



# SMART CONTRACT AUDIT REPORT

for

## Keep3r V2



Prepared By: Yiqun Chen

PeckShield  
November 7, 2021

## Document Properties

Client	Keep3r Network
Title	Smart Contract Audit Report
Target	Keep3r
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

## Version Info

Version	Date	Author(s)	Description
1.0	November 7, 2021	Xuxian Jiang	Final Release
1.0-rc	October 22, 2021	Xuxian Jiang	Release Candidate

## Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	About Keep3r . . . . .	4
1.2	About PeckShield . . . . .	5
1.3	Methodology . . . . .	5
1.4	Disclaimer . . . . .	6
<b>2</b>	<b>Findings</b>	<b>10</b>
2.1	Summary . . . . .	10
2.2	Key Findings . . . . .	11
<b>3</b>	<b>Detailed Results</b>	<b>12</b>
3.1	Improved Logic Of Keep3rJobManager::removeJob() . . . . .	12
3.2	Suggested Adherence Of Checks-Effects-Interactions Pattern . . . . .	13
3.3	Possible FrontRunning DoS Against Job Credit Withdrawal . . . . .	14
3.4	Simplified Logic of Keep3rJobFundableLiquidity::_phase() . . . . .	16
3.5	Improved Quote Calculation of Keep3rLibrary::getQuoteAtTick() . . . . .	17
3.6	Improved Job Migration in Keep3rJobMigration . . . . .	18
3.7	Possible DoS Against Keep3rJobDisputable::slashJob() . . . . .	20
<b>4</b>	<b>Conclusion</b>	<b>22</b>
	<b>References</b>	<b>23</b>

# 1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the Keep3r (V2) protocol, we outline in the report 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 Keep3r

Keep3r Network is an innovative decentralized keeper network for both projects that need external devops and external teams that need to find keeper jobs. Any person and/or a team that executes a job can be registered as a keeper and the executed job refers to a smart contract that wishes an external entity to perform an action. It is expected that the action is performed in "good will" without a malicious result. Projects wishing keepers to perform duties simply need to submit their contracts to the Keep3r Network. Once reviewed and approved via a governance process, keepers can begin fulfilling the works submitted earlier by projects. The basic information of Keep3r is as follows:

Table 1.1: Basic Information of Keep3r

Item	Description
Name	Keep3r Network
Website	<a href="https://keep3r.network/">https://keep3r.network/</a>
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 7, 2021

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

- <https://github.com/defi-wonderland/keep3r-v2-public.git> (06d800e)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- <https://github.com/defi-wonderland/keep3r-v2-public.git> (8c754d3)

## 1.2 About PeckShield

PeckShield Inc. [11] 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](mailto:contact@peckshield.com)).

Table 1.2: Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [10]:

- 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 checklist of 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 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) [9], 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

## 1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered

Table 1.3: The Full Audit Checklist

Category	Checklist Items
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

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

Category	Summary
<b>Configuration</b>	Weaknesses in this category are typically introduced during the configuration of the software.
<b>Data Processing Issues</b>	Weaknesses in this category are typically found in functionality that processes data.
<b>Numeric Errors</b>	Weaknesses in this category are related to improper calculation or conversion of numbers.
<b>Security Features</b>	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
<b>Time and State</b>	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.
<b>Error Conditions, Return Values, Status Codes</b>	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.
<b>Resource Management</b>	Weaknesses in this category are related to improper management of system resources.
<b>Behavioral Issues</b>	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
<b>Business Logic</b>	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.
<b>Initialization and Cleanup</b>	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
<b>Arguments and Parameters</b>	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
<b>Expression Issues</b>	Weaknesses in this category are related to incorrectly written expressions within code.
<b>Coding Practices</b>	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.





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 investment advice.



## 2 | Findings

### 2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `Keep3r` (V2) protocol. During the first phase of our audit, we study the smart contract source code and run 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	1	
Low	6	
Informational	0	
Total	7	

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 1 medium-severity vulnerability and 6 low-severity vulnerabilities.

Table 2.1: Key Keep3r Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Logic Of Keep3rJobManager::removeJob()	Business Logic	Fixed
PVE-002	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Fixed
PVE-003	Low	Possible FrontRunning DoS Against Job Credit Withdrawal	Time And State	Resolved
PVE-004	Low	Simplified Logic of Keep3rJobFundableLiquidity::_phase()	Coding Practices	Fixed
PVE-005	Low	Improved Quote Calculation of Keep3rLibrary::getQuoteAtTick()	Business Logic	Fixed
PVE-006	Low	Improved Job Migration in Keep3rJobMigration	Business Logic	Fixed
PVE-007	Medium	Possible DoS Against Keep3rJobDisputable::slashJob()	Business Logic	Resolved

Beside the identified issues, we emphasize that for any user-facing applications and services, 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 should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

## 3 | Detailed Results

### 3.1 Improved Logic Of Keep3rJobManager::removeJob()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: Keep3rJobManager
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

#### Description

One core functionality provided in the Keep3r protocol is the job handling. A Keep3r job can be any system that requires external execution. The protocol is not to define or restrict the actions a job may take, but to create an incentive mechanism for all parties involved. By design, a job may be submitted by any one. However, the removal of a job needs to be initiated by the governance. Each job has a number of associated states, including `jobOwner` and `jobPendingOwner`.

To elaborate, we show below the code snippet of `addJob()` and `removeJob()` functions. As the name indicates, the first function allows for adding a job while the second function is used for deleting a job. It comes to our attention when a job is deleted, the associated states are not completely removed. For example, the `jobOwner` state of a deleted job remains valid, which may be still used for job migration or ownership transfer.

```

12  /**
13   * @notice Allows governance to add new job systems
14   * @param _job address of the contract for which work should be performed
15   */
16  function addJob(address _job) external override {
17      if (_jobs.contains(_job)) revert JobAlreadyAdded();
18      if (hasBonded[_job]) revert AlreadyAKeeper();
19      _jobs.add(_job);
20      jobOwner[_job] = msg.sender;
21      emit JobAddition(_job, block.number, msg.sender);
22  }
```

```

24  /**
25   * @notice Allows governance to remove a job from the systems
26   * @param _job address of the contract for which work should be performed
27   */
28  function removeJob(address _job) external override onlyGovernance {
29      if (!_jobs.contains(_job)) revert JobUnexistent();
30      _jobs.remove(_job);
31      emit JobRemoval(_job, block.number, msg.sender);
32  }

```

Listing 3.1: `Keep3rJobManager::addJob()/removeJob()`

**Recommendation** Properly remove associated states that are not longer used after a job is deleted.

**Status** This issue has been fixed as this `removeJob()` function is removed in the upgrade: [2f3f209](#).

## 3.2 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Keep3rKeeperFundable`
- Category: Time and State [8]
- CWE subcategory: CWE-663 [3]

### Description

A common coding best practice in Solidity is the adherence of checks-effects-interactions principle. This principle is effective in mitigating a serious attack vector known as re-entrancy. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the DAO [13] exploit, and the recent Uniswap/Lendf.Me hack [12].

We notice there are several occasions the checks-effects-interactions principle is violated. Using the `Keep3rKeeperFundable` as an example, the `withdraw()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above re-entrancy.

Apparently, the interaction with the external contract (line 77) starts before effecting the update on internal states (line 80), hence violating the principle. In this particular case, if the external contract has some hidden logic that may be capable of launching re-entrancy via the very same entry function. Another function `withdrawLiquidityFromJob()` shares the same issue.

```

62  /**
63   * @notice withdraw funds after unbonding has finished
64   * @param _bonding the asset to withdraw from the bonding pool
65   */
66  function withdraw(address _bonding) external override nonReentrant {
67      if (canWithdrawAfter[msg.sender][_bonding] == 0) revert UnbondsUnexistent();
68      if (canWithdrawAfter[msg.sender][_bonding] >= block.timestamp) revert UnbondsLocked();
69      if (disputes[msg.sender]) revert Disputed();
70
71      uint256 _amount = pendingUnbonds[msg.sender][_bonding];
72
73      if (_bonding == keep3rV1) {
74          IKeep3rV1Proxy(keep3rV1Proxy).mint(_amount);
75      }
76
77      IERC20(_bonding).safeTransfer(msg.sender, _amount);
78
79      emit Withdrawal(msg.sender, _bonding, _amount);
80      pendingUnbonds[msg.sender][_bonding] = 0;
81  }

```

Listing 3.2: `Keep3rKeeperFundable::withdraw()`

**Recommendation** Apply necessary reentrancy guard or follow the checks-effects-interactions best practice.

**Status** This issue has been fixed in the commit: [2f3f209](#).

### 3.3 Possible FrontRunning DoS Against Job Credit Withdrawal

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Keep3r
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

#### Description

For each active job, the `Keep3r` protocol supports the addition (or removal) of so-called `credits` that may be used to pay out for work. Specifically, two public functions are provided `addTokenCreditsToJob`

() and `withdrawTokenCreditsFromJob()` to add or remove credits of a job, respectively. While examining these two functions, a possible denial-of-service situation may take place to block the job credits from being withdrawn for a certain time period.

To elaborate, we show below these two functions. When the credits are being added for a job, there is a job-associated state `jobTokenCreditsAddedAt` that is updated with the current time stamp. When the credits are being withdrawn, this state will be checked to ensure certain `_WITHDRAW_TOKENS_COOLDOWN` time period has passed. As a result, if an actor intends to maliciously send 1 `WEI` to the withdrawing user, the user may not be able to withdraw it in the next `_WITHDRAW_TOKENS_COOLDOWN` time period.

```

30  function addTokenCreditsToJob(
31      address _token,
32      address _job,
33      uint256 _amount
34  ) external override nonReentrant {
35      if (!_jobs.contains(_job)) revert JobUnavailable();
36      // KP3R shouldn't be used for direct token payments
37      if (_token == keep3rV1) revert TokenUnavailable();
38      uint256 _before = IERC20(_token).balanceOf(address(this));
39      IERC20(_token).safeTransferFrom(msg.sender, address(this), _amount);
40      uint256 _received = IERC20(_token).balanceOf(address(this)) - _before;
41      uint256 _fee = (_received * FEE) / BASE;
42      jobTokenCredits[_job][_token] += _received - _fee;
43      jobTokenCreditsAddedAt[_job][_token] = block.timestamp;
44      IERC20(_token).safeTransfer(governance, _fee);
45      _jobTokens[_job].add(_token);

47      emit AddCredit(_job, _token, msg.sender, block.number, _received);
48  }

50  function withdrawTokenCreditsFromJob(
51      address _token,
52      address _job,
53      uint256 _amount,
54      address _receiver
55  ) external override nonReentrant onlyJobOwner(_job) {
56      if (block.timestamp <= jobTokenCreditsAddedAt[_job][_token] +
57          _WITHDRAW_TOKENS_COOLDOWN) revert JobTokenCreditsLocked();
58      if (jobTokenCredits[_job][_token] < _amount) revert InsufficientJobTokenCredits();

59      jobTokenCredits[_job][_token] -= _amount;
60      IERC20(_token).safeTransfer(_receiver, _amount);

62      if (jobTokenCredits[_job][_token] == 0) {
63          _jobTokens[_job].remove(_token);
64      }

66      emit JobTokenCreditWithdrawal(_job, _token, _amount, msg.sender, _receiver,
        jobTokenCredits[_job][_token]);

```

67 }

Listing 3.3: `addTokenCreditsToJob()/withdrawTokenCreditsFromJob()`

**Recommendation** Revise the job credit addition/removal logic to ensure the process may not be exploited to block legitimate credit withdrawal attempt.

**Status** This issue has been confirmed and the team clarifies that the job owner can migrate the job in case the job with tokens is vulnerable.

### 3.4 Simplified Logic of `Keep3rJobFundableLiquidity::_phase()`

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Keep3rJobFundableLiquidity`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [2]

#### Description

The `Keep3r` protocol maintains job-associated states (e.g., credit and liquidity). The liquidity state may be used to compute certain rewards to a job. Among this process, there is an important notion `phase`, which is used to compute a fraction of the multiplier or the whole multiplier if equal or more than a `rewardPeriodTime` has passed.

To elaborate, we show below this `phase()` function. It implements a rather straightforward logic in computing the proportion of the given multiplier. However, the current implementation can be improved as the internal calculation of `_timePassed % rewardPeriodTime` (line 350) can be simplified as `_timePassed` under the condition of `if (_timePassed < rewardPeriodTime)` (line 349).

```

348 function _phase(uint256 _timePassed, uint256 _multiplier) internal view returns (
    uint256 _result) {
349     if (_timePassed < rewardPeriodTime) {
350         _result = ((_timePassed % rewardPeriodTime) * _multiplier) / rewardPeriodTime;
351     } else _result = _multiplier;
352 }
```

Listing 3.4: `Keep3rJobFundableLiquidity::phase()`

**Recommendation** Simplify the above `_phase()` logic by removing redundant computation.

**Status** This issue has been fixed in the commit: [2f3f209](#).



### 3.5 Improved Quote Calculation of Keep3rLibrary::getQuoteAtTick()

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Keep3rLibrary
- Category: Business Logic [7]
- CWE subcategory: CWE-837 [5]

#### Description

The Keep3r protocol has a built-in library `Keep3rLibrary` that is used to better interact with a number of `UniswapV3`-based liquidity pools. Within this library, there is a function `getQuoteAtTick()` that is used to calculate the amount of credits received in exchange when a tick and a token amount are given. Our analysis shows that this function can be improved to achieve better precision.

```

45  /// @notice Given a tick and a token amount, calculates the amount of credits received
    in exchange
46  /// @param baseAmount Amount of token to be converted
47  /// @param tickDifference Tick value used to calculate the quote
48  /// @param timeInterval Time value used to calculate the quote
49  /// @return _quoteAmount Amount of credits deserved for the baseAmount at the tick
    value
50  /* WARNING Uniswap's getQuoteAtTick was designed for a uint128 baseAmount */
51  function getQuoteAtTick(
52      uint256 baseAmount,
53      int56 tickDifference,
54      uint256 timeInterval
55  ) public pure returns (uint256 _quoteAmount) {
56      uint160 sqrtRatioX96 = getSqrtRatioAtTick(int24(tickDifference / int256(timeInterval
        )))
57      uint256 ratioX128 = mulDiv(sqrtRatioX96, sqrtRatioX96, 1 << 64);
58      _quoteAmount = mulDiv(1 << 128, baseAmount, ratioX128);
59  }
```

Listing 3.5: `Keep3rLibrary::getQuoteAtTick()`

To illustrate, we show above the `getQuoteAtTick()` function. If the intermediate `sqrtRatioX96`, when multiplied by itself, does not overflow, we can calculate the intended `quoteAmount` with better precision, i.e., `_quoteAmount = mulDiv(1 << 192, baseAmount, ratioX192)`. An example revision is shown as follows:

```

45  /// @notice Given a tick and a token amount, calculates the amount of credits received
    in exchange
46  /// @param baseAmount Amount of token to be converted
47  /// @param tickDifference Tick value used to calculate the quote
48  /// @param timeInterval Time value used to calculate the quote
```

```

49  /// @return _quoteAmount Amount of credits deserved for the baseAmount at the tick
    value
50  /* WARNING Uniswap's getQuoteAtTick was designed for a uint128 baseAmount */
51  function getQuoteAtTick(
52      uint256 baseAmount,
53      int56 tickDifference,
54      uint256 timeInterval
55  ) public pure returns (uint256 _quoteAmount) {
56      uint160 sqrtRatioX96 = TickMath.getSqrtRatioAtTick(tick);
57
58      // Calculate quoteAmount with better precision if it doesn't overflow when
        multiplied by itself
59      if (sqrtRatioX96 <= type(uint128).max) {
60          uint256 ratioX192 = uint256(sqrtRatioX96) * sqrtRatioX96;
61          _quoteAmount = mulDiv(1 << 192, baseAmount, ratioX192);
62      } else {
63          uint256 ratioX128 = mulDiv(sqrtRatioX96, sqrtRatioX96, 1 << 64);
64          _quoteAmount = mulDiv(1 << 128, baseAmount, ratioX128);
65      }
66  }

```

Listing 3.6: Keep3rLibrary::getQuoteAtTick()

**Recommendation** Improve the precision if possible in the above `getQuoteAtTick()` function.

**Status** This issue has been fixed in the commit: [2f3f209](#).

## 3.6 Improved Job Migration in Keep3rJobMigration

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Keep3rJobMigration
- Category: Business Logic [7]
- CWE subcategory: CWE-770 [4]

### Description

The Keep3r protocol supports an interesting feature, i.e., job migration. This feature is mainly implemented in two functions `migrateJob()` and `acceptJobMigration()`. The first function is designed to indicate the migration attempt, while the second function performs the actual migration after the `_MIGRATION_COOLDOWN` has passed. Our analysis shows that both functions can be improved by adjusting job-associated states due to the job migration.

```

15  function migrateJob(address _fromJob, address _toJob) external override onlyJobOwner(
    _fromJob) {
16      if (_fromJob == _toJob) revert JobMigrationImpossible();
17  }

```

```

18     pendingJobMigrations[_fromJob] = _toJob;
19     _migrationCreatedAt[_fromJob][_toJob] = block.timestamp;
20
21     emit JobMigrationRequested(_fromJob, _toJob);
22 }
23
24 function acceptJobMigration(address _fromJob, address _toJob) external override
    onlyJobOwner(_toJob) {
25     if (disputes[_fromJob] disputes[_toJob]) revert JobDisputed();
26     if (pendingJobMigrations[_fromJob] != _toJob) revert JobMigrationUnavailable();
27     if (block.timestamp < _migrationCreatedAt[_fromJob][_toJob] + _MIGRATION_COOLDOWN)
        revert JobMigrationLocked();
28
29     // force job credits update for both jobs
30     _settleJobAccountance(_fromJob);
31     _settleJobAccountance(_toJob);
32
33     // migrate tokens
34     while (_jobTokens[_fromJob].length() > 0) {
35         address _tokenToMigrate = _jobTokens[_fromJob].at(0);
36         jobTokenCredits[_toJob][_tokenToMigrate] += jobTokenCredits[_fromJob][
            _tokenToMigrate];
37         jobTokenCredits[_fromJob][_tokenToMigrate] = 0;
38         _jobTokens[_fromJob].remove(_tokenToMigrate);
39         _jobTokens[_toJob].add(_tokenToMigrate);
40     }
41
42     // migrate liquidities
43     while (_jobLiquidities[_fromJob].length() > 0) {
44         address _liquidity = _jobLiquidities[_fromJob].at(0);
45
46         liquidityAmount[_toJob][_liquidity] += liquidityAmount[_fromJob][_liquidity];
47         delete liquidityAmount[_fromJob][_liquidity];
48
49         _jobLiquidities[_toJob].add(_liquidity);
50         _jobLiquidities[_fromJob].remove(_liquidity);
51     }
52
53     // migrate job balances
54     _jobPeriodCredits[_toJob] += _jobPeriodCredits[_fromJob];
55     delete _jobPeriodCredits[_fromJob];
56
57     _jobLiquidityCredits[_toJob] += _jobLiquidityCredits[_fromJob];
58     delete _jobLiquidityCredits[_fromJob];
59
60     // stop _fromJob from being a job
61     delete rewardedAt[_fromJob];
62     _jobs.remove(_fromJob);
63
64     pendingJobMigrations[_fromJob] = address(0);
65     emit JobMigrationSuccessful(_fromJob, _toJob);

```

66 }

Listing 3.7: `Keep3rJobMigration::migrateJob()/acceptJobMigration()`

To elaborate, we show above these two functions. The first function can be improved by validating that the to-be-migrated job has not been canceled yet with the addition of the following requirement: `if (!_jobs.contains(_toJob)) revert JobUnavailable()`. The second function can be improved by removing those states associated with the migrated jobs, including `jobOwner`, `jobPendingOwner`, and `_migrationCreatedAt`.

**Recommendation** Improve the above-mentioned functions regarding the associated job states due to its migration.

**Status** This issue has been fixed in the commit: [2f3f209](#).

### 3.7 Possible DoS Against `Keep3rJobDisputable::slashJob()`

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Keep3rJobDisputable`
- Category: Business Logic [7]
- CWE subcategory: CWE-837 [5]

#### Description

Besides the job migration feature, the `Keep3r` protocol also supports the capability to slash a current job. Typically, to attract keepers for the intended job, each job has a set amount of credits that can be claimed upon the job is finished. However, certain situations may come up to slash a specific job. While reviewing the current logic for job slashing, we notice a potential denial-of-service issue to block the slashing behavior.

To elaborate, we show below the `slashJob()` function. This function iterates each job token and transfers all `jobTokenCredits` of the slashed job to the `governance`. In addition, all job-associated liquidities are also transferred to the `governance`. When the `jobTokenCredits` are taken away, the current implementation intentionally makes use of low level calls in order to avoid being reverted (lines 26-31). However, the job tokens can be dynamically added by any one without the need of being whitelisted. Therefore, a malicious actor may intentionally add a crafted token and transfer to the job as the part of job token credit. When the job is being slashed, the iteration of all job tokens invokes upon the crafted token contract, which may simply cause out-of-gas execution and thus fail the slashing operation.

```

16 function slashJob(address _job) external override nonReentrant onlySlasherOrGovernance
17 {
18     if (!disputes[_job]) revert NotDisputed();
19     // slash job tokens and token credits
20     uint256 _index = 0;
21     while (_index < _jobTokens[_job].length()) {
22         address _token = _jobTokens[_job].at(_index);
23
24         // make low level call in order to avoid reverting
25         // solhint-disable-next-line avoid-low-level-calls
26         try IERC20(_token).transfer(governance, jobTokenCredits[_job][_token]) {
27             jobTokenCredits[_job][_token] = 0;
28             _jobTokens[_job].remove(_token);
29         } catch {
30             _index++;
31         }
32     }
33
34     // slash job liquidities
35     while (_jobLiquidities[_job].length() > 0) {
36         address _liquidity = _jobLiquidities[_job].at(0);
37
38         IERC20(_liquidity).safeTransfer(governance, liquidityAmount[_job][_liquidity]);
39         liquidityAmount[_job][_liquidity] = 0;
40         _jobLiquidities[_job].remove(_liquidity);
41     }
42
43     // slash job liquidity credits
44     _jobLiquidityCredits[_job] = 0;
45     _jobPeriodCredits[_job] = 0;
46     disputes[_job] = false;
47
48     // emit event
49     emit JobSlash(_job);
50 }

```

Listing 3.8: Keep3rJobDisputable::slashJob()

The same issue is also applicable to another routine `slashTokenFromJob()`, which the given token is a crafted one.

**Recommendation** Apply a whitelist on the set of tokens that may be accepted as job tokens.

**Status** This issue has been fixed in the commit: [2f3f209](#).

## 4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Keep3r` (v2) protocol, a decentralized keeper network for projects that need external devops and for external teams to find keeper jobs. The keeper network presents an interesting and novel approach to instantiate the dynamic market for keepers and we are very impressed by the elegant design and clean implementation. We have identified several issues related to either security or performance 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.



## References

- [1] MITRE. CWE-1041: Use of Redundant Code. <https://cwe.mitre.org/data/definitions/1041.html>.
- [2] MITRE. CWE-563: Assignment to Variable without Use. <https://cwe.mitre.org/data/definitions/563.html>.
- [3] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. <https://cwe.mitre.org/data/definitions/663.html>.
- [4] MITRE. CWE-770: Allocation of Resources Without Limits or Throttling. <https://cwe.mitre.org/data/definitions/770.html>.
- [5] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. <https://cwe.mitre.org/data/definitions/837.html>.
- [6] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [7] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [8] MITRE. CWE CATEGORY: Concurrency. <https://cwe.mitre.org/data/definitions/557.html>.
- [9] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.

- [10] OWASP. Risk Rating Methodology. [https://www.owasp.org/index.php/OWASP\\_Risk\\_Rating\\_Methodology](https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology).
- [11] PeckShield. PeckShield Inc. <https://www.peckshield.com>.
- [12] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. <https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09>.
- [13] David Siegel. Understanding The DAO Attack. <https://www.coindesk.com/understanding-dao-hack-journalists>.

