AstraSec

ioLend

Security Audit Report

October 30, 2024

Contents

1 Introduction

1.1 About ioLend

ioLend is designed to aggregate the most attractive yield opportunities on the network. This is achieved through integrated yield-bearing collateral and automated smart leverage tools built atop decentralized borrowing markets. This allows users to capture multiple yield sources simultaneously while leveraging the returns.

1.2 Source Code

The following source code was reviewed during the audit:

- https://github.com/ioLend-fi/iolend-lending-contracts.git
- CommitID: 1d4b195

And this is the final version representing all fixes implemented for the issues identified in the audit:

- https://github.com/ioLend-fi/iolend-lending-contracts.git
- CommitID: a92ff55

Please note that the BountyManager.sol contract is outside the scope of the audit and will not be utilized in the current implementation. Additionally, we assume the price oracle is robust and reliable, and that asset prices are provided in a timely manner. The implementation of the price oracle itself is not included in the scope of this audit.

2 Overall Assessment

This report has been compiled to identify issues and vulnerabilities within the ioLend protocol. Throughout this audit, we identified a total of 7 issues spanning various severity levels. By employing auxiliary tool techniques to supplement our thorough manual code review, we have discovered the following findings.

Severity	Count	Acknowledged	Won’t Do	Addressed
Critical	–	–	–	–
High	3	–	–	3
Medium	1	1	–	–
Low	3	2	–	1
Informational	–	–	–	–
Undetermined	–	–	–	–

3 Vulnerability Summary

3.1 Overview

Click on an issue to jump to it, or scroll down to see them all.

~~H-1~~ Untrusted Input Leading to Asset Theft via requestFlashLoan()

~~H-2~~ Unauthorized Access in executeOperation() Enabling User Asset Theft

~~H-3~~ Arbitrary Call to Steal Authorized User Assets via flashBorrow()

M-1 Potential Risks Associated with Centralization

L-1 Improper Logic of EligibilityDataProvider::lastEligibleTime()

~~L-2~~ Revisited Slippage Control in LockZap::_zap()

L-3 Suggested _disableInitializers() in constructor()

3.2 Security Level Reference

In web3 smart contract audits, vulnerabilities are typically classified into different severity levels based on the potential impact they can have on the security and functionality of the contract. Here are the definitions for critical-severity, high-severity, medium-severity, and low-severity vulnerabilities:

Severity	Acknowledged
C-X (Critical)	A severe security flaw with immediate and significant negative consequences. It poses high risks, such as unauthorized access, financial losses, or complete disruption of functionality. Requires immediate attention and remediation.
H-X (High)	Significant security issues that can lead to substantial risks. Although not as severe as critical vulnerabilities, they can still result in unauthorized access, manipulation of contract state, or financial losses. Prompt remediation is necessary.
M-X (Medium)	Moderately impactful security weaknesses that require attention and re-mediation. They may lead to limited unauthorized access, minor financial losses, or potential disruptions to functionality.
L-X (Low)	Minor security issues with limited impact. While they may not pose significant risks, it is still recommended to address them to maintain a robust and secure smart contract.
I-X (Informational)	Warnings and things to keep in mind when operating the protocol. No immediate action required.
U-X (Undetermined)	Identified security flaw requiring further investigation. Severity and impact need to be determined. Additional assessment and analysis are necessary.

3.3 Vulnerability Details

3.3.1 [H-1] Untrusted Input Leading to Asset Theft via requestFlashLoan()

Target	Category	IMPACT	LIKELIHOOD	STATUS
BorrowFlashLoanReceiver.sol	Business Logic	High	High	Addressed

The BorrowFlashLoanReceiver::requestFlashLoan() function serves as the entry point for flashloan. However, it contains a vulnerability due to the lack of restrictions on the parameters passed to this function. Specifically, there is no validation on the input params, which allows a malicious actor to arbitrarily construct it.

This oversight exposes users to potential asset theft. A malicious actor can exploit this vulnerability by using the collateral of the params.onBehalfOf to borrow assets (line 86). They can then directly transfer the borrowed assets to their own address, thereby compromising the security of the funds.

Moreover, the RepayFlashLoanReceiver::requestFlashLoan() function shares the same issue.

iolend-lending-contracts - BorrowFlashLoanReceiver.sol

```
function requestFlashLoan(
  address onBehalfOf,
  address borrowAsset,
  uint256 borrowAmount,
  bytes calldata params
) external {
  address[] memory assets = new address[](1);
  assets[0] = borrowAsset;
  uint256[] memory amounts = new uint256[](1);
  amounts[0] = borrowAmount;
  uint256[] memory modes = new uint256[](1);
  modes[0] = 0;
  LENDING_POOL.flashLoan(address(this), assets, amounts, modes, onBehalfOf, params, 0);
}
```

iolend-lending-contracts - BorrowFlashLoanReceiver.sol

```
function executeOperation(
  address[] calldata assets,
  uint256[] calldata amounts,
  uint256[] calldata premiums,
  address,
  bytes calldata params
) external returns (bool) {
  ...
  ExecuteOperationLocalVars memory vars;
  vars.collateralAsset = assets[0];
  vars.flashLoanAmount = amounts[0];
  vars.premiumAmount = premiums[0];
  (
    vars.depositAmount,
    vars.borrowAsset,
    vars.borrowAmount,
    vars.onBehalfOf,
    vars.isFromWallet,
    vars.to,
    vars.data
  ) = abi.decode(params, (uint256, address, uint256, address, bool, address, bytes));
  (vars.maxBorrowAmount, , ) = Calculator.calculateBorrow(
    vars.collateralAsset,
    vars.depositAmount,
    vars.borrowAsset,
    DATA_PROVIDER,
    PRICE_ORACLE
  );
  if (vars.borrowAmount > vars.maxBorrowAmount) revert ExceedsMaxBorrowAmount();
  IERC20(vars.collateralAsset).forceApprove(address(LENDING_POOL), type(uint256).max);

  vars.isFromWallet
    ? LENDING_POOL.deposit(vars.collateralAsset, vars.depositAmount, vars.onBehalfOf, 0)
    : LENDING_POOL.deposit(vars.collateralAsset, vars.flashLoanAmount, vars.onBehalfOf, 0);

  LENDING_POOL.borrow(vars.borrowAsset, vars.borrowAmount, 2, 0, vars.onBehalfOf);
  IERC20(vars.borrowAsset).forceApprove(vars.to, vars.borrowAmount);
  (bool success, bytes memory result) = payable(vars.to).call(vars.data);
  ...
  return true;
}
```

Remediation Apply necessary restrictions on BorrowFlashLoanReceiver/RepayFlashLoanReceiver::requestFlashLoan() to ensure that only the Leverager contract can invoke it.

3.3.2 [H-2] Unauthorized Access in executeOperation() Enabling User Asset Theft

Target	Category	IMPACT	LIKELIHOOD	STATUS
BorrowFlashLoan Receiver.sol	Business Logic	High	High	Addressed

By design, the BorrowFlashLoanReceiver::executeOperation() function serves as the callback for the flashloan operation. However, it currently does not have any access restrictions in place, which allows any user to call it directly. A malicious actor could exploit this weakness by utilizing the collateral of the params.onBehalfOf to borrow assets and subsequently transfer those borrowed assets directly to their own address.

Moreover, the RepayFlashLoanReceiver::executeOperation() function shares the same issue.

iolend-lending-contracts - SwapLoopLeverager.sol

```
function flashBorrow(
  address collateralAsset,
  address borrowAsset,
  uint256 depositTarget,
  uint256 borrowTarget,
  Action[] calldata actions
) external {
  FlashBorrowLocalVars memory vars;
  vars.onBehalfOf = _msgSender();
  IERC20(collateralAsset).forceApprove(address(LENDING_POOL), type(uint256).max);
  uint256 i = 0;
  while (i < actions.length) {
    if (actions[i].code == 1) {
      vars.depositAmount = abi.decode(actions[i].data, (uint256));
      vars.depositTarget += vars.depositAmount;
      if (i == 0) {
        // flash borrow from wallet
        IERC20(collateralAsset).safeTransferFrom(vars.onBehalfOf, address(this), vars.depositAmount);
      }
      LENDING_POOL.deposit(collateralAsset, vars.depositAmount, vars.onBehalfOf, 0);
    } else if (actions[i].code == 2) {
      vars.borrowAmount = abi.decode(actions[i].data, (uint256));
      vars.borrowTarget += vars.borrowAmount;
      LENDING_POOL.borrow(borrowAsset, vars.borrowAmount, 2, 0, vars.onBehalfOf);
    } else if (actions[i].code == 3) {
      vars.swapTo = address(uint160(bytes20(actions[i].data[:20])));
      vars.swapData = actions[i].data[20:];
      IERC20(borrowAsset).forceApprove(vars.swapTo, type(uint256).max);
      (bool success, ) = payable(vars.swapTo).call(vars.swapData);
      if (!success) revert SwapFailed();
    } else {
      revert UnknownAction(actions[i].code);
    }
    i++;
  }
```

iolend-lending-contracts - BorrowFlashLoanReceiver.sol

```
function executeOperation(
  address[] calldata assets,
  uint256[] calldata amounts,
  uint256[] calldata premiums,
  address,
  bytes calldata params
) external returns (bool) {
  ...
  ExecuteOperationLocalVars memory vars;
  vars.collateralAsset = assets[0];
  vars.flashLoanAmount = amounts[0];
  vars.premiumAmount = premiums[0];
  (
    vars.depositAmount,
    vars.borrowAsset,
    vars.borrowAmount,
    vars.onBehalfOf,
    vars.isFromWallet,
    vars.to,
    vars.data
  ) = abi.decode(params, (uint256, address, uint256, address, bool, address, bytes));
  (vars.maxBorrowAmount, , ) = Calculator.calculateBorrow(
    vars.collateralAsset,
    vars.depositAmount,
    vars.borrowAsset,
    DATA_PROVIDER,
    PRICE_ORACLE
  );
  if (vars.borrowAmount > vars.maxBorrowAmount) revert ExceedsMaxBorrowAmount();
  IERC20(vars.collateralAsset).forceApprove(address(LENDING_POOL), type(uint256).max);

  vars.isFromWallet
    ? LENDING_POOL.deposit(vars.collateralAsset, vars.depositAmount, vars.onBehalfOf, 0)
    : LENDING_POOL.deposit(vars.collateralAsset, vars.flashLoanAmount, vars.onBehalfOf, 0);

  LENDING_POOL.borrow(vars.borrowAsset, vars.borrowAmount, 2, 0, vars.onBehalfOf);
  IERC20(vars.borrowAsset).forceApprove(vars.to, vars.borrowAmount);
  (bool success, bytes memory result) = payable(vars.to).call(vars.data);
  ...
  return true;
}
```

Remediation Improve the implementation of the BorrowFlashLoanReceiver/RepayFlashLoanReceiver::executeOperation() functions to validate both the function caller and the flashloan initiator.

3.3.3 [H-3] Arbitrary Call to Steal Authorized User Assets via flashBorrow()

Target	Category	IMPACT	LIKELIHOOD	STATUS
SwapLoopLeverager.sol	Business Logic	High	High	Addressed

The SwapLoopLeverager::flashBorrow() function is designed to execute a series of operations, including deposit, borrow, and swap, in a loop. However, upon reviewing its implementation, we identify a well-known arbitrary call vulnerability. Specifically, a malicious actor can manipulate the vars.swapTo and vars.swapData parameters (line 68), enabling them to execute unauthorized operations and potentially steal user assets that have been approved for this contract. Moreover, the SwapLoopWlpLeverager::flashBorrow() function shares the same issue.

iolend-lending-contracts - SwapLoopLeverager.sol

```
55 function flashBorrow(
56   ...
57 ) external {
58   ...
59   while (i < actions.length) {
60     if (actions[i].code == 1) {
61       ...
62     } else if (actions[i].code == 2) {
63       ...
64     } else if (actions[i].code == 3) {
65       vars.swapTo = address(uint160(bytes20(actions[i].data[:20])));
66       vars.swapData = actions[i].data[20:];
67       IERC20(borrowAsset).forceApprove(vars.swapTo, type(uint256).max);
68       (bool success, ) = payable(vars.swapTo).call(vars.swapData);
69       if (!success) revert SwapFailed();
70     } else {
71       revert UnknownAction(actions[i].code);
72     }
73     i++;
74   }
```

Remediation Whitelist the swapTo address and the function selector of swapData to ensure that only authorized addresses and functions can be executed.

3.3.4 [M-1] Potential Risks Associated with Centralization

Target	Category	IMPACT	LIKELIHOOD	STATUS
Multiple Contracts	Security	High	Low	Acknowledged

In the ioLend protocol, the existence of a privileged owner account introduces centralization risks, as it holds significant control and authority over critical operations governing the protocol. In the following, we show the representative function potentially affected by the privileges associated with the privileged account.

iolend-lending-contracts - AaveOracle.sol

```
/// @notice External function called by the Aave governance to set or replace sources of assets
/// @param assets The addresses of the assets
/// @param sources The address of the source of each asset
function setAssetSources(address[] calldata assets, address[] calldata sources) external onlyOwner {
    _setAssetsSources(assets, sources);
}

/// @notice Sets the fallbackOracle
/// - Callable only by the Aave governance
/// @param fallbackOracle The address of the fallbackOracle
function setFallbackOracle(address fallbackOracle) external onlyOwner {
    _setFallbackOracle(fallbackOracle);
}
```

Remediation To mitigate the identified issue, it is recommended to introduce multi-sig mechanism to undertake the role of the privileged account. Moreover, it is advisable to implement timelocks to govern all modifications to the privileged operations.

Response By Team This issue has been confirmed by the team.

3.3.5 [L-1] Improper Logic of EligibilityDataProvider::lastEligibleTime()

Target	Category	IMPACT	LIKELIHOOD	STATUS
EligibilityDataProvider.sol	Business Logic	Low	Low	Acknowledged

The lastEligibleTime() function is intended to return the last eligible timestamp for a user based on their locked assets. While examining its logic, we notice its current implementation needs to be improved.

The function operates under the assumption that the array of locked balances obtained from the MultiFeeDistribution contract (line 228) is sorted by unlock time. This assumption is crucial for the function to work correctly, as it traverses the locked balances in reverse order, accumulating the total locked value until it meets the required threshold. However, further analysis reveals that the current implementation of the MultiFeeDistribution::_stake() function does not guarantee that the array of locked balances is ordered by unlock time (lines 1089 - 1110). This oversight can result in incorrect eligible timestamp being returned.

iolend-lending-contracts - EligibilityDataProvider.sol

```
222  function lastEligibleTime(address user) public view returns (uint256 lastEligibleTimestamp) {
223      ...
224
225      IMultiFeeDistribution multiFeeDistribution = IMultiFeeDistribution(
226          middleFeeDistribution.getMultiFeeDistributionAddress()
227      );
228      LockedBalance[] memory lpLockData = multiFeeDistribution.lockInfo(user);
229
230      uint256 lockedLP;
231      for (uint256 i = lpLockData.length; i > 0; ) {
232          LockedBalance memory currentLockData = lpLockData[i - 1];
233          lockedLP += currentLockData.amount;
234
235          if (_lockedUsdValue(lockedLP) >= requiredValue) {
236              return currentLockData.unlockTime;
237          }
238          unchecked {
239              i--;
240          }
241      }
242  }
```

iolend-lending-contracts - MultiFeeDistribution.sol

```
function _stake(uint256 amount, address onBehalfOf, uint256 typeIndex, bool isRelock) ... {
  ...
  for (uint256 i; i < userLocksLength; ) {
    if (userLocks[i].unlockTime / AGGREGATION_EPOCH == unlockWeek &&
        userLocks[i].multiplier == rewardMultiplier) {
      _userLocks[onBehalfOf][i].amount = userLocks[i].amount + amount;
      isAggregated = true;
      break;
    }
    unchecked {
      i++;
    }
  }
  if (!isAggregated) {
    _userLocks[onBehalfOf].push(
      LockedBalance({
        amount: amount,
        unlockTime: unlockTime,
        multiplier: rewardMultiplier,
        duration: _lockPeriod[typeIndex]
      })
    );
    emit LockerAdded(onBehalfOf);
  }
  ...
}
```

Remediation Ensure that the array of locked balances (i.e., _userLocks[]) is sorted by unlock time.

3.3.6 [L-2] Revisited Slippage Control in LockZap::_zap()

Target	Category	IMPACT	LIKELIHOOD	STATUS
LockZap.sol	Business Logic	Low	Low	Addressed

The _zap() function is intended to deposit IOL and WETH into the multipool, thereby providing liquidity and minting LP tokens, which are subsequently staked in the MultiFeeDistribution contract. While examining its implementation, we observe that the lpAmountMin parameter in the deposit() function is set to 0 (line 137), meaning there is no slippage control when adding liquidity. The absence of slippage control exposes users to front-run attacks. As a result, users might end up providing liquidity at bad rates, which could lead to significant asset losses.

iolend-lending-contracts - LockZap.sol

```
122 function _zap(
123   address _asset,
124   uint256 _assetAmount,
125   uint256 _iolAmount,
126   address _from,
127   address _onBehalf,
128   uint256 _lockTypeIndex,
129   address _refundAddress
130 ) internal returns (uint256 liquidity) {
131   ...
132   iol.forceApprove(address(multipool), _iolAmount);
133   address token0 = multipool.token0();
134   address token1 = multipool.token1();
135   uint256 amount0Desired = token0 == _asset ? _assetAmount : _iolAmount;
136   uint256 amount1Desired = token1 == _asset ? _assetAmount : _iolAmount;
137   liquidity = multipool.deposit(amount0Desired, amount1Desired, address(this), 0);
138   IERC20(multipool.multipoolToken()).forceApprove(address(mfd), liquidity);
139   mfd.stake(liquidity, _onBehalf, _lockTypeIndex);
140   emit Zapped(_assetAmount, _iolAmount, _from, _onBehalf, _lockTypeIndex);
141   _refundDust(address(iol), _asset, _refundAddress);
142 }
```

Remediation Apply necessary slippage control during adding liquidity.

3.3.7 [L-3] Suggested _disableInitializers() in constructor()

Target	Category	IMPACT	LIKELIHOOD	STATUS
Multiple Contracts	Coding Practice	Low	Low	Acknowledged

The _disableInitializers() function is designed to prevent the execution of the initialization function within the contract. It is typically used in the implementation contracts of upgradeable contracts to ensure that the initialization logic cannot be inadvertently executed after deployment.

It is advisable to invoke _disableInitializers() within the constructor() of the implementation contracts. This measure effectively locks the initialization function, protecting the contract from unintended calls that could lead to security vulnerabilities.

iolend-lending-contracts - AToken.sol

```
contract AToken is VersionedInitializable, IncentivizedERC20("ATOKEN_IMPL", "ATOKEN_IMPL", 0), IAToken {
    ...
    function initialize(
        ILendingPool pool,
        address treasury,
        address underlyingAsset,
        IAaveIncentivesController incentivesController,
        uint8 aTokenDecimals,
        string calldata aTokenName,
        string calldata aTokenSymbol,
        bytes calldata params
    ) external override initializer {
        ...
    }
```

Remediation Properly apply _disableInitializers() in the constructor() of the following contracts: AToken, StableDebtToken, VariableDebtToken, Looping, Leverager, SwapLoopLeverager, SwapLoopWlpLeverager, EligibilityDataProvider, PriceProvider, ChefIncentivesController, MiddleFeeDistribution, MultiFeeDistribution, LockZap, etc.

4 Appendix

4.1 About AstraSec

AstraSec is a blockchain security company that serves to provide high-quality auditing services for blockchain-based protocols. With a team of blockchain specialists, AstraSec maintains a strong commitment to excellence and client satisfaction. The audit team members have extensive audit experience for various famous DeFi projects. AstraSec's comprehensive approach and deep blockchain understanding make it a trusted partner for the clients.

4.2 Disclaimer

The information provided in this audit report is for reference only and does not constitute any legal, financial, or investment advice. Any views, suggestions, or conclusions in the audit report are based on the limited information and conditions obtained during the audit process and may be subject to unknown risks and uncertainties. While we make every effort to ensure the accuracy and completeness of the audit report, we are not responsible for any errors or omissions in the report.

We recommend users to carefully consider the information in the audit report based on their own independent judgment and professional advice before making any decisions. We are not responsible for the consequences of the use of the audit report, including but not limited to any losses or damages resulting from reliance on the audit report.

This audit report is for reference only and should not be considered a substitute for legal documents or contracts.

4.3 Contact

Phone	+86 156 0639 2692
Email	contact@astrasec.ai
Twitter	https://twitter.com/AstraSecAI