SMART CONTRACT AUDIT REPORT

for

KEEPERDAO

Prepared By: Shuxiao Wang

Hangzhou, China

Jun. 26, 2020

Document Properties

Client	KeeperDAO
Title	Smart Contract Audit Report
Target	KeeperDAO
Version	1.0-rc1
Author	Chiachih Wu
Auditors	Huaguo Shi, Chiachih Wu
Reviewed by	Chiachih Wu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc1	Jun. 26, 2020	Chiachih Wu	Status Update
0.1	Jun. 23, 2020	Chiachih Wu	Initial Draft

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the **KeeperDAO** design document and related smart contract source code, we in the report outline our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About KeeperDAO

KeeperDAO is a protocol that economically incentivizes pooled participation in 'keeper' strategies which manage liquidations and rebalances on applications spanning margin trading, lending and exchange. This allows participants to earn passive income in a game-theory-optimal fashion whilst ensuring decentralized finance applications remain liquid and orderly.

The basic information of KeeperDAO is as follows:

Table 1.1: Basic Information of KeeperDAO

Item	Description
Issuer	KeeperDAO
Website	https://github.com/keeperdao/protocol
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	Jun. 26, 2020

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit:

- https://github.com/keeperdao/protocol (1009582d)
- https://github.com/keeperdao/protocol/pull/13

1.2 About PeckShield

PeckShield Inc. [18] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [13]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [12], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this audit does not give any warranties on finding all possible security issues of the given smart contract(s), i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as an investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the KeeperDAO implementation. During the first phase of our audit, we studied the smart contract source code and ran our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■■
Low	1	■
Informational	5	■■■■■
Total	8	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities, 1 low-severity vulnerability, and 5 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Info.	Optimization Suggestions for kToken	Business Logics	Confirmed
PVE-002	Info.	Gas Optimizations for migrate()	Business Logics	Confirmed
PVE-003	Medium	Unprotected Privileged Interfaces	Initialization and Cleanup	Confirmed
PVE-004	Info.	Gas Optimizations for borrow()/withdraw()	Business Logics	Confirmed
PVE-005	Medium	Specification Mismatch in borrow()	Documentation	Resolved
PVE-006	Low	Out-of-gas Risk in migrate()	Bad Coding Practices	Confirmed
PVE-007	Info.	Insufficient Checks in register()	Business Logics	Resolved
PVE-008	Info.	Lack of Kill Switch Implementation	Business Logics	Resolved

Please refer to Section 3 for details.

3 | Detailed Results

3.1 Optimization Suggestions for kToken

- ID: PVE-001
- Severity: Informational
- Likelihood: Low
- Impact: N/A
- Target: `KToken`
- Category: Business Logics [10]
- CWE subcategory: CWE-841 [8]

Description

In `KToken` contract, the variable, `decimals_` is used for display purposes while all the calculations are done in the underlying token contract. We noticed that `decimals_` could be set by `initialize()` and `rename()` such that the operator may set a wrong `decimals_`. This leads to a possible mismatch between the display decimals and the underlying token's real decimals. Actually, most tokens have the `decimals` public variable and corresponding getter functions implemented so that we can simply retrieve the decimals from the token contracts.

```
    function initialize(string memory name, string memory symbol, uint8 decimals,
        address _underlying) public initializer {
        // Initialize the minter and pauser roles
        ERC20Mintable.initialize(msg.sender);
        ERC20Pausable.initialize(msg.sender);
        CanReclaimTokens.initialize(msg.sender);

        name_ = name;
        symbol_ = symbol;
        decimals_ = decimals;
        underlying_ = _underlying;
    }
```

Listing 3.1: contracts/protocol/KToken.sol

```
    /// @param _name The new name of the token.
    /// @param _symbol Thew new symbol of the token.
    /// @param _decimals Thew new decimals of the token.
```

```
    function rename(string calldata _name, string calldata _symbol, uint8 _decimals) external onlyOperator {
        name_ = _name;
        symbol_ = _symbol;
        decimals_ = _decimals;
        emit LogRenamed(_name, _symbol, _decimals);
    }
```

Listing 3.2: contracts/protocol/KToken.sol

Recommendation Set `decimals_` through the underlying token's decimals in `initialize()` and remove decimals setting in `rename()`.

```
    function initialize(string memory name, string memory symbol, address _underlying) public initializer {
        // Initialize the minter and pauser roles
        ERC20Mintable.initialize(msg.sender);
        ERC20Pausable.initialize(msg.sender);
        CanReclaimTokens.initialize(msg.sender);

        name_ = name;
        symbol_ = symbol;
        underlying_ = _underlying;
        decimals_ = ERC20Detailed(underlying_).decimals();

    }
```

Listing 3.3: contracts/protocol/KToken.sol

```
    /// @param _name The new name of the token.
    /// @param _symbol Thew new symbol of the token.
    function rename(string calldata _name, string calldata _symbol) external onlyOperator {
        name_ = _name;
        symbol_ = _symbol;
        emit LogRenamed(_name, _symbol);
    }
```

Listing 3.4: contracts/protocol/KToken.sol

3.2 Gas Optimizations for migrate()

- ID: PVE-002
- Severity: Informational
- Likelihood: Low
- Impact: N/A
- Target: `LiquidityPoolV1`
- Category: Business Logics [10]
- CWE subcategory: CWE-283 [4]

Description

In KeeperDAO, the `migrate()` function enables the operator to migrate the assets and data from one `LiquidityPool` to another. Since there're multiple tokens registered in the old `LiquidityPool`, the `migrate()` function needs to go through each registered token for migrating corresponding assets and data. Before executing `migrate()`, `LiquidityPoolV1` needs to be added into the operator list of the `_newLP` contract, so that the `_newLP.register()` call in line 129 can be proceeded. However, the function does not check whether `address(this)` has the permission of `_newLP` such that the execution of `migrate()` could lead to some no-effect code, which is a waste of gas.

```
124     function migrate(ILiquidityPool _newLP) public onlyOperator {
125         for (uint256 i = 0; i < registeredTokens.length; i++) {
126             address token = registeredTokens[i];
127             kTokens[token].addMinter(address(_newLP));
128             kTokens[token].renounceMinter();
129             _newLP.register(kTokens[token]);
130             if (token != ETHEREUM) {
131                 ERC20(token).safeTransfer(address(_newLP), borrowableBalance(token));
132             } else {
133                 (bool success,) = address(_newLP).call.value(borrowableBalance(token))("
                        ");
134                 require(success, "Transfer Failed");
135             }
136         }
137         _newLP.renounceOperator();
138     }
```

Listing 3.5: LiquidityPool.sol

Recommendation Check if `LiquidityPoolV1` address has been added into the operator list of `_newLP`.

```
124     function migrate(ILiquidityPool _newLP) public onlyOperator {
125         require(_newLP.isOperator(address(this)), "migrate: LiquidityPoolV1 does not
                have the operator role of _newLP");
126
127         for (uint256 i = 0; i < registeredTokens.length; i++) {
128             address token = registeredTokens[i];
129             kTokens[token].addMinter(address(_newLP));
130             kTokens[token].renounceMinter();
131             _newLP.register(kTokens[token]);
132             if (token != ETHEREUM) {
133                 ERC20(token).safeTransfer(address(_newLP), borrowableBalance(token));
134             } else {
135                 (bool success,) = address(_newLP).call.value(borrowableBalance(token))("
                        ");
136                 require(success, "Transfer Failed");
137             }
138         }
139         _newLP.renounceOperator();
```

```
    }
```

Listing 3.6: LiquidityPool.sol

3.3 Unprotected Privileged Interfaces

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `KToken;LiquidityPoolV1`
- Category: Initialization and Cleanup Errors [11]
- CWE subcategory: CWE-454 [6]

Description

In `KToken` and `LiquidityPoolV1` contracts, the `initialize()` function utilizes the `initializer` modifier from OpenZeppelin's `Initializable` contract to ensure that the initialization would be only executed once. However, `initializer` has no authentication such that a bad actor could send out front-running transactions whenever she identifies the KeeperDAO deployer's transactions for calling `initialize()`. Whenever a `KToken` or `LiquidityPoolV1` contract is deployed, the bad actor can compromise it by front-running. Although there's no profit for the attackers, they can launch DoS attacks when someone pay them to do it.

Recommendation Keep the contract deployer in the constructor and authenticate the caller in `initialize()`.

```
    constructor() public {
        owner = msg.sender;
    }
```

Listing 3.7: LiquidityPool.sol

```
    function initialize(string memory _VERSION, address _borrower) public initializer {
        require(msg.sender == owner);
        CanReclaimTokens.initialize(msg.sender);
        ReentrancyGuard.initialize();

        VERSION = _VERSION;
        borrower = IBorrowerProxy(_borrower);
    }
```

Listing 3.8: LiquidityPool.sol

3.4 Gas Optimizations for borrow()/withdraw()

- ID: PVE-004
- Severity: Informational
- Likelihood: Medium
- Impact: N/A
- Target: `LiquidityPoolV1`
- Category: Business Logics [10]
- CWE subcategory: CWE-841 [8]

Description

In `LiquidityPoolV1` contract, the `borrow()` function allows anyone to borrow tokens or ether from the Keeper and `withdraw()` allows kToken holders burn their kTokens for withdrawing the underlying assets from the Keeper. These two functions both transfer `_amount` of `_token` (or ether) to the caller (line 76 – 81 in `withdraw()` and line 96 – 100 in `borrow()`).

```
72     function withdraw(address payable _to, address _token, uint256 _amount) external
           nonReentrant {
73         require(address(kTokens[_token]) != address(0x0), "Token is not registered");
74         uint256 initialBalance = borrowableBalance(_token);
75         uint256 burnAmount = _burnAmount(kTokens[_token].totalSupply(), initialBalance,
               _amount);
76         if (_token != ETHEREUM) {
77             ERC20(_token).safeTransfer(_to, _amount);
78         } else {
79             (bool success,) = _to.call.value(_amount)("");
80             require(success, "Transfer Failed");
81         }
```

Listing 3.9: LiquidityPool.sol

```
93     function borrow(address _token, uint256 _amount, bytes calldata _data) external
           nonReentrant {
94         require(address(kTokens[_token]) != address(0x0), "Token is not registered");
95         uint256 initialBalance = borrowableBalance(_token);
96         if (_token != ETHEREUM) {
97             ERC20(_token).safeTransfer(msg.sender, _amount);
98         } else {
99             msg.sender.call.value(_amount)("");
100        }
```

Listing 3.10: LiquidityPool.sol

However, if there's no enough `_token`, the transaction fails until the `safeTransfer()` function interacts with the underlying token contract which reverts the transaction due to not enough token balance. Actually, both `withdraw()` and `borrow()` get the current balance and store the number in `initialBalance`. We can simply compare `_amount` with `initialBalance` to reduce gas consumption in some cases.

Recommendation Compare `_amount` with `initialBalance` before the transfers.

```
72    function withdraw(address payable _to, address _token, uint256 _amount) external
          nonReentrant {
73        require(address(kTokens[_token]) != address(0x0), "Token is not registered");
74        uint256 initialBalance = borrowableBalance(_token);
75        require(initialBalance >= _amount, "Token is not enough");
76        uint256 burnAmount = _burnAmount(kTokens[_token].totalSupply(), initialBalance,
              _amount);
77        if (_token != ETHEREUM) {
78            ERC20(_token).safeTransfer(_to, _amount);
79        } else {
80            (bool success,) = _to.call.value(_amount)("");
81            require(success, "Transfer Failed");
82        }
```

Listing 3.11: LiquidityPool.sol

```
93    function borrow(address _token, uint256 _amount, bytes calldata _data) external
          nonReentrant {
94        require(address(kTokens[_token]) != address(0x0), "Token is not registered");
95        uint256 initialBalance = borrowableBalance(_token);
96        require(initialBalance >= _amount, "Token is not enough");
97        if (_token != ETHEREUM) {
98            ERC20(_token).safeTransfer(msg.sender, _amount);
99        } else {
100           msg.sender.call.value(_amount)("");
101       }
```

Listing 3.12: LiquidityPool.sol

Alternately, we can make `withdraw()`/`borrow()` partially succeed when the token balance is less than the `_amount` requested. As it depends on the corresponding spec modifications, we leave this option to the dev team.

3.5 Specification Mismatch in borrow()

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `MarketContractProxy`
- Category: Documentation [3]
- CWE subcategory: CWE-1068 [2]

Description

According to the design documentation, the caller of the `borrow()` function should return assets to the Keeper with extra fees before the end of the transaction. However, in line 103, the fee is not a requirement such that the profit of the Keeper is not guaranteed. In addition, the `_reciever` parameter described in the design document is removed. Instead, the current implementation uses the `BorrowerProxy` mechanism, which is a specification mismatch.

```
    function borrow(address _token, uint256 _amount, bytes calldata _data) external
        nonReentrant {
        require(address(kTokens[_token]) != address(0x0), "Token is not registered");
        uint256 initialBalance = borrowableBalance(_token);
        if (_token != ETHEREUM) {
            ERC20(_token).safeTransfer(msg.sender, _amount);
        } else {
            msg.sender.call.value(_amount)("");
        }
        borrower.lend(msg.sender, _data);
        uint256 finalBalance = borrowableBalance(_token);
        require(finalBalance >= initialBalance, "Borrower failed to return the borrowed
            funds");
        emit Borrowed(msg.sender, _token, _amount, finalBalance.sub(initialBalance));
    }
```

Listing 3.13: LiquidityPool.sol

Recommendation Enforce the fee charging mechanism in `borrow()` and/or revise the design documentation.

```
    function borrow(address _token, uint256 _amount, bytes calldata _data) external
        nonReentrant {
        require(address(kTokens[_token]) != address(0x0), "Token is not registered");
        uint256 initialBalance = borrowableBalance(_token);
        if (_token != ETHEREUM) {
            ERC20(_token).safeTransfer(msg.sender, _amount);
        } else {
            msg.sender.call.value(_amount)("");
        }
        borrower.lend(msg.sender, _data);
        uint256 finalBalance = borrowableBalance(_token);
        uint256 fee = _amount.mul(FEE_RATE); /* FIXME */
        require(finalBalance >= initialBalance + fee, "Borrower failed to return the
            borrowed funds");
        emit Borrowed(msg.sender, _token, _amount, finalBalance.sub(initialBalance));
    }
```

Listing 3.14: LiquidityPool.sol

3.6 Out-of-gas Risk in migrate()

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `MarketContractProxy`
- Category: Time and State [9]
- CWE subcategory: CWE-362 [5]

Description

In KeeperDAO, the `migrate()` function enables the operator to migrate the assets and data from one `LiquidityPool` to another. Since there're multiple tokens registered in the old `LiquidityPool`, the `migrate()` function needs to go through each registered token for migrating corresponding assets and data. However, the `registeredTokens.length` could be large enough such that the execution of `migrate()` could out-of-gas.

```
function migrate(ILiquidityPool _newLP) public onlyOperator {
    for (uint256 i = 0; i < registeredTokens.length; i++) {
        address token = registeredTokens[i];
        kTokens[token].addMinter(address(_newLP));
        kTokens[token].renounceMinter();
        _newLP.register(kTokens[token]);
        if (token != ETHEREUM) {
            ERC20(token).safeTransfer(address(_newLP), borrowableBalance(token));
        } else {
            (bool success,) = address(_newLP).call.value(borrowableBalance(token))("
                ");
            require(success, "Transfer Failed");
        }
    }
    _newLP.renounceOperator();
}
```

Listing 3.15: LiquidityPool.sol

Recommendation Migrate limited number of tokens in each run of migration and finalize the migration in the last run.

3.7 Insufficient Checks in register()

- ID: PVE-007
- Severity: Informational
- Likelihood: N/A
- Impact: Low
- Target: `LiquidityPoolV1`
- Category: Business Logics [10]
- CWE subcategory: CWE-754 [7]

Description

The `register()` function allows the operator to register a `kToken` into the `LiquidityPool`. With the help of the `KToken.underlying()` which returns the corresponding token address, the `kTokens` keeps the mapping from the registered token address to the KToken address. However, the uniqueness of the KToken address is not checked. Fortunately, the current implementation of KToken contract has no interface to change the `underlying_` private variable except the `initialize()` function. The operator has no chance to `register()` an already registered KToken because of the check in line 39.

```
38     function register(IKToken _kToken) external onlyOperator {
39         require(address(kTokens[_kToken.underlying()]) == address(0x0), "Underlying
               asset should not have been registered");
40         kTokens[_kToken.underlying()] = _kToken;
41         registeredTokens.push(address(_kToken.underlying()));
42         blacklistRecoverableToken(_kToken.underlying());
43     }
```

Listing 3.16: LiqidityPool.sol

Recommendation Check if the given KToken address has been registered.

```
    function register(IKToken _kToken) external onlyOperator {
        require(address(kTokens[_kToken.underlying()]) == address(0x0), "Underlying
            asset should not have been registered");
        require(registeredKTokens[_kToken] == false, "KToken should not have been
            registered");
        kTokens[_kToken.underlying()] = _kToken;
        registeredTokens.push(address(_kToken.underlying()));
        registeredKTokens[_kToken] = true;
        blacklistRecoverableToken(_kToken.underlying());
    }
```

Listing 3.17: LiqidityPool.sol

3.8 Lack of Kill Switch Implementation

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `LiquidityPoolV1`
- Category: Business Logics [10]
- CWE subcategory: CWE-841 [8]

Description

There's no kill switch implementation throughout the KeeperDAO system. As an essential part of the contingency plan, we suggest to implement a kill switch mechanism to stop the system when necessary.

Recommendation Add kill switch code on critical functions, for example, `deposit()`, `withdraw()`, `borrow()`, etc. As shown in the following code snippets, we can have a `live` variable to guard critical functions.

```
    function deposit(address _token, uint256 _amount) external payable nonReentrant
        returns (uint256) {
```

```
        require(live == 1, "deposit/not-live");
        require(address(kTokens[_token]) != address(0x0), "Token is not registered");
        uint256 initialBalance = borrowableBalance(_token);
        if (_token != ETHEREUM) {
            ERC20(_token).transferFrom(msg.sender, address(this), _amount);
        } else {
            initialBalance = initialBalance.sub(_amount);
            require(_amount == msg.value, "Incorrect eth amount");
        }
        uint256 mintAmount = _mintAmount(kTokens[_token].totalSupply(), initialBalance,
            _amount);
        kTokens[_token].mint(msg.sender, mintAmount);
        emit Deposited(msg.sender, _token, _amount, mintAmount);
    }
```

Listing 3.18: LiqidityPool.sol

The `live` variable could be initialized in `initialize()`.

```
    function initialize(string memory _VERSION, address _borrower) public initializer {
        CanReclaimTokens.initialize(msg.sender);
        ReentrancyGuard.initialize();

        VERSION = _VERSION;
        borrower = IBorrowerProxy(_borrower);
        live = 1;
    }
```

Listing 3.19: LiqidityPool.sol

The `cage()` function could be added to unset `live` as follows:

```
    function cage() external onlyOperator {
        live = 0;
    }
```

Listing 3.20: LiqidityPool.sol

3.9 Other Suggestions

Due to the fact that compiler upgrades might bring unexpected compatibility or inter-version consistencies, it is always suggested to use fixed compiler versions whenever possible. As an example, we highly encourage to explicitly indicate the Solidity compiler version, e.g., `pragma solidity 0.5.12;` instead of `pragma solidity ^0.5.12;`.

In addition, there is a known compiler issue that in all 0.5.x solidity prior to `Solidity 0.5.17`. Specifically, a private function can be overridden in a derived contract by a private function of the same name and types. Fortunately, there is no overriding issue in this code, but we still recommend using `Solidity 0.5.17` or above.

Moreover, we strongly suggest not to use experimental Solidity features or third-party unaudited libraries. If necessary, refactor current code base to only use stable features or trusted libraries.

Last but not least, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet.

4 Conclusion

In this audit, we thoroughly analyzed the KeeperDAO documentation and implementation. The audited system presents a unique innovation and we are really impressed by the design and implementation.The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

5 Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [14, 15, 16, 17, 19].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [20] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected `revert`.
- Result: Not found
- Severity: Medium

5.1.10 Unchecked External `Call`

- Description: Whether the contract has any external `call` without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless `Send`

- Description: Whether the contract is vulnerable to gasless `send`.
- Result: Not found
- Severity: Medium

5.1.12 `Send` Instead Of `Transfer`

- Description: Whether the contract uses `send` instead of `transfer`.
- Result: Not found
- Severity: Medium

5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to `Out-Of-Gas` exception.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.
- Result: Not found
- Severity: Medium

5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low

References

[1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github.com/ethereum/solidity/issues/4116.

[2] MITRE. CWE-1068: Inconsistency Between Implementation and Documented Design. https://cwe.mitre.org/data/definitions/1068.html.

[3] MITRE. CWE-1225: Documentation Issues. https://cwe.mitre.org/data/definitions/1225.html.

[4] MITRE. CWE-283: Unverified Ownership. https://cwe.mitre.org/data/definitions/283.html.

[5] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[6] MITRE. CWE-454: External Initialization of Trusted Variables or Data Stores. https://cwe.mitre.org/data/definitions/454.html.

[7] MITRE. CWE-754: Improper Check for Unusual or Exceptional Conditions. https://cwe.mitre.org/data/definitions/754.html.

[8] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[9] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

[10] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[11] MITRE. CWE CATEGORY: Initialization and Cleanup Errors. https://cwe.mitre.org/data/definitions/452.html.

[12] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[13] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[14] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[15] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[16] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[17] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

[18] PeckShield. PeckShield Inc. https://www.peckshield.com.

[19] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

[20] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.