';
import {INFTXEligibility} from '../../interfaces/INFTXEligibility.sol';
import {VersionedInitializable} from '../libraries/aave-upgradeability/VersionedInitializable.sol';
import {GenericLogic} from '../libraries/logic/GenericLogic.sol';
import {Helpers} from '../libraries/helpers/Helpers.sol';
import {WadRayMath} from '../libraries/math/WadRayMath.sol';
import {PercentageMath} from '../libraries/math/PercentageMath.sol';
import {GPv2SafeERC20} from '../../dependencies/gnosis/contracts/GPv2SafeERC20.sol';
import {Errors} from '../libraries/helpers/Errors.sol';
import {ReserveLogic} from '../libraries/logic/ReserveLogic.sol';
//import {NFTVaultLogic} from '../libraries/logic/NFTVaultLogic.sol';
import {ReserveConfiguration} from '../libraries/configuration/ReserveConfiguration.sol';
import {NFTVaultConfiguration} from '../libraries/configuration/NFTVaultConfiguration.sol';
import {UserConfiguration} from '../libraries/configuration/UserConfiguration.sol';
import {ValidationLogic} from '../libraries/logic/ValidationLogic.sol';
import {DataTypes} from '../libraries/types/DataTypes.sol';
import {LendingPoolStorage} from './LendingPoolStorage.sol';

/**
 * @title LendingPoolCollateralManager contract
 * @author Aave
 * @dev Implements actions involving management of collateral in the protocol, the main one being the
 * IMPORTANT This contract will run always via DELEGATECALL, through the LendingPool, so the chain of
 * is the same as the LendingPool, to have compatible storage layouts
 **/
contract LendingPoolCollateralManager is
  ILendingPoolCollateralManager,
  VersionedInitializable,
  LendingPoolStorage
{
  using GPv2SafeERC20 for IERC20;
  using WadRayMath for uint256;
  using PercentageMath for uint256;
  using ReserveLogic for DataTypes.ReserveData;
  using ReserveConfiguration for DataTypes.ReserveConfigurationMap;
  //using NFTVaultLogic for DataTypes.NFTVaultData;
  using NFTVaultConfiguration for DataTypes.NFTVaultConfigurationMap;
  using UserConfiguration for DataTypes.UserConfigurationMap;

  uint256 internal constant LIQUIDATION_CLOSE_FACTOR_PERCENT = 5000;

  /**
   * @dev As thIS contract extends the VersionedInitializable contract to match the state
   * of the LendingPool contract, the getRevision() function is needed, but the value is not
   * important, as the initialize() function will never be called here
   */
  function getRevision() internal pure override returns (uint256) {
    return 0;
  }

  struct NFTLiquidationCallLocalVars {
    uint256[] maxCollateralAmountsToLiquidate;
    //uint256 userStableDebt;
    uint256 userVariableDebt;
    //uint256 userTotalDebt;
    uint256 maxLiquidatableDebt;
    uint256 actualDebtToLiquidate;
    uint256 liquidationRatio;
    //uint256 userStableRate;
    uint256 extraDebtAssetToPay;
    uint256 healthFactor;
    uint256 userCollateralBalance;
    uint256 totalCollateralToLiquidate;
    uint256 liquidatorPreviousNTokenBalance;
    INToken collateralNtoken;
    IERC1155Stat collateralTokenData;
    bool isCollateralEnabled;
    DataTypes.InterestRateMode borrowRateMode;
    uint256 errorCode;
    string errorMsg;
  }

  struct AvailableNFTCollateralToLiquidateParameters {
    address collateralAsset;
    address debtAsset;
    address user;
    uint256 debtToCover;
    uint256 userTotalDebt;
    uint256[] tokenIdsToLiquidate;
    uint256[] amountsToLiquidate;
  }
```

```
function nftLiquidationCall(
  NFTLiquidationCallParameters calldata params
) external override returns (uint256, string memory) {
  DataTypes.NFTVaultData storage collateralVault = _nftVaults.data[params.collateralAsset];
  DataTypes.ReserveData storage debtReserve = _reserves.data[params.debtAsset];
  DataTypes.UserConfigurationMap storage userConfig = _usersConfig[params.user];

  NFTLiquidationCallLocalVars memory vars;

  (, , , , vars.healthFactor) = GenericLogic.calculateUserAccountData(
    params.user,
    _reserves,
    _nftVaults,
    userConfig,
    _addressesProvider.getPriceOracle()
  );

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

  (vars.errorCode, vars.errorMsg) = ValidationLogic.validateNFTLiquidationCall(
    collateralVault,
    debtReserve,
    userConfig,
    vars.healthFactor,
    0,//vars.userStableDebt,
    vars.userVariableDebt
  );

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

  vars.collateralNtoken = INToken(collateralVault.nTokenAddress);
  vars.collateralTokenData = IERC1155Stat(collateralVault.nTokenAddress);

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

  vars.actualDebtToLiquidate = vars.maxLiquidatableDebt;
  {
    AvailableNFTCollateralToLiquidateParameters memory callparams;
    callparams.collateralAsset = params.collateralAsset;
    callparams.debtAsset = params.debtAsset;
    callparams.user = params.user;
    callparams.debtToCover = vars.actualDebtToLiquidate;
    callparams.userTotalDebt = vars.userVariableDebt;
    callparams.tokenIdsToLiquidate = params.tokenIds;
    callparams.amountsToLiquidate = params.amounts;
    (
      vars.maxCollateralAmountsToLiquidate,
      vars.totalCollateralToLiquidate,
      vars.actualDebtToLiquidate,
      vars.extraDebtAssetToPay
    ) = _calculateAvailableNFTCollateralToLiquidate(
      collateralVault,
      debtReserve,
      callparams
    );
  }

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

```
    debtReserve.updateState();

    IVariableDebtToken(debtReserve.variableDebtTokenAddress).burn(
      params.user,
      vars.actualDebtToLiquidate,
      debtReserve.variableBorrowIndex
    );

    debtReserve.updateInterestRates(
      params.debtAsset,
      debtReserve.vTokenAddress,
      vars.actualDebtToLiquidate + vars.extraDebtAssetToPay,
      0
    );

    if (params.receiveNToken) {
      vars.liquidatorPreviousNTokenBalance = vars.collateralTokenData.balanceOf(msg.sender);
      vars.collateralNtoken.transferOnLiquidation(params.user, msg.sender, params.tokenIds, vars.maxC

      if (vars.liquidatorPreviousNTokenBalance == 0) {
        {
          DataTypes.UserConfigurationMap storage liquidatorConfig = _usersConfig[msg.sender];
          liquidatorConfig.setUsingNFTVaultAsCollateral(collateralVault.id, true);
        }
        emit NFTVaultUsedAsCollateralEnabled(params.collateralAsset, msg.sender);
      }
    } else {
      // Burn the equivalent amount of nToken, sending the underlying to the liquidator
      vars.collateralNtoken.burnBatch(
        params.user,
        msg.sender,
        params.tokenIds,
        vars.maxCollateralAmountsToLiquidate
      );
      INFTXEligibility(collateralVault.nftEligibility).afterLiquidationHook(params.tokenIds, vars.max
    }

    // If the collateral being liquidated is equal to the user balance,
    // we set the currency as not being used as collateral anymore
    if (vars.totalCollateralToLiquidate == vars.collateralTokenData.balanceOf(params.user)) {
      userConfig.setUsingNFTVaultAsCollateral(collateralVault.id, false);
      emit NFTVaultUsedAsCollateralDisabled(params.collateralAsset, params.user);
    }

    // Transfers the debt asset being repaid to the vToken, where the liquidity is kept
    IERC20(params.debtAsset).safeTransferFrom(
      msg.sender,
      debtReserve.vTokenAddress,
      vars.actualDebtToLiquidate + vars.extraDebtAssetToPay
    );

    if (vars.extraDebtAssetToPay != 0) {
      IVToken(debtReserve.vTokenAddress).mint(params.user, vars.extraDebtAssetToPay, debtReserve.liqu
      emit Deposit(params.debtAsset, msg.sender, params.user, vars.extraDebtAssetToPay);
    }

    NFTLiquidationCallData memory data;
    data.debtToCover = vars.actualDebtToLiquidate;
    data.extraAssetToPay = vars.extraDebtAssetToPay;
    data.liquidatedCollateralTokenIds = params.tokenIds;
    data.liquidatedCollateralAmounts = vars.maxCollateralAmountsToLiquidate;
    data.liquidator = msg.sender;
    data.receiveNToken = params.receiveNToken;
    emit NFTLiquidationCall(
      params.collateralAsset,
      params.debtAsset,
      params.user,
      abi.encode(data)
    );

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

  struct AvailableNFTCollateralToLiquidateLocalVars {
    uint256 userCompoundedBorrowBalance;
    uint256 liquidationBonus;
    uint256 collateralPrice;
    uint256 debtAssetPrice;
    uint256 valueOfDebtToLiquidate;
    uint256 valueOfAllDebt;
    uint256 valueOfCollateral;
    uint256 extraDebtAssetToPay;
    uint256 maxCollateralBalanceToLiquidate;
    uint256 totalCollateralBalanceToLiquidate;
    uint256[] collateralAmountsToLiquidate;
    uint256 debtAssetDecimals;
  }



  /**
   * @dev Calculates how much of a specific collateral can be liquidated, given
   * a certain amount of debt asset.
   * - This function needs to be called after all the checks to validate the liquidation have been pe
   *   otherwise it might fail.
   * @param collateralVault The data of the collateral vault
   * @param debtReserve The data of the debt reserve
   * @return collateralAmounts: The maximum amount that is possible to liquidate given all the liquid
   *                           (user balance, close factor)
   *         debtAmountNeeded: The amount to repay with the liquidation
   **/
  function _calculateAvailableNFTCollateralToLiquidate(
    DataTypes.NFTVaultData storage collateralVault,
    DataTypes.ReserveData storage debtReserve,
    AvailableNFTCollateralToLiquidateParameters memory params
  ) internal view returns (uint256[] memory, uint256, uint256, uint256) {
    uint256 debtAmountNeeded = 0;
    IPriceOracleGetter oracle = IPriceOracleGetter(_addressesProvider.getPriceOracle());

    AvailableNFTCollateralToLiquidateLocalVars memory vars;
    vars.collateralAmountsToLiquidate = new uint256[](params.amountsToLiquidate.length);


    vars.collateralPrice = oracle.getAssetPrice(params.collateralAsset);
    vars.debtAssetPrice = oracle.getAssetPrice(params.debtAsset);

    (, , vars.liquidationBonus) = collateralVault
      .configuration
      .getParams();
    vars.debtAssetDecimals = debtReserve.configuration.getDecimals();

    // This is the maximum possible amount of the selected collateral that can be liquidated, given t
    // max amount of liquidatable debt
    vars.valueOfCollateral = vars.collateralPrice * (10**vars.debtAssetDecimals);
    vars.maxCollateralBalanceToLiquidate = ((vars.debtAssetPrice * params.debtToCover)
      .percentMul(vars.liquidationBonus)
      + vars.valueOfCollateral - 1)
      / vars.valueOfCollateral;
    (vars.totalCollateralBalanceToLiquidate, vars.collateralAmountsToLiquidate) =
      INToken(collateralVault.nTokenAddress).getLiquidationAmounts(params.user, vars.maxCollateralBal

    debtAmountNeeded = (vars.valueOfCollateral
        * vars.totalCollateralBalanceToLiquidate
        / vars.debtAssetPrice)
        .percentDiv(vars.liquidationBonus);
    if (vars.totalCollateralBalanceToLiquidate < vars.maxCollateralBalanceToLiquidate) {
      vars.extraDebtAssetToPay = 0;
    } else if (debtAmountNeeded <= params.userTotalDebt){
      vars.extraDebtAssetToPay = 0;
    } else {
      vars.extraDebtAssetToPay = debtAmountNeeded - params.userTotalDebt;
      debtAmountNeeded = params.userTotalDebt;
    }
    return (vars.collateralAmountsToLiquidate, vars.totalCollateralBalanceToLiquidate, debtAmountNeed
  }
}
```

NToken.sol

```
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.8.11;

import {IERC1155} from '../../dependencies/openzeppelin/contracts/IERC1155.sol';
import {IERC721} from '../../dependencies/openzeppelin/contracts/IERC721.sol';
import {IERC165} from '../../dependencies/openzeppelin/contracts/IERC165.sol';
import {INToken} from '../../interfaces/INToken.sol';
import {ILendingPool} from '../../interfaces/ILendingPool.sol';
import {WadRayMath} from '../libraries/math/WadRayMath.sol';
import {Errors} from '../libraries/helpers/Errors.sol';
import {VersionedInitializable} from '../libraries/aave-upgradeability/VersionedInitializable.sol';
import {WrappedERC1155} from './WrappedERC1155.sol';
import {SafeERC721} from '../libraries/helpers/SafeERC721.sol';
import {IERC721Receiver} from '../../dependencies/openzeppelin/contracts/IERC721Receiver.sol';
import {IERC1155Receiver} from '../../dependencies/openzeppelin/contracts/IERC1155Receiver.sol';

/**
 * @title Vinci ERC1155 NToken
 * @dev Implementation of the deposited NFT token for the Vinci Protocol
 * @author Vinci
 */
 contract NToken is
   VersionedInitializable,
   WrappedERC1155,
   IERC721Receiver,
   IERC1155Receiver,
   INToken
{
    // TODO ERC1155 or ERC1155Burnable?
    using WadRayMath for uint256;
    using SafeERC721 for IERC721;

    uint256 public constant NTOKEN_REVISION = 0x1;
    bytes public constant EIP712_REVISION = bytes('1');

    bytes32 internal constant EIP712_DOMAIN =
    keccak256('EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)');
    bytes32 public constant PERMIT_TYPEHASH =
    keccak256('Permit(address owner,address spender,uint256 tokenId,uint256 amount, uint256 nonce,uin

    /// @dev owner => next valid nonce to submit with permit()
    mapping(address => uint256) public _nonces;

    bytes32 public DOMAIN_SEPARATOR; // TODO

    ILendingPool internal _pool;
```

```
address internal _underlyingNFT;

  modifier onlyLendingPool {
    require(_msgSender() == address(_pool), Errors.CT_CALLER_MUST_BE_LENDING_POOL);
    _;
}

function getRevision() internal pure virtual override returns (uint256) {
    return NTOKEN_REVISION;
}

/** TODO check the initialize, implement _setName and _setSymbol, and use _setURI to set the _uri
 * @dev Initializes the NToken
 * @param pool The address of the lending pool where this nToken will be used
 * @param underlyingNFT The address of the underlying NFT of this nToken
 * @param nTokenName The name of the vToken
 * @param nTokenSymbol The symbol of the vToken
*/
function initialize(
    ILendingPool pool,
    address underlyingNFT,
    string calldata nTokenName,
    string calldata nTokenSymbol,
    bytes calldata params
) external override initializer {
    uint256 chainId;

    //solium-disable-next-line
    assembly {
    chainId := chainid()
    }

    DOMAIN_SEPARATOR = keccak256(
    abi.encode(
        EIP712_DOMAIN,
        keccak256(bytes(nTokenName)),
        keccak256(EIP712_REVISION),
        chainId,
        address(this)
    )
    );

    _setName(nTokenName);
    _setSymbol(nTokenSymbol);

    _pool = pool;
    _underlyingNFT = underlyingNFT;

    emit Initialized(
    underlyingNFT,
    address(pool),
    nTokenName,
    nTokenSymbol,
    params
    );
}

function burn(
    address user,
    address receiverOfUnderlying,
    uint256 tokenId,
    uint256 amount
) external override onlyLendingPool {
  require(amount != 0, Errors.CT_INVALID_BURN_AMOUNT);
  _burn(user, tokenId, amount);
  IERC721(_underlyingNFT).safeTransferFrom(address(this), receiverOfUnderlying, tokenId, "");
```

```
    emit TransferSingle(msg.sender, user, address(0), tokenId, amount);
    emit Burn(user, receiverOfUnderlying, tokenId, amount);
  }

  function burnBatch(
      address user,
      address receiverOfUnderlying,
      uint256[] calldata tokenIds,
      uint256[] calldata amounts
  ) external override onlyLendingPool {
    _burnBatch(user, tokenIds, amounts);
    for(uint256 i = 0; i < tokenIds.length; ++i){
      uint256 id = tokenIds[i];
      IERC721(_underlyingNFT).safeTransferFrom(address(this), receiverOfUnderlying, id, "");
    }

    emit TransferBatch(msg.sender, user, address(0), tokenIds, amounts);
    emit BurnBatch(user, receiverOfUnderlying, tokenIds, amounts);
  }

  function mint(
      address user,
      uint256 tokenId,
      uint256 amount
  ) external override onlyLendingPool returns (bool) {
    uint256 previousNFTAmount = super._balanceOf(user);
    require(amount != 0, Errors.CT_INVALID_MINT_AMOUNT);
    _mint(user, tokenId, amount, "");
   emit TransferSingle(msg.sender, address(0), user, tokenId, amount);
   emit Mint(user, tokenId, amount);

   return previousNFTAmount == 0;
  }

  function transferOnLiquidation(
   address from,
   address to,
   uint256[] calldata tokenIds,
   uint256[] calldata amounts
 ) external override onlyLendingPool {
   // Being a normal transfer, the Transfer() and BalanceTransfer() are emitted
   // so no need to emit a specific event here
   _safeBatchTransferFrom(from, to, tokenIds, amounts, '', false);

   emit TransferBatch(msg.sender, from, to, tokenIds, amounts);
 }

 /**
  * @dev Returns the address of the underlying NFT of this nToken
  **/
 function UNDERLYING_ASSET_ADDRESS() public view override returns (address) {
   return _underlyingNFT;
 }

 /**
  * @dev Returns the address of the lending pool where this nToken is used
  **/
 function POOL() public view returns (ILendingPool) {
     return _pool;
 }

 function transferUnderlyingTo(address target, uint256 tokenId, uint256 amount)
   external
   override
   onlyLendingPool
```

```
    returns (uint256)
  {
    IERC721(_underlyingNFT).safeTransferFrom(_msgSender(), target, tokenId, '');
    return amount;
  }

  function getLiquidationAmounts(address user, uint256 maxTotal, uint256[] calldata tokenIds, uint256
    external
    view
    override
    returns(uint256, uint256[] memory)
  {
    require(user != address(0), "ERC1155: balance query for the zero address");
    require(tokenIds.length == amounts.length, "ERC1155: tokenIds and amounts length mismatch");
    uint256[] memory resultAmounts = new uint256[](tokenIds.length);
    uint256 remainTotal = maxTotal;
    uint256 i = 0;
    while(remainTotal > 0 && i < tokenIds.length) {
      uint256 vaultAmount = _balances[tokenIds[i]][user];
      uint256 parameterAmount = amounts[i];
      uint256 currentAmount = vaultAmount < parameterAmount
        ? vaultAmount
        : parameterAmount;
      if(remainTotal > currentAmount) {
        resultAmounts[i] = currentAmount;
        remainTotal = remainTotal - currentAmount;
      }
      else{
        resultAmounts[i] = remainTotal;
        remainTotal = 0;
      }
      ++i;
    }
    for(; i < tokenIds.length; ++i) {
      resultAmounts[i] = 0;
    }
    return(maxTotal - remainTotal, resultAmounts);
  }

  function supportsInterface(bytes4 interfaceId) public view virtual override(IERC165, WrappedERC1155
    return
      interfaceId == type(IERC721Receiver).interfaceId ||
      interfaceId == type(IERC1155Receiver).interfaceId ||
      super.supportsInterface(interfaceId);
  }

  function onERC721Received(
    address operator,
    address from,
    uint256 tokenId,
    bytes calldata data
  ) external override returns (bytes4)
  {
    return this.onERC721Received.selector;
  }

  function onERC1155Received(
    address,
    address,
    uint256,
    uint256,
    bytes memory
  ) public virtual override returns (bytes4) {
    return this.onERC1155Received.selector;
  }
  function onERC1155BatchReceived(
    address,
    address,
    uint256[] memory,
    uint256[] memory,
    bytes memory
  ) public virtual override returns (bytes4) {
    return this.onERC1155BatchReceived.selector;
  }

  function _safeTransferFrom(address from, address to, uint256 tokenId, uint256 amount, bytes memory
    address underlyingNFT = _underlyingNFT;
    ILendingPool pool = _pool;

    uint256 fromNFTAmountBefore = super._balanceOf(from);
    uint256 toNFTAmountBefore = super._balanceOf(to);
    super._safeTransferFrom(from, to, tokenId, amount, data);
    if(validate){
      pool.finalizeNFTSingleTransfer(underlyingNFT, from, to, tokenId, amount, fromNFTAmountBefore, t
    }
    emit BalanceTransfer(from, to, tokenId, amount);
  }

  function _safeTransferFrom(address from, address to, uint256 tokenId, uint256 amount, bytes memory
    _safeTransferFrom(from, to, tokenId, amount, data, true);
  }

  function _safeBatchTransferFrom(address from, address to, uint256[] calldata tokenIds, uint256[] ca
    address underlyingNFT = _underlyingNFT;
    ILendingPool pool = _pool;

    uint256 fromNFTAmountBefore = super._balanceOf(from);
    uint256 toNFTAmountBefore = super._balanceOf(to);
    super._safeBatchTransferFrom(from, to, tokenIds, amounts, data);
    if(validate){
      pool.finalizeNFTBatchTransfer(underlyingNFT, from, to, tokenIds, amounts, fromNFTAmountBefore,
    }
    emit BalanceBatchTransfer(from, to, tokenIds, amounts);

  }

  function _safeBatchTransferFrom(address from, address to, uint256[] calldata tokenIds, uint256[] ca
    _safeBatchTransferFrom(from, to, tokenIds, amounts, data, true);
  }


}
```

LendingPoolStorage.sol

```
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.8.11;

import {UserConfiguration} from '../libraries/configuration/UserConfiguration.sol';
import {ReserveConfiguration} from '../libraries/configuration/ReserveConfiguration.sol';
import {NFTVaultConfiguration} from '../libraries/configuration/NFTVaultConfiguration.sol';
import {ReserveLogic} from '../libraries/logic/ReserveLogic.sol';
import {NFTVaultLogic} from '../libraries/logic/NFTVaultLogic.sol';
import {ILendingPoolAddressesProvider} from '../../interfaces/ILendingPoolAddressesProvider.sol';
import {INFTXEligibility} from '../../interfaces/INFTXEligibility.sol';
import {DataTypes} from '../libraries/types/DataTypes.sol';

contract LendingPoolStorage {
  using ReserveLogic for DataTypes.ReserveData;
  using ReserveConfiguration for DataTypes.ReserveConfigurationMap;
  using NFTVaultLogic for DataTypes.NFTVaultData;
  using NFTVaultConfiguration for DataTypes.NFTVaultConfigurationMap;
  using UserConfiguration for DataTypes.UserConfigurationMap;

  ILendingPoolAddressesProvider internal _addressesProvider;
  INFTXEligibility internal _nftEligibility;

  DataTypes.PoolReservesData internal _reserves;
  DataTypes.PoolNFTVaultsData internal _nftVaults;

  mapping(address => DataTypes.UserConfigurationMap) internal _usersConfig;

  bool internal _paused;

  uint256 internal _maxStableRateBorrowSizePercent;

  uint256 internal _flashLoanPremiumTotal;

  uint256 internal _maxNumberOfReserves;
  uint256 internal _maxNumberOfNFTVaults;
}
```

NFTVaultLogic.sol

```
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.8.11;

import {NFTVaultConfiguration} from '../configuration/NFTVaultConfiguration.sol';
import {Errors} from '../helpers/Errors.sol';
import {DataTypes} from '../types/DataTypes.sol';

/**
 * @title NFTVaultLogic library
 * @author Vinci
 * @notice Implements the logic to update the vault state
 */
library NFTVaultLogic {
  using NFTVaultLogic for DataTypes.NFTVaultData;
  using NFTVaultConfiguration for DataTypes.NFTVaultConfigurationMap;

  /**
   * @dev Initializes a reserve
   * @param reserve The reserve object
   * @param nTokenAddress The address of the overlying vtoken contract
   **/
  function init(
    DataTypes.NFTVaultData storage reserve,
    address nTokenAddress,
    address nftEligibility
  ) external {
    require(reserve.nTokenAddress == address(0), Errors.NL_VAULT_ALREADY_INITIALIZED);
    reserve.nTokenAddress = nTokenAddress;
    reserve.nftEligibility = nftEligibility;
  }

}
```

LendingPoolConfigurator.sol

```
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.8.11;
pragma abicoder v2;

import {VersionedInitializable} from '../libraries/aave-upgradeability/VersionedInitializable.sol';
import {
  InitializableImmutableAdminUpgradeabilityProxy
} from '../libraries/aave-upgradeability/InitializableImmutableAdminUpgradeabilityProxy.sol';
import {ReserveConfiguration} from '../libraries/configuration/ReserveConfiguration.sol';
import {NFTVaultConfiguration} from '../libraries/configuration/NFTVaultConfiguration.sol';
import {ILendingPoolAddressesProvider} from '../../interfaces/ILendingPoolAddressesProvider.sol';
import {ILendingPool} from '../../interfaces/ILendingPool.sol';
import {IERC20Metadata} from '../../dependencies/openzeppelin/contracts/IERC20Metadata.sol';
import {IERC721} from '../../dependencies/openzeppelin/contracts/IERC721.sol';
import {Errors} from '../libraries/helpers/Errors.sol';
import {PercentageMath} from '../libraries/math/PercentageMath.sol';
import {DataTypes} from '../libraries/types/DataTypes.sol';
import {IInitializableDebtToken} from '../../interfaces/IInitializableDebtToken.sol';
import {IInitializableVToken} from '../../interfaces/IInitializableVToken.sol';
import {IInitializableNToken} from '../../interfaces/IInitializableNToken.sol';
import {IAaveIncentivesController} from '../../interfaces/IAaveIncentivesController.sol';
import {ILendingPoolConfigurator} from '../../interfaces/ILendingPoolConfigurator.sol';

/**
 * @title LendingPoolConfigurator contract
 * @author Aave
 * @dev Implements the configuration methods for the Aave protocol
 **/

contract LendingPoolConfigurator is VersionedInitializable, ILendingPoolConfigurator {
  using PercentageMath for uint256;
  using ReserveConfiguration for DataTypes.ReserveConfigurationMap;
  using NFTVaultConfiguration for DataTypes.NFTVaultConfigurationMap;

  ILendingPoolAddressesProvider internal addressesProvider;
  ILendingPool internal pool;

  modifier onlyPoolAdmin {
    require(addressesProvider.getPoolAdmin() == msg.sender, Errors.CALLER_NOT_POOL_ADMIN);
    _;
  }

  modifier onlyEmergencyAdmin {
    require(
      addressesProvider.getEmergencyAdmin() == msg.sender,
      Errors.LPC_CALLER_NOT_EMERGENCY_ADMIN
    );
    _;
  }

  uint256 internal constant CONFIGURATOR_REVISION = 0x1;

  function getRevision() internal pure override returns (uint256) {
    return CONFIGURATOR_REVISION;
  }

  function initialize(ILendingPoolAddressesProvider provider) public initializer {
    addressesProvider = provider;
    pool = ILendingPool(addressesProvider.getLendingPool());
  }

  /**
   * @dev Initializes reserves in batch
   **/
  function batchInitReserve(InitReserveInput[] calldata input) external onlyPoolAdmin {
    ILendingPool cachedPool = pool;
    for (uint256 i = 0; i < input.length; i++) {
      _initReserve(cachedPool, input[i]);
    }
  }

  function _initReserve(ILendingPool pool, InitReserveInput calldata input) internal {
    address vTokenProxyAddress =
      _initTokenWithProxy(
        input.vTokenImpl,
        abi.encodeWithSelector(
          IInitializableVToken.initialize.selector,
          pool,
          input.treasury,
          input.underlyingAsset,
          IAaveIncentivesController(input.incentivesController),
          input.underlyingAssetDecimals,
          input.vTokenName,
          input.vTokenSymbol,
          input.params
        )
      );

    address stableDebtTokenProxyAddress =
      _initTokenWithProxy(
        input.stableDebtTokenImpl,
        abi.encodeWithSelector(
          IInitializableDebtToken.initialize.selector,
          pool,
          input.underlyingAsset,
          IAaveIncentivesController(input.incentivesController),
          input.underlyingAssetDecimals,
          input.stableDebtTokenName,
          input.stableDebtTokenSymbol,
          input.params
        )
      );

    address variableDebtTokenProxyAddress =
      _initTokenWithProxy(
        input.variableDebtTokenImpl,
        abi.encodeWithSelector(
          IInitializableDebtToken.initialize.selector,
          pool,
          input.underlyingAsset,
          IAaveIncentivesController(input.incentivesController),
          input.underlyingAssetDecimals,
          input.variableDebtTokenName,
          input.variableDebtTokenSymbol,
          input.params
        )
      );

    pool.initReserve(
      input.underlyingAsset,
      vTokenProxyAddress,
      stableDebtTokenProxyAddress,
      variableDebtTokenProxyAddress,
      input.interestRateStrategyAddress
    );

    DataTypes.ReserveConfigurationMap memory currentConfig =
      pool.getConfiguration(input.underlyingAsset);

    currentConfig.setDecimals(input.underlyingAssetDecimals);

    currentConfig.setActive(true);
    currentConfig.setFrozen(false);
    pool.setConfiguration(input.underlyingAsset, currentConfig.data);

    emit ReserveInitialized(
      input.underlyingAsset,
      vTokenProxyAddress,
      stableDebtTokenProxyAddress,
      variableDebtTokenProxyAddress,
      input.interestRateStrategyAddress
    );
  }

   /**
   * @dev Initializes vaults in batch
   **/
  function batchInitNFTVault(InitNFTVaultInput[] calldata input) external onlyPoolAdmin {
    ILendingPool cachedPool = pool;
    for (uint256 i = 0; i < input.length; i++) {
      _initNFTVault(cachedPool, input[i]);
    }
  }

  function _initNFTVault(ILendingPool pool, InitNFTVaultInput calldata input) internal {
    address nTokenProxyAddress =
      _initTokenWithProxy(
        input.nTokenImpl,
        abi.encodeWithSelector(
          IInitializableNToken.initialize.selector,
          pool,
          input.underlyingAsset,
          input.nTokenName,
          input.nTokenSymbol,
          input.params
        )
      );

    pool.initNFTVault(
      input.underlyingAsset,
      nTokenProxyAddress,
      input.nftEligibility
    );

    DataTypes.NFTVaultConfigurationMap memory currentConfig =
      pool.getNFTVaultConfiguration(input.underlyingAsset);

    currentConfig.setActive(true);
    currentConfig.setFrozen(false);

    pool.setNFTVaultConfiguration(input.underlyingAsset, currentConfig.data);

    emit NFTVaultInitialized(
      input.underlyingAsset,
      nTokenProxyAddress
    );
  }

  /**
   * @dev Updates the vToken implementation for the reserve
   **/
  function updateVToken(UpdateVTokenInput calldata input) external onlyPoolAdmin {
    ILendingPool cachedPool = pool;

    DataTypes.ReserveData memory reserveData = cachedPool.getReserveData(input.asset);

    (, , , uint256 decimals, ) = cachedPool.getConfiguration(input.asset).getParamsMemory();

    bytes memory encodedCall = abi.encodeWithSelector(
        IInitializableVToken.initialize.selector,
        cachedPool,
        input.treasury,
        input.asset,
        input.incentivesController,
        decimals,
        input.name,
        input.symbol,
        input.params
      );

    _upgradeTokenImplementation(
      reserveData.vTokenAddress,
      input.implementation,
      encodedCall
    );

    emit VTokenUpgraded(input.asset, reserveData.vTokenAddress, input.implementation);
  }

  /**
   * @dev Updates the stable debt token implementation for the reserve
   **/
  function updateStableDebtToken(UpdateDebtTokenInput calldata input) external onlyPoolAdmin {
    ILendingPool cachedPool = pool;

    DataTypes.ReserveData memory reserveData = cachedPool.getReserveData(input.asset);

    (, , , uint256 decimals, ) = cachedPool.getConfiguration(input.asset).getParamsMemory();

    bytes memory encodedCall = abi.encodeWithSelector(
        IInitializableDebtToken.initialize.selector,
        cachedPool,
        input.asset,
        input.incentivesController,
        decimals,
        input.name,
        input.symbol,
        input.params
      );

    _upgradeTokenImplementation(
      reserveData.stableDebtTokenAddress,
      input.implementation,
      encodedCall
    );

    emit StableDebtTokenUpgraded(
      input.asset,
      reserveData.stableDebtTokenAddress,
      input.implementation
    );
  }

  /**
   * @dev Updates the variable debt token implementation for the asset
   **/
  function updateVariableDebtToken(UpdateDebtTokenInput calldata input)
    external
    onlyPoolAdmin
  {
    ILendingPool cachedPool = pool;

    DataTypes.ReserveData memory reserveData = cachedPool.getReserveData(input.asset);

    (, , , uint256 decimals, ) = cachedPool.getConfiguration(input.asset).getParamsMemory();
    bytes memory encodedCall = abi.encodeWithSelector(
        IInitializableDebtToken.initialize.selector,
        cachedPool,
        input.asset,
        input.incentivesController,
        decimals,
        input.name,
        input.symbol,
        input.params
      );

    _upgradeTokenImplementation(
      reserveData.variableDebtTokenAddress,
      input.implementation,
      encodedCall
    );

    emit VariableDebtTokenUpgraded(
      input.asset,
      reserveData.variableDebtTokenAddress,
      input.implementation
    );
  }

  /**
   * @dev Updates the nToken implementation for the vault
   **/
  function updateNToken(UpdateNTokenInput calldata input) external onlyPoolAdmin {
    ILendingPool cachedPool = pool;

    DataTypes.NFTVaultData memory vaultData = cachedPool.getNFTVaultData(input.asset);

    bytes memory encodedCall = abi.encodeWithSelector(
        IInitializableNToken.initialize.selector,
        cachedPool,
        input.asset,
        input.name,
        input.symbol,
        input.params
      );

    _upgradeTokenImplementation(
      vaultData.nTokenAddress,
      input.implementation,
      encodedCall
    );

    emit NTokenUpgraded(input.asset, vaultData.nTokenAddress, input.implementation);
  }

  /**
   * @dev Enables borrowing on a reserve
   * @param asset The address of the underlying asset of the reserve
   * @param stableBorrowRateEnabled True if stable borrow rate needs to be enabled by default on this
   **/
  function enableBorrowingOnReserve(address asset, bool stableBorrowRateEnabled)
    external
    onlyPoolAdmin
  {
    DataTypes.ReserveConfigurationMap memory currentConfig = pool.getConfiguration(asset);

    currentConfig.setBorrowingEnabled(true);
    currentConfig.setStableRateBorrowingEnabled(stableBorrowRateEnabled);

    pool.setConfiguration(asset, currentConfig.data);
    emit BorrowingEnabledOnReserve(asset, stableBorrowRateEnabled);
  }

  /**
   * @dev Disables borrowing on a reserve
   * @param asset The address of the underlying asset of the reserve
   **/
  function disableBorrowingOnReserve(address asset) external onlyPoolAdmin {
    DataTypes.ReserveConfigurationMap memory currentConfig = pool.getConfiguration(asset);

    currentConfig.setBorrowingEnabled(false);

    pool.setConfiguration(asset, currentConfig.data);
    emit BorrowingDisabledOnReserve(asset);
  }

  /**
   * @dev Configures the nft vault collateralization parameters
   * all the values are expressed in percentages with two decimals of precision. A valid value is 100
   * @param nft The address of the underlying NFT of the vault
   * @param ltv The loan to value of the NFT when used as collateral
   * @param liquidationThreshold The threshold at which loans using this NFT as collateral will be co
   * @param liquidationBonus The bonus liquidators receive to liquidate this NFT. The values is alway
   * means the liquidator will receive a 5% bonus
   **/
  function configureNFTVaultAsCollateral(
    address nft,
    uint256 ltv,
    uint256 liquidationThreshold,
    uint256 liquidationBonus
  ) external onlyPoolAdmin {
    DataTypes.NFTVaultConfigurationMap memory currentConfig = pool.getNFTVaultConfiguration(nft);

    //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
      require(
        liquidationBonus > PercentageMath.PERCENTAGE_FACTOR,
        Errors.LPC_INVALID_CONFIGURATION
      );

      //if threshold * bonus is less than PERCENTAGE_FACTOR, it's guaranteed that at the moment
      //a loan is taken there is enough collateral available to cover the liquidation bonus
      require(
        liquidationThreshold.percentMul(liquidationBonus) <= 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(nft);
    }

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

    pool.setNFTVaultConfiguration(nft, currentConfig.data);
    emit CollateralConfigurationChanged(nft, ltv, liquidationThreshold, liquidationBonus);
  }

  /**
   * @dev Enable stable rate borrowing on a reserve
   * @param asset The address of the underlying asset of the reserve
   **/
  function enableReserveStableRate(address asset) external onlyPoolAdmin {
    DataTypes.ReserveConfigurationMap memory currentConfig = pool.getConfiguration(asset);

    currentConfig.setStableRateBorrowingEnabled(true);

    pool.setConfiguration(asset, currentConfig.data);

    emit StableRateEnabledOnReserve(asset);
  }

  /**
   * @dev Disable stable rate borrowing on a reserve
   * @param asset The address of the underlying asset of the reserve
   **/
  function disableReserveStableRate(address asset) external onlyPoolAdmin {
    DataTypes.ReserveConfigurationMap memory currentConfig = pool.getConfiguration(asset);

    currentConfig.setStableRateBorrowingEnabled(false);

    pool.setConfiguration(asset, currentConfig.data);

    emit StableRateDisabledOnReserve(asset);
  }

  /**
   * @dev Activates a reserve
   * @param asset The address of the underlying asset of the reserve
   **/
  function activateReserve(address asset) external onlyPoolAdmin {
    DataTypes.ReserveConfigurationMap memory currentConfig = pool.getConfiguration(asset);

    currentConfig.setActive(true);

    pool.setConfiguration(asset, currentConfig.data);

    emit ReserveActivated(asset);
  }

  /**
   * @dev Deactivates a reserve
   * @param asset The address of the underlying asset of the reserve
   **/
  function deactivateReserve(address asset) external onlyPoolAdmin {
    _checkNoLiquidity(asset);

    DataTypes.ReserveConfigurationMap memory currentConfig = pool.getConfiguration(asset);

    currentConfig.setActive(false);

    pool.setConfiguration(asset, currentConfig.data);

    emit ReserveDeactivated(asset);
  }

  /**
   * @dev Freezes a reserve. A frozen reserve doesn't allow any new deposit, borrow
   *  but allows repayments, liquidations, and withdrawals
   * @param asset The address of the underlying asset of the reserve
```

```
 **/
function freezeReserve(address asset) external onlyPoolAdmin {
  DataTypes.ReserveConfigurationMap memory currentConfig = pool.getConfiguration(asset);

  currentConfig.setFrozen(true);

  pool.setConfiguration(asset, currentConfig.data);

  emit ReserveFrozen(asset);
}

/**
 * @dev Unfreezes a reserve
 * @param asset The address of the underlying asset of the reserve
 **/
function unfreezeReserve(address asset) external onlyPoolAdmin {
  DataTypes.ReserveConfigurationMap memory currentConfig = pool.getConfiguration(asset);

  currentConfig.setFrozen(false);

  pool.setConfiguration(asset, currentConfig.data);

  emit ReserveUnfrozen(asset);
}

/**
 * @dev Updates the reserve factor of a reserve
 * @param asset The address of the underlying asset of the reserve
 * @param reserveFactor The new reserve factor of the reserve
 **/
function setReserveFactor(address asset, uint256 reserveFactor) external onlyPoolAdmin {
  DataTypes.ReserveConfigurationMap memory currentConfig = pool.getConfiguration(asset);

  currentConfig.setReserveFactor(reserveFactor);

  pool.setConfiguration(asset, currentConfig.data);

  emit ReserveFactorChanged(asset, reserveFactor);
}

/**
 * @dev Sets the interest rate strategy of a reserve
 * @param asset The address of the underlying asset of the reserve
 * @param rateStrategyAddress The new address of the interest strategy contract
 **/
function setReserveInterestRateStrategyAddress(address asset, address rateStrategyAddress)
  external
  onlyPoolAdmin
{
  pool.setReserveInterestRateStrategyAddress(asset, rateStrategyAddress);
  emit ReserveInterestRateStrategyChanged(asset, rateStrategyAddress);
}

/**
 * @dev Activates a vault
 * @param asset The address of the underlying NFT of the vault
 **/
function activateNFTVault(address asset) external onlyPoolAdmin {
  DataTypes.NFTVaultConfigurationMap memory currentConfig = pool.getNFTVaultConfiguration(asset);

  currentConfig.setActive(true);

  pool.setNFTVaultConfiguration(asset, currentConfig.data);

  emit NFTVaultActivated(asset);
}
```

```
/**
 * @dev Deactivates a vault
 * @param asset The address of the underlying NFT of the vault
 **/
function deactivateNFTVault(address asset) external onlyPoolAdmin {
  _checkNFTVaultNoLiquidity(asset);

  DataTypes.NFTVaultConfigurationMap memory currentConfig = pool.getNFTVaultConfiguration(asset);

  currentConfig.setActive(false);

  pool.setNFTVaultConfiguration(asset, currentConfig.data);

  emit NFTVaultDeactivated(asset);
}

/**
 * @dev Freezes a vault. A frozen vault doesn't allow any new deposit,
 *  but allows liquidations and withdrawals
 * @param asset The address of the underlying NFT of the reserve
 **/
function freezeNFTVault(address asset) external onlyPoolAdmin {
  DataTypes.NFTVaultConfigurationMap memory currentConfig = pool.getNFTVaultConfiguration(asset);

  currentConfig.setFrozen(true);

  pool.setNFTVaultConfiguration(asset, currentConfig.data);

  emit NFTVaultFrozen(asset);
}

/**
 * @dev pauses or unpauses all the actions of the protocol, including vToken transfers
 * @param val true if protocol needs to be paused, false otherwise
 **/
function setPoolPause(bool val) external onlyEmergencyAdmin {
  pool.setPause(val);
}

function _initTokenWithProxy(address implementation, bytes memory initParams)
  internal
  returns (address)
{
  InitializableImmutableAdminUpgradeabilityProxy proxy =
    new InitializableImmutableAdminUpgradeabilityProxy(address(this));

  proxy.initialize(implementation, initParams);

  return address(proxy);
}

function _upgradeTokenImplementation(
  address proxyAddress,
  address implementation,
  bytes memory initParams
) internal {
  InitializableImmutableAdminUpgradeabilityProxy proxy =
    InitializableImmutableAdminUpgradeabilityProxy(payable(proxyAddress));

  proxy.upgradeToAndCall(implementation, initParams);
}

function _checkNoLiquidity(address asset) internal view {
  DataTypes.ReserveData memory reserveData = pool.getReserveData(asset);
```

```
    uint256 availableLiquidity = IERC20Metadata(asset).balanceOf(reserveData.vTokenAddress);

    require(
      availableLiquidity == 0 && reserveData.currentLiquidityRate == 0,
      Errors.LPC_RESERVE_LIQUIDITY_NOT_0
    );
  }

  function _checkNFTVaultNoLiquidity(address asset) internal view {
    DataTypes.NFTVaultData memory vaultData = pool.getNFTVaultData(asset);

    uint256 availableLiquidity = IERC721(asset).balanceOf(vaultData.nTokenAddress);

    require(
      availableLiquidity == 0,
      Errors.LPC_RESERVE_LIQUIDITY_NOT_0
    );
  }
}
```

WrappedERC1155.sol

```
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.8.11;

import {Context} from '../../dependencies/openzeppelin/contracts/Context.sol';
import {Address} from '../../dependencies/openzeppelin/contracts/Address.sol';
import {IERC165} from '../../dependencies/openzeppelin/contracts/IERC165.sol';
import {ERC165} from '../../dependencies/openzeppelin/contracts/ERC165.sol';
import {IERC1155} from '../../dependencies/openzeppelin/contracts/IERC1155.sol';
import {IERC1155Metadata} from '../../interfaces/IERC1155Metadata.sol';
import {IERC1155Stat} from '../../interfaces/IERC1155Stat.sol';
import {IERC1155MetadataURI} from '../../dependencies/openzeppelin/contracts/IERC1155MetadataURI.sol'
import {IERC1155Receiver} from '../../dependencies/openzeppelin/contracts/IERC1155Receiver.sol';
import {SafeMath} from '../../dependencies/openzeppelin/contracts/SafeMath.sol';
import {EnumerableSet} from "../../dependencies/openzeppelin/contracts/EnumerableSet.sol";
import {EnumerableMap} from "../../dependencies/openzeppelin/contracts/EnumerableMap.sol";

abstract contract WrappedERC1155 is Context, ERC165, IERC1155, IERC1155Metadata, IERC1155MetadataURI,
    using SafeMath for uint256;
    using Address for address;
    using EnumerableSet for EnumerableSet.UintSet;
    using EnumerableSet for EnumerableSet.AddressSet;

    uint256 internal _totalSupply;
    mapping(uint256 => mapping(address => uint256)) internal _balances;

    mapping(address => uint256) internal _balanceByAccount;

    mapping(address => mapping(address => bool)) private _operatorApprovals;


    mapping(uint256 => uint256) internal _totalSupplyByToken;
    mapping(uint256 => EnumerableSet.AddressSet) _holdersByToken;
    mapping(address => EnumerableSet.UintSet) _tokensByHolder;

    string private _uri;

    string private _name;
    string private _symbol;

    /*constructor(
        string memory uri_,
        string memory name_,
        string memory symbol_
    ) {
        _setURI(uri_);
        _name = name_;
        _symbol = symbol_;
    }*/

    event BurnBatch(address user, address receiverOfUnderlying, uint256[] tokenIds, uint256[] amounts

    /**
    * @return The name of the token
    **/
    function name() public view override returns (string memory) {
        return _name;
    }

    /**
    * @return The symbol of the token
    **/
    function symbol() public view override returns (string memory) {
        return _symbol;
    }

    /**
    * @return The total supply of the token
    **/
    function totalSupply() external virtual view override returns (uint256) {
        return _totalSupplyImpl();
    }

    function totalSupply(uint256 tokenId) external virtual view override returns (uint256) {
        return _totalSupplyImpl(tokenId);
    }

    function totalSupplyBatch(uint256[] calldata tokenIds) external virtual view override returns (ui
        return _totalSupplyBatchImpl(tokenIds);
    }

    function totalHolders(uint256 tokenId) public virtual view override returns (uint256) {
        return _holdersByToken[tokenId].length();
    }

    function holdersByToken(uint256 tokenId) external virtual view override returns (address[] memory
        EnumerableSet.AddressSet storage  holders = _holdersByToken[tokenId];
        address[] memory holderAddresses = new address[](holders.length());

        for(uint256 i; i < holders.length(); ++i) {
            holderAddresses[i] = holders.at(i);
        }

        return holderAddresses;

    }

    function tokensByAccount(address account) external virtual view override returns (uint256[] memor
         EnumerableSet.UintSet storage tokens = _tokensByHolder[account];

        uint256[] memory tokenIds = new uint256[](tokens.length());

        for (uint256 i; i < tokens.length(); ++i) {
            tokenIds[i] = tokens.at(i);
        }
        return tokenIds;
    }
    /**
     * @dev See {IERC165-supportsInterface}.
     */
    function supportsInterface(bytes4 interfaceId) public view virtual override(ERC165, IERC165) retu
        return
            interfaceId == type(IERC1155).interfaceId ||
            interfaceId == type(IERC1155MetadataURI).interfaceId ||
            super.supportsInterface(interfaceId);
    }

    /**
     * @dev See {IERC1155MetadataURI-uri}.
     *
     * This implementation returns the same URI for *all* token types. It relies
     * on the token type ID substitution mechanism
     * https://eips.ethereum.org/EIPS/eip-1155#metadata[defined in the EIP].
     *
     * Clients calling this function must replace the `\{id\}` substring with the
     * actual token type ID.
     */
    function uri(uint256) public view virtual override returns (string memory) {
        return _uri;
    }

    /**
     * @dev See {IERC1155-balanceOf}.
     *
     * Requirements:
     *
     * - `account` cannot be the zero address.
     */
    function balanceOf(address account, uint256 id) external view virtual override returns (uint256)
        return _balanceOf(account, id);
    }

    /**
     * @dev See {IERC1155-balanceOfBatch}.
     *
     * Requirements:
     *
     * - `accounts` and `ids` must have the same length.
     */
    function balanceOfBatch(address[] memory accounts, uint256[] memory ids)
        external
        view
        virtual
        override
        returns (uint256[] memory)
    {
        return _balanceOfBatch(accounts, ids);
    }

    function balanceOfBatch(address account, uint256[] memory ids)
        external
        view
        virtual
        override
        returns (uint256[] memory)
    {
        return _balanceOfBatch(account, ids);
    }

    function balanceOf(address account) external view virtual override returns (uint256) {
        return _balanceOf(account);
    }
```

```
function getUserBalanceAndSupply(address account)
    external
    view
    override
    returns (uint256, uint256)
{
    return (_balanceOf(account), _totalSupplyImpl());
}

/**
 * @dev See {IERC1155-setApprovalForAll}.
 */
function setApprovalForAll(address operator, bool approved) public virtual override {
    _setApprovalForAll(_msgSender(), operator, approved);
}

/**
 * @dev See {IERC1155-isApprovedForAll}.
 */
function isApprovedForAll(address account, address operator) public view virtual override returns
    return _operatorApprovals[account][operator];
}

/**
 * @dev See {IERC1155-safeTransferFrom}.
 */
function safeTransferFrom(
    address from,
    address to,
    uint256 id,
    uint256 amount,
    bytes memory data
) public virtual override {
    require(
        from == _msgSender() || isApprovedForAll(from, _msgSender()),
        "ERC1155: caller is not owner nor approved"
    );
    _safeTransferFrom(from, to, id, amount, data);
    emit TransferSingle(_msgSender(), from, to, id, amount);
}

/**
 * @dev See {IERC1155-safeBatchTransferFrom}.
 */
function safeBatchTransferFrom(
    address from,
    address to,
    uint256[] memory ids,
    uint256[] memory amounts,
    bytes memory data
) public virtual override {
    require(
        from == _msgSender() || isApprovedForAll(from, _msgSender()),
        "ERC1155: transfer caller is not owner nor approved"
    );
    _safeBatchTransferFrom(from, to, ids, amounts, data);
    emit TransferBatch(_msgSender(), from, to, ids, amounts);
}


function _safeTransferFrom(
    address from,
    address to,
    uint256 id,
    uint256 amount,
    bytes memory data
) internal virtual {
    require(to != address(0), "ERC1155: transfer to the zero address");

    address operator = _msgSender();

    _beforeTokenTransfer(operator, from, to, _asSingletonArray(id), _asSingletonArray(amount), da



    uint256 fromBalance = _balances[id][from];
    _balances[id][from] = fromBalance.sub(amount, "ERC1155: insufficient balance for transfer");

    _balanceByAccount[to] =  _balanceByAccount[to] + amount;
    _balances[id][to] =  _balances[id][to] + amount;

    uint256 fromUserBalance = _balanceByAccount[from];
    _balanceByAccount[from] = fromUserBalance.sub(amount, "ERC1155: insufficient balance for tran


    _doSafeTransferAcceptanceCheck(operator, from, to, id, amount, data);
}

function _safeBatchTransferFrom(
    address from,
    address to,
    uint256[] memory ids,
    uint256[] memory amounts,
    bytes memory data
) internal virtual {
    require(ids.length == amounts.length, "ERC1155: ids and amounts length mismatch");
    require(to != address(0), "ERC1155: transfer to the zero address");

    address operator = _msgSender();

    _beforeTokenTransfer(operator, from, to, ids, amounts, data);

    for (uint256 i = 0; i < ids.length; ++i) {
        uint256 id = ids[i];
        uint256 amount = amounts[i];

        uint256 fromBalance = _balances[id][from];
        _balances[id][from] = fromBalance.sub(amount, "ERC1155: insufficient balance for transfer

        _balanceByAccount[to] =   _balanceByAccount[to] + amount;

        _balances[id][to] = _balances[id][to] + amount;

        uint256 fromTotalBalance = _balanceByAccount[from];
        _balanceByAccount[from] = fromTotalBalance.sub(amount, "ERC1155: insufficient balance for

    }

    _doSafeBatchTransferAcceptanceCheck(operator, from, to, ids, amounts, data);
}

function _setURI(string memory newuri) internal virtual {
    _uri = newuri;
}

function _mint(
    address to,
    uint256 id,
    uint256 amount,
    bytes memory data
) internal virtual {
    require(to != address(0), "ERC1155: mint to the zero address");
    address operator = _msgSender();

    _beforeTokenTransfer(operator, address(0), to, _asSingletonArray(id), _asSingletonArray(amoun

    _balanceByAccount[to] = _balanceByAccount[to] + amount;

    uint256 oldAccountBalance = _balances[id][to];
    _balances[id][to] = _balances[id][to] + amount;

    _doSafeTransferAcceptanceCheck(operator, address(0), to, id, amount, data);
}

function _mintBatch(
    address to,
    uint256[] memory ids,
    uint256[] memory amounts,
    bytes memory data
) internal virtual {
    require(to != address(0), "ERC1155: mint to the zero address");
    require(ids.length == amounts.length, "ERC1155: ids and amounts length mismatch");

    address operator = _msgSender();

    _beforeTokenTransfer(operator, address(0), to, ids, amounts, data);

    for (uint256 i = 0; i < ids.length; i++) {
        uint256 id = ids[i];
        uint256 amount = amounts[i];

        _balanceByAccount[to] = _balanceByAccount[to]+ amount;
        _balances[id][to] = _balances[id][to] + amount;
    }

    _doSafeBatchTransferAcceptanceCheck(operator, address(0), to, ids, amounts, data);
}

function _burn(
    address from,
    uint256 id,
    uint256 amount
) internal virtual {
    require(from != address(0), "ERC1155: burn from the zero address");

    address operator = _msgSender();

    _beforeTokenTransfer(operator, from, address(0), _asSingletonArray(id), _asSingletonArray(amo

    uint256 fromBalance = _balances[id][from];
    _balances[id][from] = fromBalance.sub(amount, "ERC1155: burn amount exceeds balance");

    uint256 fromTotalBalance = _balanceByAccount[from];
    _balanceByAccount[from] = fromTotalBalance.sub(amount, "ERC1155: burn amount exceeds balance"

}

function _burnBatch(
    address from,
    uint256[] memory ids,
    uint256[] memory amounts
) internal virtual {
    require(from != address(0), "ERC1155: burn from the zero address");
    require(ids.length == amounts.length, "ERC1155: ids and amounts length mismatch");

    address operator = _msgSender();
    _beforeTokenTransfer(operator, from, address(0), ids, amounts, "");

    for (uint256 i = 0; i < ids.length; i++) {
        uint256 id = ids[i];
        uint256 amount = amounts[i];

        uint256 fromBalance = _balances[id][from];
        _balances[id][from] = fromBalance.sub(amount, "ERC1155: burn amount exceeds balance");

        uint256 fromTotalBalance = _balanceByAccount[from];
        _balanceByAccount[from] = fromTotalBalance.sub(amount, "ERC1155: burn amount exceeds bala
    }
}

function _setApprovalForAll(
    address owner,
    address operator,
    bool approved
) internal virtual {
    require(owner != operator, "ERC1155: setting approval status for self");
    _operatorApprovals[owner][operator] = approved;
    emit ApprovalForAll(owner, operator, approved);
}

function _setName(string memory newName) internal {
    _name = newName;
}

function _setSymbol(string memory newSymbol) internal {
    _symbol = newSymbol;
}

function _beforeTokenTransfer(
    address operator,
    address from,
    address to,
    uint256[] memory tokenIds,
    uint256[] memory amounts,
    bytes memory data
) internal virtual {
    if (from != to) {
        EnumerableSet.UintSet storage fomrTokens = _tokensByHolder[from];
        EnumerableSet.UintSet storage toTokens = _tokensByHolder[to];
        for(uint256 i; i < tokenIds.length; ++i) {
            uint256 amount = amounts[i];
            if (amount > 0) {
                uint256 id = tokenIds[i];
                if (from == address(0)) {
                    _totalSupplyByToken[id] = _totalSupplyByToken[id] + amount;
                    _totalSupply = _totalSupply + amount;

                } else {
                    if (_balanceOf(from, id) == amount) {
                        _holdersByToken[id].remove(from);
                        _tokensByHolder[from].remove(id);
                    }
                }
                if (to == address(0)) {
                    _totalSupplyByToken[id] = _totalSupplyByToken[id] - amount;
                    _totalSupply = _totalSupply + amount;
                } else {
                    if (_balanceOf(to, id) == 0) {
                        _holdersByToken[id].add(to);
                        _tokensByHolder[to].add(id);
                    }
                }
            }
        }
    }
}

function _doSafeTransferAcceptanceCheck(
    address operator,
    address from,
    address to,
    uint256 id,
    uint256 amount,
    bytes memory data
) private {
    if (to.isContract()) {
        try IERC1155Receiver(to).onERC1155Received(operator, from, id, amount, data) returns (byt
            if (response != IERC1155Receiver.onERC1155Received.selector) {
                revert("ERC1155: ERC1155Receiver rejected tokens");
            }
        } catch Error(string memory reason) {
            revert(reason);
        } catch {
            revert("ERC1155: transfer to non ERC1155Receiver implementer");
        }
    }
}

function _doSafeBatchTransferAcceptanceCheck(
    address operator,
    address from,
    address to,
    uint256[] memory ids,
    uint256[] memory amounts,
    bytes memory data
) private {
    if (to.isContract()) {
        try IERC1155Receiver(to).onERC1155BatchReceived(operator, from, ids, amounts, data) retur
            bytes4 response
        ) {
            if (response != IERC1155Receiver.onERC1155BatchReceived.selector) {
                revert("ERC1155: ERC1155Receiver rejected tokens");
            }
        } catch Error(string memory reason) {
            revert(reason);
        } catch {
            revert("ERC1155: transfer to non ERC1155Receiver implementer");
        }
    }
}

function _asSingletonArray(uint256 element) private pure returns (uint256[] memory) {
    uint256[] memory array = new uint256[](1);
    array[0] = element;

    return array;
}

function _balanceOf(address account, uint256 id) internal view virtual returns (uint256) {
    require(account != address(0), "ERC1155: balance query for the zero address");
    return _balances[id][account];
}

/**
 * @dev See {IERC1155-balanceOfBatch}.
 *
 * Requirements:
 *
```

```
      * - `accounts` and `ids` must have the same length.
      */
     function _balanceOfBatch(address[] memory accounts, uint256[] memory ids)
         internal
         view
         virtual
         returns (uint256[] memory)
     {
         require(accounts.length == ids.length, "ERC1155: accounts and ids length mismatch");

         uint256[] memory batchBalances = new uint256[](accounts.length);

         for (uint256 i = 0; i < accounts.length; ++i) {
             batchBalances[i] = _balanceOf(accounts[i], ids[i]);
         }

         return batchBalances;
     }

     function _balanceOfBatch(address account, uint256[] memory ids)
         internal
         view
         virtual
         returns (uint256[] memory)
     {
         require(account != address(0), "ERC1155: balance query for the zero address");
         uint256[] memory batchBalances = new uint256[](ids.length);

         for (uint256 i = 0; i < ids.length; ++i) {
             batchBalances[i] = _balances[ids[i]][account];
         }

         return batchBalances;
     }

     function _balanceOf(address account) internal view virtual returns (uint256) {
         require(account != address(0), "ERC1155: balance query for the zero address");
         return _balanceByAccount[account];
     }

     /**
     * @return The total supply of the token
     **/
     function _totalSupplyImpl() internal virtual view returns (uint256) {
         return _totalSupply;
     }

     function _totalSupplyImpl(uint256 tokenId) internal virtual view returns (uint256) {
         return _totalSupplyByToken[tokenId];
     }

     function _totalSupplyBatchImpl(uint256[] calldata tokenIds) internal virtual view returns (uint25
         uint256[] memory batchTotalSupply = new uint256[](tokenIds.length);
         for(uint256 i = 0; i < tokenIds.length; ++i){
             batchTotalSupply[i] = _totalSupplyByToken[tokenIds[i]];
         }
         return batchTotalSupply;
     }

 }
```

ValidationLogic.sol

```
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.8.11;
pragma abicoder v2;

import {IERC20} from '../../../dependencies/openzeppelin/contracts/IERC20.sol';
import {ReserveLogic} from './ReserveLogic.sol';
import {NFTVaultLogic} from './NFTVaultLogic.sol';
import {GenericLogic} from './GenericLogic.sol';
import {WadRayMath} from '../math/WadRayMath.sol';
import {PercentageMath} from '../math/PercentageMath.sol';
import {GPv2SafeERC20} from '../../../dependencies/gnosis/contracts/GPv2SafeERC20.sol';
import {NFTVaultConfiguration} from '../configuration/NFTVaultConfiguration.sol';
import {ReserveConfiguration} from '../configuration/ReserveConfiguration.sol';
import {UserConfiguration} from '../configuration/UserConfiguration.sol';
import {Errors} from '../helpers/Errors.sol';
import {Helpers} from '../helpers/Helpers.sol';
import {IReserveInterestRateStrategy} from '../../../interfaces/IReserveInterestRateStrategy.sol';
import {INFTXEligibility} from '../../../interfaces/INFTXEligibility.sol';
import {DataTypes} from '../types/DataTypes.sol';

/**
 * @title ReserveLogic library
 * @author Aave
 * @notice Implements functions to validate the different actions of the protocol
 */
library ValidationLogic {
  using ReserveLogic for DataTypes.ReserveData;
  using NFTVaultLogic for DataTypes.NFTVaultData;
  using WadRayMath for uint256;
  using PercentageMath for uint256;
  using GPv2SafeERC20 for IERC20;
  using ReserveConfiguration for DataTypes.ReserveConfigurationMap;
  using NFTVaultConfiguration for DataTypes.NFTVaultConfigurationMap;
  using UserConfiguration for DataTypes.UserConfigurationMap;

  uint256 public constant REBALANCE_UP_LIQUIDITY_RATE_THRESHOLD = 4000;
  uint256 public constant REBALANCE_UP_USAGE_RATIO_THRESHOLD = 0.95 * 1e27; //usage ratio of 95%

  /**
   * @dev Validates a deposit action
   * @param reserve The reserve object on which the user is depositing
   * @param amount The amount to be deposited
   */
  function validateDeposit(DataTypes.ReserveData storage reserve, uint256 amount) external view {
    (bool isActive, bool isFrozen, , ) = reserve.configuration.getFlags();

    require(amount != 0, Errors.VL_INVALID_AMOUNT);
    require(isActive, Errors.VL_NO_ACTIVE_RESERVE);
    require(!isFrozen, Errors.VL_RESERVE_FROZEN);
  }

  function validateDepositNFT(DataTypes.NFTVaultData storage vault, uint256[] memory ids, uint256[] m
    (bool isActive, bool isFrozen) = vault.configuration.getFlags();

    require(ids.length != 0, Errors.VL_INVALID_AMOUNT);
    for(uint256 i = 0; i < ids.length; ++i) {
      require(amounts[i] != 0, Errors.VL_INVALID_AMOUNT);
    }
    require(isActive, Errors.VL_NO_ACTIVE_RESERVE);
    require(!isFrozen, Errors.VL_RESERVE_FROZEN);
    INFTXEligibility eligibility = INFTXEligibility(vault.nftEligibility);
    require(eligibility.checkAllEligible(ids), Errors.VL_NFT_INELIGIBLE_TOKEN_ID);
  }

  /**
```

```
   * @dev Validates a withdraw action
   * @param reserveAddress The address of the reserve
   * @param amount The amount to be withdrawn
   * @param userBalance The balance of the user
   * @param reserves The reserves state
   * @param userConfig The user configuration
   * @param oracle The price oracle
   */
  function validateWithdraw(
    address reserveAddress,
    uint256 amount,
    uint256 userBalance,
    DataTypes.PoolReservesData storage reserves,
    DataTypes.UserConfigurationMap storage userConfig,
    address oracle
  ) external view {
    require(amount != 0, Errors.VL_INVALID_AMOUNT);
    require(amount <= userBalance, Errors.VL_NOT_ENOUGH_AVAILABLE_USER_BALANCE);

    (bool isActive, , , ) = reserves.data[reserveAddress].configuration.getFlags();
    require(isActive, Errors.VL_NO_ACTIVE_RESERVE);
  }

  /**
   * @dev Validates a withdraw action
   * @param vaultAddress The address of the vault
   * @param tokenIds The array of token ids of the NFTs to be withdrawn
   * @param amounts The array of amounts of the NFTs to be withdrawn
   * @param userBalances The array of balances of every NFT in `tokenIds` of the user
   * @param reserves The reserves state
   * @param nftVaults The vaults state
   * @param userConfig The user configuration
   * @param oracle The price oracle
   */
  function validateWithdrawNFT(
    address vaultAddress,
    uint256[] calldata tokenIds,
    uint256[] calldata amounts,
    uint256[] calldata userBalances,
    DataTypes.PoolReservesData storage reserves,
    DataTypes.PoolNFTVaultsData storage nftVaults,
    DataTypes.UserConfigurationMap storage userConfig,
    address oracle
  ) external view {
    require(tokenIds.length == amounts.length, Errors.VL_INVALID_AMOUNT);
    uint256 amount;
    for(uint256 i = 0; i < tokenIds.length; ++i) {
      require(amounts[i] != 0, Errors.VL_INVALID_AMOUNT);
      require(amounts[i] <= userBalances[i], Errors.VL_NOT_ENOUGH_AVAILABLE_USER_BALANCE);
      amount = amount + amounts[i];
    }

    (bool isActive, ) = nftVaults.data[vaultAddress].configuration.getFlags();
    require(isActive, Errors.VL_NO_ACTIVE_RESERVE);

    require(
      GenericLogic.balanceDecreaseAllowed(
        vaultAddress,
        msg.sender,
        amount,
        reserves,
        nftVaults,
        userConfig,
        oracle
      ),
      Errors.VL_TRANSFER_NOT_ALLOWED
    );
  }


  struct ValidateBorrowLocalVars {
    uint256 currentLtv;
    uint256 currentLiquidationThreshold;
    uint256 amountOfCollateralNeededETH;
    uint256 userCollateralBalanceETH;
    uint256 userBorrowBalanceETH;
    uint256 availableLiquidity;
    uint256 healthFactor;
    bool isActive;
    bool isFrozen;
    bool borrowingEnabled;
    //bool stableRateBorrowingEnabled;
  }

  /**
   * @dev Validates a borrow action
   * @param asset The address of the asset to borrow
   * @param reserve The reserve state from which the user is borrowing
   * @param userAddress The address of the user
   * @param amount The amount to be borrowed
   * @param amountInETH The amount to be borrowed, in ETH
   * @param interestRateMode The interest rate mode at which the user is borrowing
   * @param maxStableLoanPercent The max amount of the liquidity that can be borrowed at stable rate,
   * @param reserves The reserves state
   * @param nftVaults The vaults state
   * @param userConfig The user configuration
   * @param oracle The price oracle
   */
  function validateBorrow(
    address asset,
    DataTypes.ReserveData storage reserve,
    address userAddress,
    uint256 amount,
    uint256 amountInETH,
    uint256 interestRateMode,
    uint256 maxStableLoanPercent,
    DataTypes.PoolReservesData storage reserves,
    DataTypes.PoolNFTVaultsData storage nftVaults,
    DataTypes.UserConfigurationMap storage userConfig,
    address oracle
  ) external view {
    ValidateBorrowLocalVars memory vars;

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

    require(vars.isActive, Errors.VL_NO_ACTIVE_RESERVE);
    require(!vars.isFrozen, Errors.VL_RESERVE_FROZEN);
    require(amount != 0, Errors.VL_INVALID_AMOUNT);

    require(vars.borrowingEnabled, Errors.VL_BORROWING_NOT_ENABLED);

    //validate interest rate mode
    require(
      uint256(DataTypes.InterestRateMode.VARIABLE) == interestRateMode,
      Errors.VL_INVALID_INTEREST_RATE_MODE_SELECTED
    );

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

    require(vars.userCollateralBalanceETH > 0, Errors.VL_COLLATERAL_BALANCE_IS_0);

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

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

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

  }

  /**
   * @dev Validates a repay action
   * @param reserve The reserve state from which the user is repaying
   * @param amountSent The amount sent for the repayment. Can be an actual value or uint(-1)
   * @param onBehalfOf The address of the user msg.sender is repaying for
   * @param stableDebt The borrow balance of the user
   * @param variableDebt The borrow balance of the user
   */
  function validateRepay(
    DataTypes.ReserveData storage reserve,
    uint256 amountSent,
    DataTypes.InterestRateMode rateMode,
    address onBehalfOf,
    uint256 stableDebt,
    uint256 variableDebt
  ) external view {
    bool isActive = reserve.configuration.getActive();

    require(isActive, Errors.VL_NO_ACTIVE_RESERVE);

    require(amountSent > 0, Errors.VL_INVALID_AMOUNT);

    require(
        (variableDebt > 0 &&
          DataTypes.InterestRateMode(rateMode) == DataTypes.InterestRateMode.VARIABLE),
      Errors.VL_NO_DEBT_OF_SELECTED_TYPE
    );

    require(
      amountSent != type(uint256).max || msg.sender == onBehalfOf,
      Errors.VL_NO_EXPLICIT_AMOUNT_TO_REPAY_ON_BEHALF
    );
  }

  /**
   * @dev Validates the liquidation action
   * @param collateralVault The vault data of the collateral
   * @param principalReserve The reserve data of the principal
   * @param userConfig The user configuration
   * @param userHealthFactor The user's health factor
   * @param userStableDebt Total stable debt balance of the user
   * @param userVariableDebt Total variable debt balance of the user
   **/
  function validateNFTLiquidationCall(
    DataTypes.NFTVaultData storage collateralVault,
    DataTypes.ReserveData storage principalReserve,
    DataTypes.UserConfigurationMap storage userConfig,
    uint256 userHealthFactor,
    uint256 userStableDebt,
    uint256 userVariableDebt
  ) internal view returns (uint256, string memory) {
    if (
      !collateralVault.configuration.getActive() || !principalReserve.configuration.getActive()
    ) {
      return (
        uint256(Errors.CollateralManagerErrors.NO_ACTIVE_RESERVE),
        Errors.VL_NO_ACTIVE_RESERVE
      );
    }

    if (userHealthFactor >= GenericLogic.HEALTH_FACTOR_LIQUIDATION_THRESHOLD) {
      return (
        uint256(Errors.CollateralManagerErrors.HEALTH_FACTOR_ABOVE_THRESHOLD),
        Errors.LPCM_HEALTH_FACTOR_NOT_BELOW_THRESHOLD
      );
    }

    bool isCollateralEnabled =
      collateralVault.configuration.getLiquidationThreshold() > 0 &&
        userConfig.isUsingNFTVaultAsCollateral(collateralVault.id);

    //if collateral isn't enabled as collateral by user, it cannot be liquidated
    if (!isCollateralEnabled) {
      return (
        uint256(Errors.CollateralManagerErrors.COLLATERAL_CANNOT_BE_LIQUIDATED),
        Errors.LPCM_COLLATERAL_CANNOT_BE_LIQUIDATED
      );
    }

    if (userVariableDebt == 0) {
      return (
        uint256(Errors.CollateralManagerErrors.CURRRENCY_NOT_BORROWED),
        Errors.LPCM_SPECIFIED_CURRENCY_NOT_BORROWED_BY_USER
      );
    }

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

  /**
   * @dev Validates a vToken transfer or an nToken transfer
   * @param from The user from which the vTokens/nTokens are being transferred
   * @param reserves The state of all the reserves
   * @param nftVaults The state of all the NFT vaults
   * @param userConfig The state of the user for the specific reserve
   * @param oracle The price oracle
   */
  function validateTransfer(
    address from,
    DataTypes.PoolReservesData storage reserves,
    DataTypes.PoolNFTVaultsData storage nftVaults,
    DataTypes.UserConfigurationMap storage userConfig,
    address oracle
  ) internal view {
    (, , , , uint256 healthFactor) =
      GenericLogic.calculateUserAccountData(
        from,
        reserves,
        nftVaults,
        userConfig,
        oracle
      );

    require(
      healthFactor >= GenericLogic.HEALTH_FACTOR_LIQUIDATION_THRESHOLD,
      Errors.VL_TRANSFER_NOT_ALLOWED
    );
  }
}
```

NFTVaultConfiguration.sol

```
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.8.11;

import {Errors} from '../helpers/Errors.sol';
import {DataTypes} from '../types/DataTypes.sol';

/**
 * @title NFTVaultConfiguration library
 * @author Vinci
 * @notice Implements the bitmap logic to handle the NFT vault configuration
 */
library NFTVaultConfiguration {
  uint256 constant LTV_MASK =                   0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
  uint256 constant LIQUIDATION_THRESHOLD_MASK = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
  uint256 constant LIQUIDATION_BONUS_MASK =     0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
  // uint256 constant DECIMALS_MASK =              0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
  uint256 constant ACTIVE_MASK =                0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEF
  uint256 constant FROZEN_MASK =                0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFDF
  // uint256 constant BORROWING_MASK =             0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
  // uint256 constant STABLE_BORROWING_MASK =      0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
  // uint256 constant RESERVE_FACTOR_MASK =        0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0000

  /// @dev For the LTV, the start bit is 0 (up to 15), hence no bitshifting is needed
  uint256 constant LIQUIDATION_THRESHOLD_START_BIT_POSITION = 16;
  uint256 constant LIQUIDATION_BONUS_START_BIT_POSITION = 32;
  // uint256 constant RESERVE_DECIMALS_START_BIT_POSITION = 48;
  uint256 constant IS_ACTIVE_START_BIT_POSITION = 56;
  uint256 constant IS_FROZEN_START_BIT_POSITION = 57;
  // uint256 constant BORROWING_ENABLED_START_BIT_POSITION = 58;
  // uint256 constant STABLE_BORROWING_ENABLED_START_BIT_POSITION = 59;
  // uint256 constant RESERVE_FACTOR_START_BIT_POSITION = 64;

  uint256 constant MAX_VALID_LTV = 65535;
  uint256 constant MAX_VALID_LIQUIDATION_THRESHOLD = 65535;
  uint256 constant MAX_VALID_LIQUIDATION_BONUS = 65535;
  // uint256 constant MAX_VALID_DECIMALS = 255;
  // uint256 constant MAX_VALID_RESERVE_FACTOR = 65535;

  /**
   * @dev Sets the Loan to Value of the reserve
   * @param self The reserve configuration
   * @param ltv the new ltv
   **/
```

```
function setLtv(DataTypes.NFTVaultConfigurationMap memory self, uint256 ltv) internal pure {
  require(ltv <= MAX_VALID_LTV, Errors.RC_INVALID_LTV);

  self.data = (self.data & LTV_MASK) | ltv;
}

/**
 * @dev Gets the Loan to Value of the reserve
 * @param self The reserve configuration
 * @return The loan to value
 **/
function getLtv(DataTypes.NFTVaultConfigurationMap storage self) internal view returns (uint256) {
  return self.data & ~LTV_MASK;
}

/**
 * @dev Sets the liquidation threshold of the reserve
 * @param self The reserve configuration
 * @param threshold The new liquidation threshold
 **/
function setLiquidationThreshold(DataTypes.NFTVaultConfigurationMap memory self, uint256 threshold)
  internal
  pure
{
  require(threshold <= MAX_VALID_LIQUIDATION_THRESHOLD, Errors.RC_INVALID_LIQ_THRESHOLD);

  self.data =
    (self.data & LIQUIDATION_THRESHOLD_MASK) |
    (threshold << LIQUIDATION_THRESHOLD_START_BIT_POSITION);
}

/**
 * @dev Gets the liquidation threshold of the reserve
 * @param self The reserve configuration
 * @return The liquidation threshold
 **/
function getLiquidationThreshold(DataTypes.NFTVaultConfigurationMap storage self)
  internal
  view
  returns (uint256)
{
  return (self.data & ~LIQUIDATION_THRESHOLD_MASK) >> LIQUIDATION_THRESHOLD_START_BIT_POSITION;
}

/**
 * @dev Sets the liquidation bonus of the reserve
 * @param self The reserve configuration
 * @param bonus The new liquidation bonus
 **/
function setLiquidationBonus(DataTypes.NFTVaultConfigurationMap memory self, uint256 bonus)
  internal
  pure
{
  require(bonus <= MAX_VALID_LIQUIDATION_BONUS, Errors.RC_INVALID_LIQ_BONUS);

  self.data =
    (self.data & LIQUIDATION_BONUS_MASK) |
    (bonus << LIQUIDATION_BONUS_START_BIT_POSITION);
}

/**
 * @dev Gets the liquidation bonus of the reserve
 * @param self The reserve configuration
 * @return The liquidation bonus
 **/
function getLiquidationBonus(DataTypes.NFTVaultConfigurationMap storage self)
```

```
    internal
    view
    returns (uint256)
  {
    return (self.data & ~LIQUIDATION_BONUS_MASK) >> LIQUIDATION_BONUS_START_BIT_POSITION;
  }

  /**
   * @dev Sets the active state of the reserve
   * @param self The reserve configuration
   * @param active The active state
   **/
  function setActive(DataTypes.NFTVaultConfigurationMap memory self, bool active) internal pure {
    self.data =
      (self.data & ACTIVE_MASK) |
      (uint256(active ? 1 : 0) << IS_ACTIVE_START_BIT_POSITION);
  }

  /**
   * @dev Gets the active state of the reserve
   * @param self The reserve configuration
   * @return The active state
   **/
  function getActive(DataTypes.NFTVaultConfigurationMap storage self) internal view returns (bool) {
    return (self.data & ~ACTIVE_MASK) != 0;
  }

  /**
   * @dev Sets the frozen state of the reserve
   * @param self The reserve configuration
   * @param frozen The frozen state
   **/
  function setFrozen(DataTypes.NFTVaultConfigurationMap memory self, bool frozen) internal pure {
    self.data =
      (self.data & FROZEN_MASK) |
      (uint256(frozen ? 1 : 0) << IS_FROZEN_START_BIT_POSITION);
  }

  /**
   * @dev Gets the frozen state of the reserve
   * @param self The reserve configuration
   * @return The frozen state
   **/
  function getFrozen(DataTypes.NFTVaultConfigurationMap storage self) internal view returns (bool) {
    return (self.data & ~FROZEN_MASK) != 0;
  }

   /**
   * @dev Gets the configuration flags of the reserve
   * @param self The reserve configuration
   * @return The state flags representing active, frozen, borrowing enabled, stableRateBorrowing enab
   **/
  function getFlags(DataTypes.NFTVaultConfigurationMap storage self)
    internal
    view
    returns (
      bool,
      bool
    )
  {
    uint256 dataLocal = self.data;

    return (
      (dataLocal & ~ACTIVE_MASK) != 0,
      (dataLocal & ~FROZEN_MASK) != 0
    );
  }

  /**
   * @dev Gets the configuration paramters of the reserve
   * @param self The reserve configuration
   * @return The state params representing ltv, liquidation threshold, liquidation bonus, the reserve
   **/
  function getParams(DataTypes.NFTVaultConfigurationMap storage self)
    internal
    view
    returns (
      uint256,
      uint256,
      uint256
    )
  {
    uint256 dataLocal = self.data;

    return (
      dataLocal & ~LTV_MASK,
      (dataLocal & ~LIQUIDATION_THRESHOLD_MASK) >> LIQUIDATION_THRESHOLD_START_BIT_POSITION,
      (dataLocal & ~LIQUIDATION_BONUS_MASK) >> LIQUIDATION_BONUS_START_BIT_POSITION
    );
  }

  /**
   * @dev Gets the configuration paramters of the reserve from a memory object
   * @param self The reserve configuration
   * @return The state params representing ltv, liquidation threshold, liquidation bonus, the reserve
   **/
  function getParamsMemory(DataTypes.NFTVaultConfigurationMap memory self)
    internal
    pure
    returns (
      uint256,
      uint256,
      uint256
    )
  {
    return (
      self.data & ~LTV_MASK,
      (self.data & ~LIQUIDATION_THRESHOLD_MASK) >> LIQUIDATION_THRESHOLD_START_BIT_POSITION,
      (self.data & ~LIQUIDATION_BONUS_MASK) >> LIQUIDATION_BONUS_START_BIT_POSITION
    );
  }

  /**
   * @dev Gets the configuration flags of the reserve from a memory object
   * @param self The reserve configuration
   * @return The state flags representing active, frozen, borrowing enabled, stableRateBorrowing enab
   **/
  function getFlagsMemory(DataTypes.NFTVaultConfigurationMap memory self)
    internal
    pure
    returns (
      bool,
      bool
    )
  {
    return (
      (self.data & ~ACTIVE_MASK) != 0,
      (self.data & ~FROZEN_MASK) != 0
    );
  }
}
```

Analysis of audit results

Re-Entrancy

- **Description:**
  One of the features of smart contracts is the ability to call and utilise code of other external contracts. Contracts also typically handle Blockchain Currency, and as such often send Blockchain Currency to various external user addresses. The operation of calling external contracts, or sending Blockchain Currency to an address, requires the contract to submit an external call. These external calls can be hijacked by attackers whereby they force the contract to execute further code (i.e. through a fallback function) , including calls back into itself. Thus the code execution "re-enters" the contract. Attacks of this kind were used in the infamous DAO hack.
- **Detection results:**
  ```
  PASSED!
  ```
- **Security suggestion:**
  no.

Arithmetic Over/Under Flows

- **Description:**
  The Virtual Machine (EVM) specifies fixed-size data types for integers. This means that an integer variable, only has a certain range of numbers it can represent. A uint8 for example, can only store numbers in the range [0,255]. Trying to store 256 into a uint8 will result in 0. If care is not taken, variables in Solidity can be exploited if user input is unchecked and calculations are performed which result in numbers that lie outside the range of the data type that stores them.
- **Detection results:**
  ```
  PASSED!
  ```
- **Security suggestion:**
  no.

Unexpected Blockchain Currency

- **Description:**
  Typically when Blockchain Currency is sent to a contract, it must execute either the fallback function, or another function described in the contract. There are two exceptions to this, where Blockchain Currency can exist in a contract without having executed any code. Contracts which rely on code execution for every Blockchain Currency sent to the contract can be vulnerable to attacks where Blockchain Currency is forcibly sent to a contract.
- **Detection results:**
  ```
  PASSED!
  ```
- **Security suggestion:** no.

Delegatecall

- **Description:**
The CALL and DELEGATECALL opcodes are useful in allowing developers to modularise their code. Standard external message calls to contracts are handled by the CALL opcode whereby code is run in the context of the external contract/function. The DELEGATECALL opcode is identical to the standard message call, except that the code executed at the targeted address is run in the context of the calling contract along with the fact that msg.sender and msg.value remain unchanged. This feature enables the implementation of libraries whereby developers can create reusable code for future contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:** no.

Default Visibilities

- **Description:**
Functions in Solidity have visibility specifiers which dictate how functions are allowed to be called. The visibility determines whBlockchain Currency a function can be called externally by users, by other derived contracts, only internally or only externally. There are four visibility specifiers, which are described in detail in the Solidity Docs. Functions default to public allowing users to call them externally. Incorrect use of visibility specifers can lead to some devestating vulernabilities in smart contracts as will be discussed in this section.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
no.

Entropy Illusion

- **Description:**
All transactions on the blockchain are deterministic state transition operations. Meaning that every transaction modifies the global state of the ecosystem and it does so in a calculable way with no uncertainty. This ultimately means that inside the blockchain ecosystem there is no source of entropy or randomness. There is no rand() function in Solidity. Achieving decentralised entropy (randomness) is a well established problem and many ideas have been proposed to address this (see for example, RandDAO or using a chain of Hashes as described by Vitalik in this post).
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
no.

External Contract Referencing

- **Description:**
One of the benefits of the global computer is the ability to re-use code and interact with contracts already deployed on the network. As a result, a large number of contracts reference external contracts and in general

operation use external message calls to interact with these contracts. These external message calls can mask malicious actors intentions in some non-obvious ways, which we will discuss.

- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unsolved TODO comments

- **Description:**
  Check for Unsolved TODO comments
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Short Address/Parameter Attack

- **Description:**
  This attack is not specifically performed on Solidity contracts themselves but on third party applications that may interact with them. I add this attack for completeness and to be aware of how parameters can be manipulated in contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unchecked CALL Return Values

- **Description:**
  There a number of ways of performing external calls in solidity. Sending Blockchain Currency to external accounts is commonly performed via the transfer() method. However, the send() function can also be used and, for more versatile external calls, the CALL opcode can be directly employed in solidity. The call() and send() functions return a boolean indicating if the call succeeded or failed. Thus these functions have a simple caveat, in that the transaction that executes these functions will not revert if the external call (intialised by call() or send()) fails, rather the call() or send() will simply return false. A common pitfall arises when the return value is not checked, rather the developer expects a revert to occur.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Race Conditions / Front Running

- **Description:**
The combination of external calls to other contracts and the multi-user nature of the underlying blockchain gives rise to a variety of potential Solidity pitfalls whereby users race code execution to obtain unexpected states. Re-Entrancy is one example of such a race condition. In this section we will talk more generally about different kinds of race conditions that can occur on the blockchain. There is a variety of good posts on this subject, a few are: Wiki - Safety, DASP - Front-Running and the Consensus - Smart Contract Best Practices.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
no.

Denial Of Service (DOS)

- **Description:**
This category is very broad, but fundamentally consists of attacks where users can leave the contract inoperable for a small period of time, or in some cases, permanently. This can trap Blockchain Currency in these contracts forever, as was the case with the Second Parity MultiSig hack
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
no.

Block Timestamp Manipulation

- **Description:**
Block timestamps have historically been used for a variety of applications, such as entropy for random numbers (see the Entropy Illusion section for further details), locking funds for periods of time and various state-changing conditional statements that are time-dependent. Miner's have the ability to adjust timestamps slightly which can prove to be quite dangerous if block timestamps are used incorrectly in smart contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
no.

Constructors with Care

- **Description:**
Constructors are special functions which often perform critical, privileged tasks when initialising contracts. Before solidity v0.4.22 constructors were defined as functions that had the same name as the contract that contained them. Thus, when a contract name gets changed in development, if the constructor name isn't changed, it becomes a normal, callable function. As you can imagine, this can (and has) lead to some interesting contract hacks.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unintialised Storage Pointers

- **Description:**
  The EVM stores data either as storage or as memory. Understanding exactly how this is done and the default types for local variables of functions is highly recommended when developing contracts. This is because it is possible to produce vulnerable contracts by inappropriately intialising variables.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Floating Points and Numerical Precision

- **Description:**
  As of this writing (Solidity v0.4.24), fixed point or floating point numbers are not supported. This means that floating point representations must be made with the integer types in Solidity. This can lead to errors/vulnerabilities if not implemented correctly.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

tx.origin Authentication

- **Description:**
  Solidity has a global variable, tx.origin which traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authentication in smart contracts leaves the contract vulnerable to a phishing-like attack.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Permission restrictions

- **Description:**
  Contract managers who can control liquidity or pledge pools, etc., or impose unreasonable restrictions on other users.

- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

armors.io

contact@armors.io