ChainAudit

SMART CONTRACT AUDIT REPORT
for
Hadouken Lending Protocol

Prepared By : Manmeet Singh Parmar
ChainAudit
25 Nov 2022

Smart Contract Final Audit Report for Hadouken Lending Contract

Document Properties

Client	Hadouken Finance
Title	Smart Contract Audit Report
Version (code freeze hash)	e9e68dd0aedac562291b60f3d066c164f1351b02
Author	Manmeet Singh Parmar
Auditor	Manmeet Singh Parmar

Overview

The audited commit is `e9e68dd0aedac562291b60f3d066c164f1351b02` and all Solidity contract in the `lending-contracts/contracts` folder were in scope. Before moving to the full list of issues found in the project, some introductory remarks about the project's current status are in order

Project Status

Hadouken Finance is a decentralized finance platform that allows users to borrow, lend, swap, and bridge all in the same place, utilizing the near-instant transaction times and near-zero fees that have been made possible by the Nervos Godwoken network. Hadouken Lending is a fork of Aave v2. `61c2273a992f655c6d3e7d716a0c2f1b97a55a92` is the commit hash that served as a base when contracts were copied to the Hadouken repository. Hadouken protocol also introduced deposit and borrow cap for total market deposits.

Oracles

DIA

Hadouken Finance has officially integrated DIA's transparent price oracles to support their lending protocol on the Nervos Network. DIA has deployed a dedicated smart contract oracle that provides on-chain price feeds to power Hadouken's lending platform. The oracle supports feeds for asset pairs CKB/USD, WBTC/USD, ETH/USD, USDC/USD, USDT/USD, and BNB/USD. To calculate the price of the assets, the oracle employs a Volume Weighted Average Price (VWAP) methodology, based on a 15-minute time interval.

BAND's Oracle

BAND is a decentralized cross-chain data-oracle network. BAND's oracle is being used to power the lending component of the Hadouken. The Hadouken team is currently working to allow crowdsourced liquidations on their dApp and Band Protocol's integration is used for this.

B. Protocol's B. AMM integration over Hadouken

B-Protocol, a decentralized backstop liquidity protocol, where backstop liquidity providers (BLP) buy their right to liquidate under-collateralized loans and share their profits with the users of the platform. As a result, the users (borrowers and lenders) receive additional yield to their usual interest rate. The proposed mechanism eliminates the need for gas wars between liquidators, and thus transfers a big part of the protocol value back to the borrowers and lenders, which in turn improves their effective interest rate. B. AMM is originally designed for the compound's cToken, for Aave's aToken compatibility, an `AaveToCTokenAdapter` smart contract is implemented which wraps the aToken functionality with the needed cToken functionality.

Audit Update

The Hadouken team applied several fixes and acknowledge few isses based on the recommendations. The audit fix is `d21b5be0e9a52a64513ef5d1afa500095eee0358`. Addressed below are the fixes introduced as part of this first audit. Note that some of the original issues reported have been identified as non-issues, but are still kept for completeness. Some final remarks after reviewing the fixes:

- Issues classified as High [H01]]priority is fixed, issues classified as Warning [W01] and one of the Comments issue [Co01] is acknowledged and other Comments issue - [Co02] and [Co03] are fixed.
- Before the code is open-sourced and the project launched, I encourage the Hadouken team to add a "Security" section in the main README of the project, including instructions for the responsible disclosure of any security vulnerabilities found in the project.

Security Assessment

Classification of issues

1. CRITICAL: Bugs leading to Ether or token theft, fund access locking, or any other loss of Ether/tokens to be transferred to any party (for example, dividends)

2.
3. HIGH: Bugs that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement

4. WARNINGS: Bugs that can break the intended contract logic or expose it to DoS attacks.

5. COMMENTS: Other issues and recommendations

Security Assessment Methodology

Engagement Goals

Specifically, I sought to answer the following questions:

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

Stages of audit

- Structural Analysis
- Static Analysis
- Code Review / Manual Analysis

Detected Issues

CRITICAL

Not Found

HIGH

[H01] Borrow SafeMath - Subtraction Overflow

If someone tries to make a borrow that is bigger than the available liquidity, `DefaultReserveInterestRateStrategy#calculateInterestRates` will throw a SafeMath exception, in this case, the `availableLiquidity < liquidityTaken`

The error can be reproduced with the following scenario:

```
{
  "title": "LendingPool: Borrow that is bigger than the available liquidity
(reverts)",
  "description": "Test cases for the borrow function.",
  "stories": [
    {
      "description": "User 0 deposits 1000 DAI, user 1 deposits 1 WETH as
collateral and tries to borrow 10000 DAI (revert expected)",
      "actions": [
        {
          "name": "mint",
          "args": {
            "reserve": "DAI",
            "amount": "1000",
            "user": "0"
          },
          "expected": "success"
        },
        {
          "name": "approve",
          "args": {
            "reserve": "DAI",
            "user": "0"
          },
          "expected": "success"
        },
        {
          "name": "deposit",
          "args": {
            "reserve": "DAI",
            "amount": "1000",
            "user": "0"
          },
          "expected": "success"
        },
        {
          "name": "mint",
          "args": {
            "reserve": "WETH",
```

```
            "amount": "1",
            "user": "1"
          },
          "expected": "success"
        },
        {
          "name": "approve",
          "args": {
            "reserve": "WETH",
            "user": "1"
          },
          "expected": "success"
        },
        {
          "name": "deposit",
          "args": {
            "reserve": "WETH",
            "amount": "1",
            "user": "1"
          },
          "expected": "success"
        },
        {
          "name": "borrow",
          "args": {
            "reserve": "DAI",
            "amount": "10000",
            "borrowRateMode": "variable",
            "user": "1"
          },
          "expected": "revert",
          "revertMessage": "Current available liquidity not enough"
        }
      ]
    }
  ]
}
```

Fixes and Recommendation :

This can be fixed by this adding this code to `ValidationLogic#validateBorrow` :

```
vars.availableLiquidity =
IERC20(asset).balanceOf(reserve.aTokenAddress);
require(
          amount <= vars.availableLiquidity,
          Errors.VL_CURRENT_AVAILABLE_LIQUIDITY_NOT_ENOUGH
        );
```

Note :
#1 This can be handled inside the interest rate strategy `DefaultReserveInterestRateStrategy#calculateInterestRates` method where availableLiquidity and liquidityTaken are available to compare but to maintain the single responsibility principle used by ValidationLogic this check should be moved to `ValidationLogic#validateBorrow`

#2 Keep in mind that this method is called from `LendingPool#flashLoan` , meaning that flashloan was called with an open debt and the underlying asset was already released, so we've to check this, one option could be to use the struct ExecuteBorrowParams and read the releaseUnderlying flag, because the floashloan released the underlying asset first, probably the `maxLoanSizeStable` for stable rates is not accurate and also needs to read the `releaseUnderlying` flag.

Update : Fixed, Borrow validation , when out of liquidity is accounted for [#PR85]

WARNING

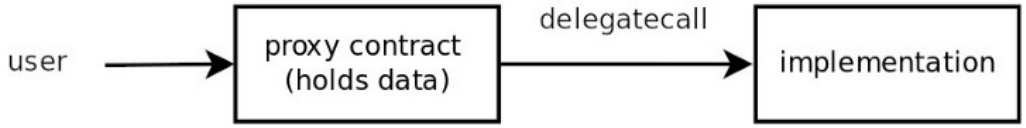
[W01] Unprotected Upgrade - LendingPool

`LendingPool.sol` is an implementation contract that is behind a proxy. It has an initialize function that changes the `_addressesProvider` property. The `initializer modifier` allows calling this function by anyone if it has never been called before.

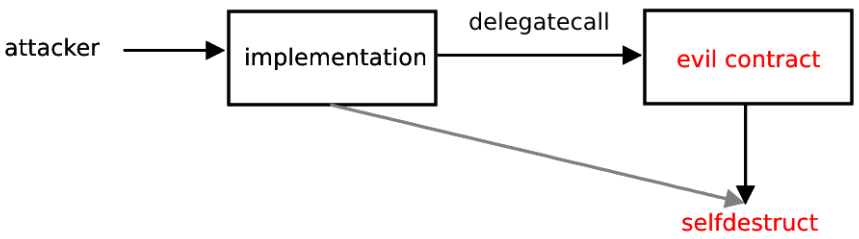
Ref : `LendingPool#initialize`

```
  function initialize(ILendingPoolAddressesProvider provider) public
initializer {
    _addressesProvider = provider;
    _maxStableRateBorrowSizePercent = 2500;
    _flashLoanPremiumTotal = 9;
    _maxNumberOfReserves = 128;
  }
```

Normally, this wouldn't be a problem even if anyone could call initialize on an implementation contract. This is because an implementation contract only provides the necessary logic for the proxy contract to function - The actual data is stored in a proxy contract. Making state changes to the implementation contract has absolutely no effect on the logic.



There is one caveat. If an implementation contract itself uses `delegatecall` and an attacker is able to make it call an attacker's contract, the attacker's contract can then call `selfdestruct` on behalf of the implementation contract, and subsequently destroys the implementation contract.



This is exactly the case for Hadouken, The function uses delegatecall to any attacker-selected address (returned via `_addressesProvider.getLendingPoolCollateralManager)`

`Ref : LendingPool#liquidateCall`

```
  function liquidationCall(
    address collateralAsset,
    address debtAsset,
    address user,
    uint256 debtToCover,
    bool receiveAToken
  ) external override whenNotPaused {
    IBProtocol bProtocol = IBProtocol(_bProtocol[debtAsset]);

    if(bProtocol != IBProtocol(0) && bProtocol.canLiquidate(debtAsset, collateralAsset,
debtToCover)) {
      require(msg.sender == address(bProtocol), "only B.Protocol can liquidate");
    }

    address collateralManager = _addressesProvider.getLendingPoolCollateralManager();

    //solium-disable-next-line
    (bool success, bytes memory result) = collateralManager.delegatecall(
      abi.encodeWithSignature(
        'liquidationCall(address,address,address,uint256,bool)',
        collateralAsset,
        debtAsset,
        user,
        debtToCover,
        receiveAToken
      )
    );

    require(success, Errors.LP_LIQUIDATION_CALL_FAILED);

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

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

Fixes and Recommendations :

1. Make sure the logic / implementation contract is initialized
2. LendingPool contract should be initialized in the deployment script itself
3. If possible add a constructor in all logic contracts to disable the initialize function
4. Check for the existence of a contract in the delegatecall proxy fallback function
5. Carefully review delegatecall pitfalls

Update : Acknowledged , the LendinPool is properly initialized in hardhat deployment scripts

COMMENTS

[Co01] Borrow Validation does not include borrowCap validation

Hadouken protocol introduced deposit and borrow cap for total market's deposits and borrows. The ValidationLogic contract follows single responsibility principle where all deposit, borrow and other related validations are done. `ValidationLogic#validateDeposit` is changes to check for `depositCap` as well (check git-diff below).

```
-  function validateDeposit(DataTypes.ReserveData storage reserve,
uint256 amount) external view {
+  function validateDeposit(DataTypes.ReserveData storage reserve,
uint256 amount, uint256 totalDepositBalance, uint256 depositCap)
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);
+    if (depositCap !=  0) {
+      require(depositCap >= totalDepositBalance + amount,
Errors.DEPOSIT_CAP_REACHED);
+    }
   }
```

But in the borrow validation, borrowCap validation is not accounted for in `ValidationLogic#validateBorrow`

The validation of `borrowCap` is done in `LendingPool#_executeBorrow (L920-933)`

```
    if (borrowCap != 0) {
      uint256 totalVariableDebtTokens =
IVariableDebtToken(reserve.variableDebtTokenAddress).scaledTotalSupply().rayMul(reserve.variableBorr
owIndex);

      (
        ,
        uint256 totalStableDebtTokens,
        ,

      ) = IStableDebtToken(reserve.stableDebtTokenAddress).getSupplyData();

      uint256 totalDebt = totalStableDebtTokens + totalVariableDebtTokens + vars.amount;

      require(totalDebt <= borrowCap, Errors.BORROW_CAP_REACHED);
    }
```

Recommendation:
Move the above logic inside `ValidationLogic#validateBorrow` which is being called right before LendingPool#L920 so that this check is not being called explicitly and is not missed anywhere else.

Update : Acknowledged[Moving the logic as recommended above gives stack too deep, this is a chain limitation]

[Co02] Missing parameter information for modified function

In ValidationLogic#validateDeposit two new parameters are added `totalDepositBalance` and `depositCap` param info in docstring is missing for these two parameters.

```
    * @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 {
+  function validateDeposit(DataTypes.ReserveData storage reserve,
uint256 amount, uint256 totalDepositBalance, uint256 depositCap)
external view {
     (bool isActive, bool isFrozen, , ) =
reserve.configuration.getFlags();
```

Update : Fixed in [#PR85] , related docstring info was added

[Co03] Divide before multiply

`GenericLogic.calculateUserAccountData(address,mapping(address => DataTypes.ReserveData),DataTypes.UserConfigurationMap,mapping(uint256 => address),uint256,address)` (contracts/protocol/libraries/logic/GenericLogic.sol#150-234) performs a multiplication on the result of a division:

- `liquidityBalanceETH = vars.reserveUnitPrice.mul(vars.compoundedLiquidityBalance).div(vars.tokenUnit)` (contracts/protocol/libraries/logic/GenericLogic.sol#192-193

- `vars.avgLtv = vars.avgLtv.add(liquidityBalanceETH.mul(vars.ltv))` (contracts/protocol/libraries/logic/GenericLogic.sol#197)

`GenericLogic.calculateUserAccountData(address,mapping(address => DataTypes.ReserveData),DataTypes.UserConfigurationMap,mapping(uint256 => address),uint256,address)` (contracts/protocol/libraries/logic/GenericLogic.sol#150-234) performs a multiplication on the result of a division:

- `liquidityBalanceETH = vars.reserveUnitPrice.mul(vars.compoundedLiquidityBalance).div(vars.tokenUnit)` (contracts/protocol/libraries/logic/GenericLogic.sol#192-193)
- `vars.avgLiquidationThreshold = vars.avgLiquidationThreshold.add(liquidityBalanceETH.mul(vars.liquidationThreshold))` (contracts/protocol/libraries/logic/GenericLogic.sol#L198-200)

Recommendation:
Consider ordering multiplication before division.
Update : Fixed in [PR#85]

Closing Summary

Findings List

Level	Amount
CRITICAL	0
HIGH	1 - FIXED
WARNING	1 - ACKNOWLEDGED
COMMENTS	3 - FIXED