CD SECURITY

AUDIT REPORT

Keiko Finance

September 2024

Prepared by
yotov721
Arnie

Introduction

A time-boxed security review of the **Keiko** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **Keiko**

The protocol lets users create custom vaults where they deposit collateral in exchange for KEI - the protocol's stablecoin.

Chainlink price feeds are used to fetch collateral prices, keeping a stable peg. Users can stake their KEI tokens which are used to liquidate undercolateralised vaults. In exchange stakers get the liquidated vaults collateral at a discount.

The protocol also offers users the ability to redeem collateral from a vaults by paying off the corresponding debt amount. This is done in an orderly manner and vaults are passed by the vault sorter contract until the redemption amount is met.

Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic, and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - ec88053f0a86c7f80aeb71a5c73cea9357e2da70**

Scope

The following smart contracts were in scope of the audit:

- `VaultOperations.sol`
- `VaultManager.sol`
- `KEI.sol`

The following number of issues were found, categorized by their severity:

- Critical & High: 1 issues
- Medium: 7 issues
- Low: 1 issues

Findings Summary

ID	Title	Severity
[H-01]	User can close a vault that should be liquidated	High
[M-01]	Addresses are set to the 0 addr in constructor	Medium
[M-02]	Owner doesn't have to wait before whitelisting	Medium
[M-03]	Users can burn protocol fees before the recipients are set	Medium
[M-04]	Use SafeERC20 to transfer tokens	Medium
[M-05]	A user could evade acquiring interest on his debt	Medium
[M-06]	The protocol could acquire bad debt	Medium
[M-07]	Interest acquired on debt compounds when it shouldn't	Medium
[L-01]	Possible DoS due to unbounded loop	Low

Detailed Findings

[H-01] User can close a vault that should be liquidated

Severity

Impact: High

Likelihood: Medium

Description

The `closeVault` function closes a vault and returns the collateral to the user. The problem occurs because the function does not check if the vault that is to be closed is eligble for liquidation as seen below.

```
    function closeVault(address vaultCollateral) external {
        (uint256 collateralAmount, uint256 debtAmount,) =
manageDebtInterest(vaultCollateral, msg.sender);
        require(collateralAmount != 0, "Vault doesnt exists");

        totalCollateral[vaultCollateral] -= collateralAmount;
        totalDebt[vaultCollateral] -= debtAmount;
        totalProtocolDebt -= debtAmount;
        activeVaults -= 1;
        lastDebtUpdateTime[msg.sender][vaultCollateral] == 0;

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

        IERC20(debtToken).burn(msg.sender, debtAmount);
        IERC20(vaultCollateral).transfer(msg.sender, collateralAmount);

        emit VaultClosed(msg.sender, collateralAmount, debtAmount);
    }
```

This means that when the user goes to close the vault, he will leave the protocol with bad debt an increase the chance that the protocol will become insolvent.

This will also allow the user to bypass liquidations by front running the liquidation call with closing the vault.

Recommendations

It is important to not allow vaults who are not eligible for liquidation to close vaults, users should increase collateral and achieve the MCR in order to close said vault.

[M-01] Addresses are set to the 0 addr in constructor

Severity

Impact: Medium

Likelihood: High

Description

In the constructor the addresses for `vaultManager` and the `StabilityPool` are set. This is evident in the code snippet below.

```
    constructor() ERC20("KEI Stablecoin", "KEI", DECIMALS) {
        whitelisted[vaultManager] = true;
```

```
        whitelisted[stabilityPool] = true;
    }
```

The problem is that during construction, the addresses have not yet been set and thus we are whitelisting the 0 address here.

Below we can observe that the addresses are indeed set in the `setAddresses` function in AddressBook.sol since this address can only be called after the construction of the contract, when whitelisting the `vaultManager` and the `stabilityPool`, we will be whitelisting the 0 address.

And because we must wait 2 weeks before we can whitelist the addresses in the `addWhitelist` function, the protocol with be DOSed for 2 weeks. This can be observed below...

```
    function addWhitelist(address _address) external onlyOwner {
        require(block.timestamp > requestedWhitelistTimestamp[_address] +
TIMELOCK_DURATION, "Timelock period has not passed");

        allowanceWhitelist[_address] = true;
        mintWhitelist[_address] = true;
        delete requestedWhitelistTimestamp[_address];
        emit WhitelistChanged(_address, true);
    }
```

Let us note that `TIMELOCK_DURATION` = 14 days

Recommendations

set the addresses in the constructor before whitelisting said addresses in order to ensure the correct address is being whitelisted.

[M-02] Owner doesn't have to wait before whitelisting

Severity

Impact: High

Likelihood: Medium

Description

Given that whitelisted addresses have a lot of power, the protocol wants to ensure a 14 day timelock is in place before any address can be whitelisted this is evident from the function comments below...

```
     * @notice Initiates a request to whitelist an address by starting the
timelock period
     * @notice Given the admin key retains the ability to add debttoken
minters for future
```

```
                    deployments or upgrades a long timelock (14d) is
established.
      * @param _address The address to be considered for whitelisting
      * @dev Can only be called by the contract owner
     */
     function requestWhitelist(address _address) external onlyOwner {
```

The problem occurs because the owner does not need to wait the full `TIMELOCK_DURATION` when adding an address to the whitelist, let me explain.

The normal flow of whitelisting an address is simple, call `requestWhitelist`, this will set the `requestedWhitelistTimestamp` as shown below

```
     function requestWhitelist(address _address) external onlyOwner {
          require(!whitelisted[_address], "Address already whitelisted");

          requestedWhitelistTimestamp[_address] = block.timestamp;
          emit WhitelistRequested(_address, block.timestamp);
     }
```

this variable is then used in the function `addWhitelist`

```
     function addWhitelist(address _address) external onlyOwner {
          require(block.timestamp > requestedWhitelistTimestamp[_address] +
TIMELOCK_DURATION, "Timelock period has not passed");

          allowanceWhitelist[_address] = true;
          mintWhitelist[_address] = true;
          delete requestedWhitelistTimestamp[_address];
          emit WhitelistChanged(_address, true);
     }
```

The code is trying to enforce the 2 week timelock in the require statement.

The problem is that the owner can simply call `addWhitelist` first without calling `requestWhitelist`. because the default value of uint256 is 0, then the require statement will pass and let the owner whitelist an address without waiting the `TIMELOCK_DURATION`.

Recommendations

When requesting whitelist i suggest adding another variable, a mapping named `whitelistRequested` that tracks addresses to boolean. When the `requestWhitelist` function is called, set the variable `whitelistRequested` to true for a given address. Then in the `addWhitelist` Function we can also require the value is true and therefore the TIMELOCK_DURATION is not compromised. A possible fix is show below

```
    function addWhitelist(address _address) external onlyOwner {
        require(block.timestamp > requestedWhitelistTimestamp[_address] +
TIMELOCK_DURATION, "Timelock period has not passed");
    require(whitelistRequested[_address] == true, "this address has not
been requested for whitelist");
        ....
        ....
```

Another potential fix that is more simple is to check that `requestedWhitelistTimestamp[_address] > 0` in the `addWhitelist` function.

[M-03] Users can burn protocol fees before the recipients are set

Severity

Impact: High

Likelihood: Low

Description

The function `mintVaultsInterest` will send the interest generated by the protocol to the recipients:

```
    function mintVaultsInterest() external {
        uint256 interestSinceLastMint = totalAccruedDebt -
lastRecordedAccruedDebt;

        require(interestSinceLastMint > 0, "No interest to mint");
        lastRecordedAccruedDebt = totalAccruedDebt;  // Update the last
recorded debt to the current

        uint256 remainingInterest = interestSinceLastMint;

        // Mint to configured recipients
        for (uint i = 0; i < mintRecipients.length; i++) {
            uint256 amountToMint = (interestSinceLastMint *
mintRecipients[i].percentage) / 10000;
            if (amountToMint > 0) {
                IERC20(debtToken).mint(mintRecipients[i].recipient,
amountToMint);
                remainingInterest -= amountToMint;
            }
        }

        // Mint any remaining amount to the default recipient
        if (remainingInterest > 0 && defaultInterestRecipient !=
address(0)) {
```

```
                IERC20(debtToken).mint(defaultInterestRecipient,
remainingInterest);
            }

            emit VaultInterestMinted(interestSinceLastMint);
        }
```

the function is external meaning that it can be called by anyone. If there is no recipients the fees will just be lost because debt token will not be minted to anyone and lastRecordedAccruedDebt will be updated.

Because the recipients and the `defaultInterestRecipient` are not set in the constructor and must be set after by the owner, a user may call the function `mintVaultsInterest` and essentially not allow the interest to be claimed since the recipients have yet to be set.

Recommendations

Set the `mintVaultsInterest` function to onlyOwner.

[M-04] Use SafeERC20 to transfer tokens

Severity

Impact: High

Likelihood: Low

Description

Tokens not compliant with the ERC20 specification could return `false` from the transfer function call to indicate the transfer failed, while the calling contract would not notice the failure if the return value is not checked. Checking the return value is a requirement, as written in the EIP-20 specification:

> Callers MUST handle false from returns (bool success). Callers MUST NOT assume that false is never returned!

If the return value is not checked this could lead to loss of funds for users or the protocol getting drained.

There are several occurrences of this issue: `VaultOperations::createVault()` - `transferFrom` `VaultOperations::closeVault()` - `transfer` `VaultOperations::adjustVault()` - two occurrences `VaultOperations::redeemVault()` - `transfer`

Recommendations

Use the SafeERC20 library implementation from OpenZeppelin and call `safeTransfer` or `safeTransferFrom` when transferring ERC20 tokens.

[M-05] A user could evade acquiring interest on his debt

Severity

Impact: Low

Likelihood: High

Description

The protocol lets users create vaults where they store collateral and mint the protocol's stable coin in return. The amount of stable coin minted is defined as debt. The debt minted acquires interest over time which calculated in the `VaultOperations::calculateAccruedInterest` function and is time dependent. The `lastDebtUpdateTime` mapping is used to store the last timestamp interest was acquired. The only place where the function is called in from the `manageDebtInterest()` function, where if the `lastDebtUpdateTime` is zero, it is set to `block.timestamp` and no interest is accrued.

```
=>      uint256 lastUpdated = lastDebtUpdateTime[_vaultOwner]
[_vaultCollateral]; // load last update debt time
        uint256 currentTimestamp = block.timestamp; // ok

        if (lastUpdated == 0 || lastUpdated >= currentTimestamp ||
debtAmount == 0) {
=>          lastDebtUpdateTime[_vaultOwner][_vaultCollateral] =
currentTimestamp;
            return (collateralAmount, debtAmount, vaultMCR);
        }

        uint256 vaultInterestRate =
IVaultManager(vaultManager).getVaultInterestRate(_vaultCollateral,
_vaultOwner);
=>      uint256 accruedInterest = calculateAccruedInterest(debtAmount,
vaultInterestRate, lastUpdated, currentTimestamp);
```

When a user is liquidated his `lastDebtUpdateTime` is set to 0. This is fine, but not if he is partially liquidated. If a user is partially liquidated his `lastDebtUpdateTime` will set 0 and stop accruing interest even if there is collateral and debt left.

Also a user can self-liquidate.

Having this in mind a malicious user could just lower his min CR and self liquidate just a little amount so his `lastDebtUpdateTime` is set to 0 and stop acquiring interest. In the same transaction he cal call `adjustVault` and add more collateral so someone else doesn't liquidate him.

This way a malicious user could escape acquiring interest on his debt.

Recommendations

In the `VaultOperations::liquidateVault` function set the `lastDebtUpdateTime` to zero only if it is full liquidation, otherwise set it to `block.timestamp` which done by default in `manageDebtInterest`.

```
 function liquidateVault(address vaultCollateral, address vaultOwner,
address prevId, address nextId) external {
        (uint256 collateralAmount, uint256 debtAmount, uint256 vaultMCR) =
manageDebtInterest(vaultCollateral, vaultOwner);
...

-       lastDebtUpdateTime[vaultOwner][vaultCollateral] = 0;

        // Full liquidation
        if (debtToOffset == debtAmount) {
+           lastDebtUpdateTime[vaultOwner][vaultCollateral] = 0;
            activeVaults -= 1;
            IVaultSorter(vaultSorter).removeVault(vaultCollateral,
vaultOwner);
            IVaultManager(vaultManager).adjustVaultData(vaultCollateral,
vaultOwner, 0, 0, 0);

            // Update total debt and collateral
            totalDebt[vaultCollateral] -= debtAmount;
            totalCollateral[vaultCollateral] -= collateralAmount;

        // Partial liquidation
        } else {
...
    }
```

[M-06] The protocol could acquire bad debt

Severity

Impact: High

Likelihood: Low

Description

The protocol mints a stable coin in exchange for collateral. Users open vaults where they deposit collateral and are minted the stable coin. They also set a min Collateral Ratio if fallen bellow which they can get liquidated.

Liquidations here happen by stakers staking `debtTokens` in the `StabilityPool` contract - when someone is liquidated part (or all) of the staked tokens are burned. In exchange the stakers get part (or all) of the collateral **at a discount** as a reward for staking debt tokens.

The problem with this approach is that if collateral token prices fall fast (crash), which is not uncommon in the crypto market and there are no staked debt tokens left to cover all the liquidatable positions, the protocol could acquire bad debt.

Recommendations

Add some mechanism that limit users from minting more debt tokens unless there is staked debt to cover liquidations - this can be in some ratio.

[M-06] Interest acquired on debt compounds when it shouldn't

Severity

Impact: Medium

Likelihood: High

Description

Users can create vault in which they deposit collateral in exchange for minting the protocol's stable coin debt token - `KEI`. When users have debt their debt acquires interest over time. When interest is acquired it is added to the totalDebt amount of the user's vault. This however is a problem since the next time a user acquires interest, interest would be added on top of the interest that was last acquired, effectively compounding.

Example a user has 10_000 in debt with 2 % interest per year. If his interest is updated only one at the end of the 365 day he will not have 10_000 + 2 % = 10_200 debt. However if the his interest is updated once on the 6th month and once on the 12th that would equal to: 10_000 + 1 % = 10_100 + 1 % = 10_101 debt.

Having in mind that the lower the min CR a user sets the higher interest he gets and that anyone could call `updateVaultInterest` at anytime, a user's debt could grow a lot faster than expected.

Recommendations

Use a separate mapping to store the acquired interest and gather interest only on the base debt amount.

[L-01] Possible DoS due to unbounded loop

The protocol uses an array to store the valid collaterals. When a user wants to open a new vault, the collateral address he passes is checked against the list of valid addresses.

```
    function isAddressValid(address _address) public view returns(bool) {
         for (uint i = 0; i < validCollateral.length; i++) {
             if (validCollateral[i] == _address) {
                 return true;
             }
         }

         return false;
     }
```

This is fine, but on more expensive chains like Mainnet if the array gets to big it could cause out of gas error and block the protocol.

Recommendations

Use a mapping of address => bool instead to check if an address is valid collateral