Keiko Finance Audit Report

Prepared by KupiaSec

Version 1.0

Auditors

KupiaSec

Nov 24th, 2024

Contents

1 About KupiaSec

KupiaSec is a team of Web3 security experts that operates with transparency and a meritocratic spirit.

KupiaSec executes the modified MPA model for the Private Audits, a.k.a. **Solo Audit by a Lead + Internal Competition + Mitigation Review**.

- Solo Audit by a Lead

  KupiaSec assigns a senior auditor as a Lead Auditor based on the protocol category. The Lead Auditor is responsible for the first phase and will be the main point of contact for the client. The Lead Auditor shares the analysis and findings with the team.

- Internal Competition

  KupiaSec assigns 5~7 assist auditors to conduct the second phase. The auditors compete to find the most issues and the best solutions. This phase ensures the protocol goes through a rigorous review process by "many eyes" in a competitive environment.

- Mitigation Review

  After the protocol team has fixed the issues, KupiaSec conducts a final review to ensure all the issues are resolved.

2 Disclaimer

The KupiaSec team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Keiko Finance is a permissionless CDP protocol deploying on the Hyperliquid network where you can open a collateralised debt position and mint KEI against different assets of the Hyperliquid ecosystem.

The protocol enables users to create collateralized debt positions (CDP). Which means you can deposit certain supported assets from the Hyperliquid ecosystem as collateral and, in return, borrow a stablecoin called KEI. KEI is designed to maintain a stable value, making it useful for various financial activities.

5 Audit Scope

AddressBook.sol KEI.sol VaultOperations.sol VaultManager.sol

Summary of Commits

Project Name	Keiko Finance
Repository	keiko-contracts
Initial Commit	ee5bab420338...
Mitigation Commit	eda96f141932...

6 Executive Summary

KupiaSec executed a modified Multi-Phase Audit model, a.k.a. **Solo Audit by a Lead + Internal Competition + Mitigation Review**.

3 security researchers involved in the audit and reviewed the scope.

Execution Timeline

Phase-1: Audit by a Lead	Nov 19th - Nov 20th
Phase-2: Internal Competition	Nov 20th - Nov 22th
Initial Report Delivery	Nov 24th, 2024
Phase-3: Mitigation Review	N/A
Final Report Delivery	Nov 24th, 2024

Issues Found

Critical Risk	1
High Risk	1
Medium Risk	3
Low Risk	0
Informational	0
Gas Optimizations	0
Total Issues	5

Summary of Findings

[C-1] Removing vault from `VaultSorter` freezes vault	Open
[H-1] Accruing interests in units of minutes allows vault owners not to pay interests for their debts	Open
[M-1] Incorrect min range validation	Open
[M-2] `setAddresses` function can be front-run	Open

[M-3] Incorrect subtract amount of collateral	Open

7 Findings

7.1 Critical Risk

7.1.1 Removing vault from `VaultSorter` freezes vault

Description:

```
function redeemVault(address vaultCollateral, uint256 redemptionAmount, address prevId, address nextId)
↪   external {
    // ...

    while (redemptionAmount > 0 && currentVault != address(0)) {
        // ...

        uint256 vaultARS = IVaultManager(vaultManager).calculateARS(vaultCollateral, currentVault);
        emit VaultRedeemed(currentVault, vaultCollateral, msg.sender, redeemableAmount, vaultARS);

        if (redemptionAmount > 0) {
@>          IVaultSorter(vaultSorter).removeVault(vaultCollateral, currentVault);
            currentVault = IVaultSorter(vaultSorter).getLast(vaultCollateral);
        } else {
            IVaultSorter(vaultSorter).reInsertVault(vaultCollateral, currentVault, vaultARS, prevId,
            ↪   nextId);
        }
    }

    if (totalDebtRedeemed > 0) {
        IERC20(debtToken).burn(msg.sender, totalDebtRedeemed);
    }

    if (totalCollateralRedeemed > 0) {
        IERC20(vaultCollateral).transfer(msg.sender, totalCollateralRedeemed);
    }
}
```

`redeemVault` function is used to exchange collateral token with underlying debt token, which subtracts debt from vault owners. However, as shown in the code snippet above, when all debt of a vault is redeemed, it removes the vault from `VaultSorter`.

This exposes a critical vulnerability like:

- Once the vault is removed from `VaultSorter`, the vault owner can't take further actions on their own vaults, because it will revert in further `remove` or `reInsertVault` calls.
- Malicious attacker can iterate through all vaults and redeem all debts to make every fault freezed.

Impact: User vaults can be frozen which causes assets stuck in the contract and loss for users.

Proof Of Concept:

Recommended Mitigation: It should use `reInsertVault` instead of `removeVault`.

Client: Confirmed. Fixed in PR

KupiaSec: Verified.

7.2 High Risk

7.2.1 Accruing interests in units of minutes allows vault owners not to pay interests for their debts

Description:

```
function manageDebtInterest(address _vaultCollateral, address _vaultOwner) internal returns (uint256,
↪  uint256, uint256) {
    (uint256 collateralAmount, uint256 debtAmount, uint256 vaultMCR) =
    ↪  IVaultManager(vaultManager).getVaultData(_vaultCollateral, _vaultOwner);
    uint256 lastUpdated = lastDebtUpdateTime[_vaultOwner][_vaultCollateral];
    uint256 currentTimestamp = block.timestamp;
    uint256 timeElapsed = currentTimestamp - lastUpdated;

    if (timeElapsed > 0) {
        uint256 vaultInterestRate = IVaultManager(vaultManager).getVaultInterestRate(_vaultCollateral,
        ↪  _vaultOwner);
@>      uint256 accruedInterest = calculateAccruedInterest(debtAmount, vaultInterestRate, timeElapsed);

        debtAmount += accruedInterest;
        totalAccruedDebt += accruedInterest;
        totalDebt[_vaultCollateral] += accruedInterest;
        totalProtocolDebt += accruedInterest;

        // Update the vault data with the new debt amount including accrued interest
        IVaultManager(vaultManager).adjustVaultData(_vaultCollateral, _vaultOwner, collateralAmount,
        ↪  debtAmount, vaultMCR);
    }

    // Reset the last update time to the current timestamp
    lastDebtUpdateTime[_vaultOwner][_vaultCollateral] = currentTimestamp;

    return (collateralAmount, debtAmount, vaultMCR);
}
```

In `VaultOperations` contract, `manageDebtInterest` function is called every time there happens changes to Vaults, of which the goal is to accrue interests of debt of users to increase the debts.

```
function calculateAccruedInterest(uint256 _currentDebt, uint256 _interestRate, uint256 _timeElapsed)
↪  internal pure returns (uint256) {
    if (_currentDebt == 0 || _interestRate == 0 || _timeElapsed == 0) {
        return 0;
    }

    uint256 minutesElapsed = _timeElapsed / 1 minutes;
    uint256 baseRate = DECIMAL_PRECISION + (_interestRate / MINUTES_IN_YEAR); // Convert annual interest
    ↪  rate to per-minute rate
    uint256 compoundFactor = VaultMath.decPow(baseRate, minutesElapsed);
    uint256 newDebt = (_currentDebt * compoundFactor) / DECIMAL_PRECISION;

    return newDebt - _currentDebt;
}
```

When `calculateAccruedInterest` function is called, it calculates interest based on minutes elapsed.

This exposes a vulnerability that users can avoid paying interest for their debts by making changes to their vault every 40-50 seconds, for example, adding or removing 1 wei of collateral to/from their vault.

Impact: Users avoid paying interest rate which is core functionality of the protocol

Proof Of Concept:

Recommended Mitigation: The best mitigation would be handling interest based on `seconds` elapsed, or rounding

up in calculation of minutes elapsed would also fix the issue.

Client: Confirmed. Fixed in PR

KupiaSec: Verified.

7.3 Medium Risk

7.3.1 Incorrect min range validation

Description: In `VaultManager` contract, `setMinRange` function is called by owner, which is to set minimum collateral range of a collateral asset.

```
function setMinRange(address collateral, uint256 minRange) public onlyOwner {
    require(isAddressValid(collateral), "Invalid collateral");
@>  require(minRange >= 100e16, "Min MCR 100%");

    CollateralParams storage collParams = collateralParams[collateral];
    collParams.minRange = minRange;
    emit MinRangeUpdated(collateral, minRange);
}
```

However, it validates the input against `100e16`, which is wrong. The correct minimum range should be `100e18`.

Impact: Incorrect parameter setting by owner will allow users to drain from vaults.

Proof Of Concept:

Recommended Mitigation:

```
function setMinRange(address collateral, uint256 minRange) public onlyOwner {
    require(isAddressValid(collateral), "Invalid collateral");
-   require(minRange >= 100e16, "Min MCR 100%");
+   require(minRange >= 100e18, "Min MCR 100%");

    CollateralParams storage collParams = collateralParams[collateral];
    collParams.minRange = minRange;
    emit MinRangeUpdated(collateral, minRange);
}
```

Client: Confirmed. Fixed in PR

KupiaSec: Verified.

7.3.2 `setAddresses` function can be front-run

Description: In `AddressBook` contract, `setAddresses` function is used to initialize addresses of protocol contracts which can be used in `VaultOperations` further.

However, the function lacks of access control, so that any malicious user can call the function right after the deployment.

Impact: Invalid addresses can be set which might cause further potential issues.

Proof Of Concept:

Recommended Mitigation: Add authorization to the function, like `onlyOwner` modifier.

Client: Confirmed. Fixed in PR

KupiaSec: Verified.

7.3.3 Incorrect subtract amount of collateral

Description:

```
function liquidateVault(address vaultCollateral, address vaultOwner, address prevId, address nextId)
↪  external {
  // ...

  if (debtToOffset == debtAmount) {
      activeVaults -= 1;
      lastDebtUpdateTime[vaultOwner][vaultCollateral] = 0;

      IVaultSorter(vaultSorter).removeVault(vaultCollateral, vaultOwner);
      IVaultManager(vaultManager).adjustVaultData(vaultCollateral, vaultOwner, 0, 0, 0);

      // Update total debt and collateral
      totalDebt[vaultCollateral] -= debtAmount;
@>    totalCollateral[vaultCollateral] -= spDistribution;

      if (remainingCollateral > 0) {
          IERC20(vaultCollateral).transfer(vaultOwner, remainingCollateral);
      }

    // Partial liquidation (Not enough KEI in SP)
  } else {
    // ...
  }

  // ...
}
```

The `liquidateVault` function is used to liquidate users' vault that go below minimum collateral ratio. After the vault is liquidated, it updates corresponding state data as shown above.

However, when the vault is completely liquidated, it updates `totalCollateral` incorrect way. It deducts `spDistribution` from `totalCollateral` but it has to be `collateralAmount`.

Impact: Incorrect accounting of total collateral and the value gets inflated.

Proof Of Concept:

Recommended Mitigation: It should deduct `collateralAmount` instead of `spDistribution`.

Client: Confirmed. Fixed in PR

KupiaSec: Verified.