SMART CONTRACT AUDIT REPORT

for

KEEP3R NETWORK

Prepared By: Shuxiao Wang

Hangzhou, China

November 1, 2020

Document Properties

Client	Keep3r Network
Title	Smart Contract Audit Report
Target	Keep3r
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Jeff Liu
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 1, 2020	Xuxian Jiang	Final Release
0.2	October 30, 2020	Xuxian Jiang	Additional Findings
0.1	October 27, 2020	Xuxian Jiang	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 design document and related smart contract source code of **Keep3r**, 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 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
Issuer	Keep3r Network
Website	https://keep3r.network/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 1, 2020

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Keep3r assumes a trusted oracle with timely fastgas-related price feeds and the oracle itself is not part of this audit.

- https://github.com/keep3r-network/keep3r.network.git (e974e43)

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

- https://github.com/keep3r-network/keep3r.network.git (501dc9d)

1.2 About PeckShield

PeckShield Inc. [20] 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

		Likelihood		
		High	*Medium*	*Low*
Impact	*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 [15]:

- 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 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) [14], 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-based assessment 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 investment advice.

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

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 design and implementation of Keep3r. 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	2	■■
Medium	5	■■■■■■
Low	5	■■■■■
Informational	6	■■■■■■
Total	18	

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 high-severity vulnerabilities, 5 medium-severity vulnerabilities, 5 low-severity vulnerabilities, and 6 informational recommendations.

Table 2.1: Key Keep3r Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Potential Reentrancy Risks in User-Facing Functions	Time and State	Fixed
PVE-002	Low	Validation of transferFrom() Return Values	Coding Practices	Fixed
PVE-003	Low	Simplification of the bond() Logic	Coding Practices	Fixed
PVE-004	Medium	Business Logic Error in _bond()	Business Logics	Fixed
PVE-005	Medium	Inaccurate Credit Attribution in receipt()	Business Logics	Fixed
PVE-006	Medium	The Apples And Oranges Issues in totalBonded	Business Logics	Fixed
PVE-007	Medium	Inaccurate Credit Calculation in applyCreditToJob()	Business Logics	Fixed
PVE-008	Medium	Inaccurate Credit Deduction in unbondLiquidityFromJob()	Business Logics	Fixed
PVE-009	Informational	Unused Code Removal	Coding Practices	Fixed
PVE-010	Informational	Avoidance of Repeated Calculation of domainSeparator	Coding Practices	Fixed
PVE-011	Informational	Consistency Between Function Definitions And Return Statements	Coding Practices	Fixed
PVE-012	High	Flashloan/Sandwich Attacks For Job Credit Manipulation	Time and State	Mitigated
PVE-013	Informational	Improved Events in SubmitJob/UnbondJob/RemoveJob	Error Conditions, Return Values, Status Codes	Fixed
PVE-014	Low	Improved Handling of Corner Cases in Proposal Submissions	Business Logics	Fixed
PVE-015	High	Necessity Of Non-Public Functions For Time-locked Transactions	Coding Practices	Fixed
PVE-016	Informational	Full Charge of Proposal Execution Cost From Accompanying msg.value	Business Logics	Confirmed
PVE-017	Low	Improved Sanity Checks For System Parameters	Coding Practices	Fixed
PVE-018	Informational	Incompatibility With Deflationary/Rebasing Tokens	Business Logics	Fixed

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 Potential Reentrancy Risks in User-Facing Functions

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Keep3r`
- Category: Time and State [12]
- CWE subcategory: CWE-663 [6]

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` [23] exploit, and the recent `Uniswap/Lendf.Me` hack [21].

We notice there are several occasions the `checks-effects-interactions` principle is violated. Using the `Keep3r` as an example, the `addCredit()` 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 540) starts before effecting the update on internal states (line 542), 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 `addCredit()` function.

```
531    /**
532     * @notice Add credit to a job to be paid out for work
533     * @param credit the credit being assigned to the job
534     * @param job the job being credited
535     * @param amount the amount of credit being added to the job
```

```
     */
    function addCredit(address credit, address job, uint amount) external {
        require(jobs[job], "addCreditETH: !job");
        uint _before = ERC20(credit).balanceOf(address(this));
        ERC20(credit).transferFrom(msg.sender, address(this), amount);
        uint _after = ERC20(credit).balanceOf(address(this));
        credits[job][credit] = credits[job][credit].add(_after.sub(_before));

        emit AddCredit(credit, job, msg.sender, block.number, _after.sub(_before));
    }
```

Listing 3.1: Keep3r.sol

Another similar violation can be found in other routines within the same contract, including `bond()`, `addLiquidityToJob()`, `withdraw()`, and `slash()`.

In the meantime, we should mention that the `UniswapV2`'s LP tokens and most standard ERC20 tokens implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`.

Recommendation Apply necessary reentrancy guard or follow the `checks-effects-interactions` best practice. The above `addCredit()` function can be revised as follows:

```
    /**
     * @notice Add credit to a job to be paid out for work
     * @param credit the credit being assigned to the job
     * @param job the job being credited
     * @param amount the amount of credit being added to the job
     */
    function addCredit(address credit, address job, uint amount) external nonReentrant {
        require(jobs[job], "addCreditETH: !job");
        uint _before = ERC20(credit).balanceOf(address(this));
        ERC20(credit).transferFrom(msg.sender, address(this), amount);
        uint _after = ERC20(credit).balanceOf(address(this));
        credits[job][credit] = credits[job][credit].add(_after.sub(_before));

        emit AddCredit(credit, job, msg.sender, block.number, _after.sub(_before));
    }
```

Listing 3.2: Keep3r.sol (revised)

Status This issue has been fixed in the commit: a8af8ec2e4ab2b0b119de8320663519ab1fbea23.

3.2 Validation of transferFrom() Return Values

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Keep3r`
- Category: Coding Practices [10]
- CWE subcategory: CWE-1041 [2]

Description

One core functionality provided in the `Keep3r` contract is the bonding process for keeper onboarding. Specifically, to join the keeper network as a keeper, a user needs to follow the protocol by firstly calling `bond()` on the `Keep3r` contract. (There is no need to have the governance tokens, i.e., `KPR`.) There is a 3 day bonding delay for the registered user to `activate()` as a true keeper.

To elaborate, we show below the code snippet of `bond()`. This routine is designed to set up the bonding activation clock and transfer the bonding assets with the specified amount to the contract.

```
888     /**
889      * @notice begin the bonding process for a new keeper
890      * @param bonding the asset being bound
891      * @param amount the amount of bonding asset being bound
892      */
893     function bond(address bonding, uint amount) external {
894         require(pendingbonds[msg.sender][bonding] == 0, "bond: bonding");
895         require(!blacklist[msg.sender], "bond: blacklisted");
896         bondings[msg.sender][bonding] = now.add(BOND);
897         if (bonding == address(this)) {
898             _transferTokens(msg.sender, address(this), amount);
899         } else {
900             ERC20(bonding).transferFrom(msg.sender, address(this), amount);
901         }
902         pendingbonds[msg.sender][bonding] = pendingbonds[msg.sender][bonding].add(amount
                );
903         emit KeeperBonding(msg.sender, block.number, bondings[msg.sender][bonding],
                amount);
904     }
```

Listing 3.3: Keep3r.sol

While reviewing the logic, we notice that when the assets are transferred to the contract, it is handled with the normal `transferFrom()` (line 900) that does not validate the return value. To better accommodate various idiosyncrasies associated with different implementations/customizations of ERC20 tokens, we strongly suggest to replace all occurrences of `transferFrom()` with the safe version of `safeTransferFrom()` from `OpenZeppelin`.

The issue is also applicable to other two unsafe transfers in the same contract, i.e., `addCredit()` and `addLiquidityToJob()`.

Recommendation Replace all occurrences of `transferFrom()` with the safe version of `safeTransferFrom()` from `OpenZeppelin`. Similarly, replace unsafe `transfer()`, if any, with `safeTransfer()` as well.

Status This issue has been fixed in the commit: 4ab4562da7f9e7de8655195af3a5469014863065.

3.3 Simplification of the bond() Logic

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Keep3r`
- Category: Coding Practices [10]
- CWE subcategory: CWE-1041 [2]

Description

As mentioned in Section 3.2, the keeper onboarding process requires the new keeper to deposit certain bonding assets via the `bond()` function. For illustration, we show below the code snippet of `bond()`. The routine basically performs two operations: the first one sets up the bonding activation clock and the second one transfers the bonding assets with the specified amount to the contract.

```
888     /**
889      * @notice begin the bonding process for a new keeper
890      * @param bonding the asset being bound
891      * @param amount the amount of bonding asset being bound
892      */
893     function bond(address bonding, uint amount) external {
894         require(pendingbonds[msg.sender][bonding] == 0, "bond: bonding");
895         require(!blacklist[msg.sender], "bond: blacklisted");
896         bondings[msg.sender][bonding] = now.add(BOND);
897         if (bonding == address(this)) {
898             _transferTokens(msg.sender, address(this), amount);
899         } else {
900             ERC20(bonding).transferFrom(msg.sender, address(this), amount);
901         }
902         pendingbonds[msg.sender][bonding] = pendingbonds[msg.sender][bonding].add(amount
            );
903         emit KeeperBonding(msg.sender, block.number, bondings[msg.sender][bonding],
            amount);
904     }
```

Listing 3.4: Keep3r.sol

The optimization is related to the `pendingbonds` update (line 902): `pendingbonds[msg.sender][bonding] = pendingbonds[msg.sender][bonding].add(amount)`. Recall the requirement at line 894 of `pendingbonds[msg.sender][bonding] == 0`, we can simplify the logic to `pendingbonds[msg.sender][bonding] = amount`. Another option is to remove this requirement and allows the onboarding keeper to make repeated calls to `bond()`.

Recommendation Revise the `bond()` logic as follows:

```
    /**
     * @notice begin the bonding process for a new keeper
     * @param bonding the asset being bound
     * @param amount the amount of bonding asset being bound
     */
    function bond(address bonding, uint amount) external {
        require(pendingbonds[msg.sender][bonding] == 0, "bond: bonding");
        require(!blacklist[msg.sender], "bond: blacklisted");
        bondings[msg.sender][bonding] = now.add(BOND);
        if (bonding == address(this)) {
            _transferTokens(msg.sender, address(this), amount);
        } else {
            ERC20(bonding).transferFrom(msg.sender, address(this), amount);
        }
        pendingbonds[msg.sender][bonding] = amount;
        emit KeeperBonding(msg.sender, block.number, bondings[msg.sender][bonding],
            amount);
    }
```

Listing 3.5: Keep3r.sol (revised)

Status This issue has been fixed in the commit: ace06c753c8101802a0606c25674d1d01a13cf41.

3.4 Business Logic Error in _bond()

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Keep3r`
- Category: Business Logics [11]
- CWE subcategory: CWE-837 [9]

Description

The keeper onboarding process starts from the keeper calling the `bond()`, which further delegates to the internal `_bond()` routine for bookkeeping. The bookkeeping involves the total bonded amount as well as the bonded amount from each keeper. If the bonded asset is `KPR`, the bonded amount will add the keeper's voting power.

While reviewing the internal record of bookkeeping, we notice an issue in its logic. Specifically, as shown in the following code snippet of `_bond()`, the voting power is updated at line 776 if the bonded asset is `KPR`. However, the condition of determining whether the bonded asset is `KPR` is evaluated as `if (_from == address(this))...`, which needs to be adjusted as `if (bonding == address(this))...`.

```
772     function _bond(address bonding, address _from, uint _amount) internal {
773         bonds[_from][bonding] = bonds[_from][bonding].add(_amount);
774         totalBonded = totalBonded.add(_amount);
775         if (_from == address(this)) {
776             _moveDelegates(address(0), delegates[_from], _amount);
777         }
778     }
```

Listing 3.6: Keep3r.sol

Recommendation Adjust the bonding logic to properly credit the voting power to the onboarding keepers. An example revision is shown below. Note that it also contains another PVE006-related revision regarding the calculation of total bonded amount - `totalBonded` (line 776).

```
772     function _bond(address bonding, address _from, uint _amount) internal {
773         bonds[_from][bonding] = bonds[_from][bonding].add(_amount);
774         if (bonding == address(this)) {
775             totalBonded = totalBonded.add(_amount);
776             _moveDelegates(address(0), delegates[_from], _amount);
777         }
778     }
```

Listing 3.7: Keep3r.sol (revised)

Status This issue has been fixed in the commit: 990517dc928909fd65a0dd33fa329ffe8242b515.

3.5 Inaccurate Credit Attribution in receipt()

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Keep3r`
- Category: Business Logics [11]
- CWE subcategory: CWE-837 [9]

Description

After a keeper is successfully registered, the keeper can fulfill the jobs requested by different projects. Each job has a set amount of credits that they can award keepers with.

While reviewing the credit-attribution logic, we notice an issue in one of three receipt-related reward functions. Specifically, these three functions are `workReceipt()`, `receiptETH()`, and `receipt()`. The main difference among these functions is the assets used for credit attribution: The first function uses the governance token KPR and the second one uses ETH while the last one works for all ERC20-compliant tokens.

The first two functions properly attribute the credit to the keeper who performed the work. However, the last function mis-attributes the credit to msg.sender, not the intended keeper.

```
    /**
     * @notice Implemented by jobs to show that a keeper performed work
     * @param credit the asset being awarded to the keeper
     * @param keeper address of the keeper that performed the work
     * @param amount the reward that should be allocated
     */
    function receipt(address credit, address keeper, uint amount) external {
        require(jobs[msg.sender], "receipt: !job");
        credits[msg.sender][credit] = credits[msg.sender][credit].sub(amount, "
            workReceipt: insufficient funds");
        lastJob[keeper] = now;
        ERC20(credit).transfer(msg.sender, amount);
        emit KeeperWorked(credit, msg.sender, keeper, block.number);
    }
```

Listing 3.8: Keep3r.sol

Recommendation Make necessary changes in the attribution logic to properly credit the keeper with the reward amount. An example revision is shown below.

```
    /**
     * @notice Implemented by jobs to show that a keeper performed work
     * @param credit the asset being awarded to the keeper
     * @param keeper address of the keeper that performed the work
     * @param amount the reward that should be allocated
     */
    function receipt(address credit, address keeper, uint amount) external {
        require(jobs[msg.sender], "receipt: !job");
        credits[msg.sender][credit] = credits[msg.sender][credit].sub(amount, "
            workReceipt: insufficient funds");
        lastJob[keeper] = now;
        ERC20(credit).transfer(keeper, amount);
        emit KeeperWorked(credit, msg.sender, keeper, block.number);
    }
```

Listing 3.9: Keep3r.sol (revised)

Status This issue has been fixed in the commit: 27142cde57635400501ac1a5034e5416800ce567.

3.6 The Apples And Oranges Issues in totalBonded

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Keep3r
- Category: Business Logics [11]
- CWE subcategory: CWE-770 [8]

Description

Following the discussion in Section 3.4, the keeper onboarding process requires the invocation of bond (), which internally calls the _bond() helper routine. While it requires transferring bonded assets into the contract, there is no need to have the governance tokens, i.e., KPR. If fact, any ERC20-compliant token can serve the purpose. Note that there is a 3 day bonding delay for the registered user to activate() as a true keeper.

Because of the rather loose requirement on the required bonding assets, different users may call bond() with different bonding assets. However, these assets are simply added into the same (single) state to keep track of the total bonded amount. We believe two amounts from two different bonding assets cannot be simply added to represent the total bonded amount. The related code snippet in _bond() is shown below:

```
    function _bond(address bonding, address _from, uint _amount) internal {
        bonds[_from][bonding] = bonds[_from][bonding].add(_amount);
        totalBonded = totalBonded.add(_amount);
        if (_from == address(this)) {
            _moveDelegates(address(0), delegates[_from], _amount);
        }
    }
```

Listing 3.10: Keep3r.sol

There are two possible solutions: One is to take KPR into account as total bonded amount, and the other is to maintain totalBonded as an array to differentiate the bonded amount per asset.

Recommendation Avoid the mixed calculation of the total bonded amount in _bond(). An example revision (combined with PVE004 fixup) is shown below:

```
    function _bond(address bonding, address _from, uint _amount) internal {
        bonds[_from][bonding] = bonds[_from][bonding].add(_amount);
        if (bonding == address(this)) {
            totalBonded = totalBonded.add(_amount);
            _moveDelegates(address(0), delegates[_from], _amount);
        }
    }
```

Listing 3.11: Keep3r.sol (revised)

Status This issue has been fixed in the commit: 990517dc928909fd65a0dd33fa329ffe8242b515.

3.7 Inaccurate Credit Calculation in applyCreditToJob()

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Keep3r`
- Category: Business Logics [11]
- CWE subcategory: CWE-837 [9]

Description

As mentioned earlier, a job can be any system that requires external execution, the scope of Keep3r is not to define or restrict the action taken, but to create an incentive mechanism for all parties involved.

There are two core ways to create a job in Keep3r. The first approach is to submit a proposal via the built-in `Governance`, including the contract as a job. (If governance approves, no further steps are required.) The second approach is to call `addLiquidityToJob()` on the Keep3r contract. To attract keepers for the intended job, each job has a set amount of credits that can be claimed upon the job is finished.

While reviewing possible ways to recharge the credit amount to a job, we notice the logic in one main function `applyCreditToJob()` needs to be revised. To elaborate, we show the routine's code snippet below.

```
    /**
     * @notice Applies the credit provided in addLiquidityToJob to the job
     * @param provider the liquidity provider
     * @param liquidity the pair being added as liquidity
     * @param job the job that is receiving the credit
     */
    function applyCreditToJob(address provider, address liquidity, address job) external
        {
        require(liquidityAccepted[liquidity], "addLiquidityToJob: !pair");
        require(liquidityApplied[provider][liquidity][job] != 0, "credit: no bond");
        require(liquidityApplied[provider][liquidity][job] < now, "credit: bonding");
        uint _liquidity = balances[address(liquidity)];
        uint _credit = _liquidity.mul(liquidityAmount[msg.sender][liquidity][job]).div(
            ERC20(liquidity).totalSupply());
        _mint(address(this), _credit);
        credits[job][address(this)] = credits[job][address(this)].add(_credit);
        liquidityAmount[msg.sender][liquidity][job] = 0;

        emit ApplyCredit(job, msg.sender, block.number, _credit);
```

```
647     }
```

Listing 3.12: Keep3r.sol

The recharged credit does not come from the purchased `KPR` tokens. Instead, it is calculated from the provided `KPR-WETH` liquidity in `Uniswap`. By doing so, you can get an amount of credits equal to the amount of `KPR` tokens in the liquidity you provide. However, there exists an issue in the credit calculation (line 641): Currently, the credit is determined as `_liquidity.mul(liquidityAmount[msg.sender][liquidity][job]).div(ERC20(liquidity).totalSupply())`, which calculates the share from `msg.sender`, not the liquidity provider. A correct calculation should be the following: `_liquidity.mul(liquidityAmount[provider][liquidity][job]).div(ERC20(liquidity).totalSupply())`.

Recommendation Modify the `applyCreditToJob` logic to properly apply the liquidity credit to the specified job. An example revision is shown below.

```
630     /**
631      * @notice Applies the credit provided in addLiquidityToJob to the job
632      * @param provider the liquidity provider
633      * @param liquidity the pair being added as liquidity
634      * @param job the job that is receiving the credit
635      */
636     function applyCreditToJob(address provider, address liquidity, address job) external
             {
637         require(liquidityAccepted[liquidity], "addLiquidityToJob: !pair");
638         require(liquidityApplied[provider][liquidity][job] != 0, "credit: no bond");
639         require(liquidityApplied[provider][liquidity][job] < now, "credit: bonding");
640         uint _liquidity = balances[address(liquidity)];
641         uint _credit = _liquidity.mul(liquidityAmount[provider][liquidity][job]).div(
                ERC20(liquidity).totalSupply());
642         _mint(address(this), _credit);
643         credits[job][address(this)] = credits[job][address(this)].add(_credit);
644         liquidityAmount[provider][liquidity][job] = 0;
645 
646         emit ApplyCredit(job, msg.sender, block.number, _credit);
647     }
```

Listing 3.13: Keep3r.sol (revised)

Status This issue has been fixed in the commit: c5391f1ac8465e65ab15637978302e331ca29840.

3.8 Inaccurate Credit Deduction in unbondLiquidityFromJob()

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Keep3r`
- Category: Business Logics [11]
- CWE subcategory: CWE-837 [9]

Description

In Section 3.7, we have discussed the ways to contribute to a job's credit. Naturally, there is a de-contribute support in unbonding the provided liquidity from a job.

While reviewing possible ways to unbond the liquidity from a job, we notice the logic in the handler function `unbondLiquidityFromJob()` needs to be revised. To elaborate, we show the routine's code snippet below.

```
    /**
     * @notice Unbond liquidity for a job
     * @param liquidity the pair being unbound
     * @param job the job being unbound from
     * @param amount the amount of liquidity being removed
     */
    function unbondLiquidityFromJob(address liquidity, address job, uint amount)
        external {
        require(liquidityAmount[msg.sender][liquidity][job] == 0, "credit: pending
            credit");
        liquidityUnbonding[msg.sender][liquidity][job] = now.add(UNBOND);
        liquidityAmountsUnbonding[msg.sender][liquidity][job] =
            liquidityAmountsUnbonding[msg.sender][liquidity][job].add(amount);
        require(liquidityAmountsUnbonding[msg.sender][liquidity][job] <=
            liquidityProvided[msg.sender][liquidity][job], "unbondLiquidityFromJob:
            insufficient funds");

        uint _liquidity = balances[address(liquidity)];
        uint _credit = _liquidity.mul(amount).div(ERC20(liquidity).totalSupply());
        if (_credit > credits[job][address(this)]) {
            _burn(address(this), credits[job][address(this)]);
            credits[job][address(this)] = 0;
        } else {
            _burn(address(this), _credit);
            credits[job][address(this)].sub(_credit);
        }

        emit UnbondJob(job, msg.sender, block.number, liquidityProvided[msg.sender][
            liquidity][job]);
    }
```

Listing 3.14: Keep3r.sol

The unbonded credit is calculated from the provided `KPR-WETH` liquidity in `Uniswap`. By doing so, you can get an amount of credits equal to the amount of `KPR` tokens in the liquidity you provide. However, there exists an issue that does not reduce the job's credit in `credits[job][address(this)]` (line 668). The proper reduction needs to be `credits[job][address(this)] = credits[job][address(this)].sub(_credit)`.

Recommendation Modify the `unbondLiquidityFromJob` logic to properly remove the liquidity credit from the specified job. An example revision is shown below.

```
649     /**
650      * @notice Unbond liquidity for a job
651      * @param liquidity the pair being unbound
652      * @param job the job being unbound from
653      * @param amount the amount of liquidity being removed
654      */
655     function unbondLiquidityFromJob(address liquidity, address job, uint amount)
            external {
656         require(liquidityAmount[msg.sender][liquidity][job] == 0, "credit: pending
                credit");
657         liquidityUnbonding[msg.sender][liquidity][job] = now.add(UNBOND);
658         liquidityAmountsUnbonding[msg.sender][liquidity][job] =
                liquidityAmountsUnbonding[msg.sender][liquidity][job].add(amount);
659         require(liquidityAmountsUnbonding[msg.sender][liquidity][job] <=
                liquidityProvided[msg.sender][liquidity][job], "unbondLiquidityFromJob:
                insufficient funds");
660
661         uint _liquidity = balances[address(liquidity)];
662         uint _credit = _liquidity.mul(amount).div(IERC20(liquidity).totalSupply());
663         if (_credit > credits[job][address(this)]) {
664             _burn(address(this), credits[job][address(this)]);
665             credits[job][address(this)] = 0;
666         } else {
667             _burn(address(this), _credit);
668             credits[job][address(this)] = credits[job][address(this)].sub(_credit);
669         }
670
671         emit UnbondJob(job, msg.sender, block.number, amount);
672     }
```

Listing 3.15: Keep3r.sol (revised)

Status This issue has been fixed in the commit: 4af0a610f1d20663cfd2ee5d7d098f0ee43a6911.

3.9 Unused Code Removal

- ID: PVE-009
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Keep3r
- Category: Coding Practices [10]
- CWE subcategory: CWE-563 [5]

Description

Keep3r makes good use of a number of reference contracts, such as ERC20, SafeERC20, and SafeMath, to facilitate its code implementation and organization. For example, the Keep3r smart contract has so far imported at least four reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the states defined in the Keep3r contract, there is an array named work that is not used anymore. This array was apparently used to track the amount of job executions for a keeper, which is currently deprecated or not used anymore.

```
    /// @notice tracks if a keeper has a pending dispute
    mapping(address => bool) public disputes;

    /// @notice tracks last job performed for a keeper
    mapping(address => uint) public lastJob;
    /// @notice tracks the amount of job executions for a keeper
    mapping(address => uint) public work;
    /// @notice tracks the total job executions for a keeper
    mapping(address => uint) public workCompleted;
    /// @notice list of all jobs registered for the keeper system
    mapping(address => bool) public jobs;
```

Listing 3.16: Keep3r.sol

Recommendation Consider the removal of the unused code.

Status This issue has been fixed in the commit: f0d5cd1a8ac27b89ec9f7f7180a62efe1a8b0e7a.

3.10 Avoidance of Repeated Calculation of domainSeparator

- ID: PVE-010
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Keep3r
- Category: Coding Practices [10]
- CWE subcategory: CWE-563 [5]

Description

The Keep3r contract implements the EIP2612 specification by providing the permit() support. This specification is proposed to address a limiting factor in the earlier ERC20 design where the ERC20 approve() function itself is defined in terms of msg.sender. This EIP-2612 specification basically extends the ERC20 standard with a new function permit(), which allows users to modify the allowance mapping using a signed message, instead of through msg.sender.

To elaborate, we show the permit() implementation.

```
    /**
     * @notice Triggers an approval from owner to spends
     * @param owner The address to approve from
     * @param spender The address to be approved
     * @param amount The number of tokens that are approved (2^256-1 means infinite)
     * @param deadline The time at which to expire the signature
     * @param v The recovery byte of the signature
     * @param r Half of the ECDSA signature pair
     * @param s Half of the ECDSA signature pair
     */
    function permit(address owner, address spender, uint amount, uint deadline, uint8 v,
         bytes32 r, bytes32 s) external {
        bytes32 domainSeparator = keccak256(abi.encode(DOMAIN_TYPEHASH, keccak256(bytes(
            name)), _getChainId(), address(this)));
        bytes32 structHash = keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender,
            amount, nonces[owner]++, deadline));
        bytes32 digest = keccak256(abi.encodePacked("\x19\x01", domainSeparator,
            structHash));
        address signatory = ecrecover(digest, v, r, s);
        require(signatory != address(0), "permit: signature");
        require(signatory == owner, "permit: unauthorized");
        require(now <= deadline, "permit: expired");

        allowances[owner][spender] = amount;

        emit Approval(owner, spender, amount);
    }
```

Listing 3.17: Keep3r.sol

In the same vein, the contract also supports `delegateBySig()` that avoids the `delegate()` call to be initiated from `msg.sender`. We notice that both re-calculate the same `domainSeparator`. To avoid unnecessary repeated calculation, we suggest to move this calculation to the contract's `constructor()` at the time when the contract is deployed.

```
    /**
     * @notice Delegates votes from signatory to `delegatee`
     * @param delegatee The address to delegate votes to
     * @param nonce The contract state required to match the signature
     * @param expiry The time at which to expire the signature
     * @param v The recovery byte of the signature
     * @param r Half of the ECDSA signature pair
     * @param s Half of the ECDSA signature pair
     */
    function delegateBySig(address delegatee, uint nonce, uint expiry, uint8 v, bytes32
        r, bytes32 s) public {
        bytes32 domainSeparator = keccak256(abi.encode(DOMAIN_TYPEHASH, keccak256(bytes(
            name)), _getChainId(), address(this)));
        bytes32 structHash = keccak256(abi.encode(DELEGATION_TYPEHASH, delegatee, nonce,
             expiry));
        bytes32 digest = keccak256(abi.encodePacked("\x19\x01", domainSeparator,
            structHash));
        address signatory = ecrecover(digest, v, r, s);
        require(signatory != address(0), "delegateBySig: sig");
        require(nonce == nonces[signatory]++, "delegateBySig: nonce");
        require(now <= expiry, "delegateBySig: expired");
        return _delegate(signatory, delegatee);
    }
```

Listing 3.18: Keep3r.sol

Recommendation Calculate `domainSeparator` once in the `constructor()` and reference it later without repeated calculations.

Status This issue has been fixed in the commit: 8a9db4f506f83f95eaeb2b1db7d83ced6837737b.

3.11 Consistency Between Function Definitions And Return Statements

- ID: PVE-011
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Keep3r`, `Keep3rGovernance`
- Category: Coding Practices [10]
- CWE subcategory: CWE-1041 [2]

Description

In the `Keep3r` contract, the `delegate()/delegateBySig()` functions are used to delegate votes from `msg.sender` or an intended signatory to the given `delegatee`. As shown at line 264 in the following code snippet, the `delegate()` function returns by calling the internal routine `_delegate()`, and the `return` keyword is not needed as the function is declared without any return value.

```
259     /**
260      * @notice Delegate votes from `msg.sender` to `delegatee`
261      * @param delegatee The address to delegate votes to
262      */
263     function delegate(address delegatee) public {
264         return _delegate(msg.sender, delegatee);
265     }

267     /**
268      * @notice Delegates votes from signatory to `delegatee`
269      * @param delegatee The address to delegate votes to
270      * @param nonce The contract state required to match the signature
271      * @param expiry The time at which to expire the signature
272      * @param v The recovery byte of the signature
273      * @param r Half of the ECDSA signature pair
274      * @param s Half of the ECDSA signature pair
275      */
276     function delegateBySig(address delegatee, uint nonce, uint expiry, uint8 v, bytes32
            r, bytes32 s) public {
277         bytes32 domainSeparator = keccak256(abi.encode(DOMAIN_TYPEHASH, keccak256(bytes(
                name)), _getChainId(), address(this)));
278         bytes32 structHash = keccak256(abi.encode(DELEGATION_TYPEHASH, delegatee, nonce,
                 expiry));
279         bytes32 digest = keccak256(abi.encodePacked("\x19\x01", domainSeparator,
                structHash));
280         address signatory = ecrecover(digest, v, r, s);
281         require(signatory != address(0), "delegateBySig: sig");
282         require(nonce == nonces[signatory]++, "delegateBySig: nonce");
283         require(now <= expiry, "delegateBySig: expired");
284         return _delegate(signatory, delegatee);
```

```
    }
```

Listing 3.19: Keep3r.sol

The same issue is also applicable for the `delegateBySig()` function.

Recommendation Remove the `return` keyword in the above two functions. An example revision is shown as follows.

```
    /**
     * @notice Delegate votes from `msg.sender` to `delegatee`
     * @param delegatee The address to delegate votes to
     */
    function delegate(address delegatee) public {
        _delegate(msg.sender, delegatee);
    }

    /**
     * @notice Delegates votes from signatory to `delegatee`
     * @param delegatee The address to delegate votes to
     * @param nonce The contract state required to match the signature
     * @param expiry The time at which to expire the signature
     * @param v The recovery byte of the signature
     * @param r Half of the ECDSA signature pair
     * @param s Half of the ECDSA signature pair
     */
    function delegateBySig(address delegatee, uint nonce, uint expiry, uint8 v, bytes32
        r, bytes32 s) public {
        bytes32 domainSeparator = keccak256(abi.encode(DOMAIN_TYPEHASH, keccak256(bytes(
            name)), _getChainId(), address(this)));
        bytes32 structHash = keccak256(abi.encode(DELEGATION_TYPEHASH, delegatee, nonce,
             expiry));
        bytes32 digest = keccak256(abi.encodePacked("\x19\x01", domainSeparator,
            structHash));
        address signatory = ecrecover(digest, v, r, s);
        require(signatory != address(0), "delegateBySig: sig");
        require(nonce == nonces[signatory]++, "delegateBySig: nonce");
        require(now <= expiry, "delegateBySig: expired");
        _delegate(signatory, delegatee);
    }
```

Listing 3.20: Keep3r.sol (revised)

Status This issue has been fixed in the commit: 83f54c2ed99ee3ead58e5101fe33e876ec23b189.

3.12 Flashloan/Sandwich Attacks For Job Credit Manipulation

- ID: PVE-012
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Keep3r`
- Category: Time and State [12]
- CWE subcategory: CWE-663 [6]

Description

A unique feature implemented in Keep3r is the support of liquidity credit for an active job. In particular, a liquidity provider to the `Uniswap` pool of `KPR-WETH` can be credited to bond the pool token to `Keep3r`. By doing so, the liquidity provider has the credit that equals to the amount of `KPR` tokens in the liquidity you provide (Section 3.7).

In the following, we show the code snippet of the `applyCreditToJob()` routine. This routine is responsible for calculating the due credit from the provided pool token and assigning the credit to the intended job.

```
    /**
     * @notice Applies the credit provided in addLiquidityToJob to the job
     * @param provider the liquidity provider
     * @param liquidity the pair being added as liquidity
     * @param job the job that is receiving the credit
     */
    function applyCreditToJob(address provider, address liquidity, address job) external
         {
        require(liquidityAccepted[liquidity], "addLiquidityToJob: !pair");
        require(liquidityApplied[provider][liquidity][job] != 0, "credit: no bond");
        require(liquidityApplied[provider][liquidity][job] < now, "credit: bonding");
        uint _liquidity = balances[address(liquidity)];
        uint _credit = _liquidity.mul(liquidityAmount[msg.sender][liquidity][job]).div(
            ERC20(liquidity).totalSupply());
        _mint(address(this), _credit);
        credits[job][address(this)] = credits[job][address(this)].add(_credit);
        liquidityAmount[msg.sender][liquidity][job] = 0;

        emit ApplyCredit(job, msg.sender, block.number, _credit);
    }
```

Listing 3.21: Keep3r.sol

Our analysis shows this `applyCreditToJob()` routine can be affected by a flashloan-assisted sandwiching attack such that the calculated credit amount can be substantially huge. Specifically, to perform the attack, a malicious actor can first request a flashloan to deposit into the `Uniswap` pool so that the pool's `KPR` balance is greatly increased, then invoke `applyCreditToJob()` to perform the credit amount (line) and enjoy the freshly minted credit to the job (thanks to the greatly increased KPR balance in the Uniswap pool), and finally withdraw via skim() to return the flashloan.

Recommendation Revise current execution logic of applyCreditToJob() to defensively detect sudden changes to a reserve balance and block malicious attempts. A mitigation scheme would be the use of require(msg.sender == tx.origin) to prevent flashloans from being applied.

Status This issue has been mitigated in the commit: 675d7509aa10f6731d8d1ac08928fc76c0a9089c.

3.13 Improved Events in SubmitJob/UnbondJob/RemoveJob

- ID: PVE-013
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Keep3r
- Category: Status Codes [13]
- CWE subcategory: CWE-391 [4]

Description

Meaningful events are an important part in smart contract design as they can not only greatly expose the runtime dynamics of smart contracts, but also allow for better understanding about their behavior and facilitate off-chain analytics. In general, events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed.

In the following, we list a few representative events that have been defined in Keep3r.

```
    function addLiquidityToJob(address liquidity, address job, uint amount) external {
        require(liquidityAccepted[liquidity], "addLiquidityToJob: !pair");
        ERC20(liquidity).transferFrom(msg.sender, address(this), amount);
        liquidityProvided[msg.sender][liquidity][job] = liquidityProvided[msg.sender][
            liquidity][job].add(amount);

        liquidityApplied[msg.sender][liquidity][job] = now.add(LIQUIDITYBOND);
        liquidityAmount[msg.sender][liquidity][job] = liquidityAmount[msg.sender][
            liquidity][job].add(amount);

        if (!jobs[job] && jobProposalDelay[job] < now) {
            Governance(governance).proposeJob(job);
            jobProposalDelay[job] = now.add(UNBOND);
        }
        emit SubmitJob(job, msg.sender, block.number, amount);
    }
```

Listing 3.22: Keep3r.sol

It comes to our attention that the event `SubmitJob` does not have the liquidity information. Currently, the event is defined as `event SubmitJob(address indexed job, address indexed provider, uint block, uint credit)`. However, the last information on `credit` indicates an amount, but without the information on the denominated asset.

Moreover, note that each emitted event is represented as a topic that usually consists of the signature (from a `keccak256` hash) of the event name and the types (`uint256`, `string`, etc.) of its parameters. Each indexed type will be treated like an additional topic. If an argument is not indexed, which means it will be attached as data (instead of a separate topic). Considering that the liquidity asset is typically queried, it is suggested to be considered as a topic via the `indexed` keyword.

There are a few other events that share similar issue without the liquidity information. Examples include `UnbondJob` and `RemoveJob`.

Recommendation Revise the above events by properly enclosing (and optionally indexing) the related liquidity information.

Status This issue has been fixed in the commit: 74bd30830345328f88c299a55aa748df58bdb335.

3.14 Improved Handling of Corner Cases in Proposal Submissions

- ID: PVE-014
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Keep3rGovernance`
- Category: Business Logics [11]
- CWE subcategory: CWE-837 [9]

Description

Keep3r adopts the governance implementation from `Compound` by accordingly adjusting its governance token and related parameters, e.g., `quorumVotes()` and `proposalThreshold()`. In the following, we elaborate a corner case during a proposal submission, especially regarding the proposer qualification.

To be qualified as a proposer, the governance subsystem requires the proposer to obtain a sufficient number of votes, including from the proposer herself and other voters. The threshold is specified by `proposalThreshold()`. In Keep3r, this number requires 1% of total bonded amount.

```
    function propose(address[] memory targets, uint[] memory values, string[] memory
        signatures, bytes[] memory calldatas, string memory description) public returns
        (uint) {
        require(VOTER.getPriorVotes(msg.sender, block.number.sub(1)) > proposalThreshold
            (), "Governance::propose: proposer votes below proposal threshold");
```

```
349         require(targets.length == values.length && targets.length == signatures.length && targets.length == calldatas.length, "Governance::propose: proposal function information arity mismatch");
350         require(targets.length != 0, "Governance::propose: must provide actions");
351         require(targets.length <= proposalMaxOperations(), "Governance::propose: too many actions");

353         uint latestProposalId = latestProposalIds[msg.sender];
354         if (latestProposalId != 0) {
355           ProposalState proposersLatestProposalState = state(latestProposalId);
356           require(proposersLatestProposalState != ProposalState.Active, "Governance::propose: one live proposal per proposer, found an already active proposal");
357           require(proposersLatestProposalState != ProposalState.Pending, "Governance::propose: one live proposal per proposer, found an already pending proposal");
358         }

360         return _propose(targets, values, signatures, calldatas, description);
361     }
```

Listing 3.23: Keep3rGovernance.sol

If we examine the `propose()` logic, when a proposal is being submitted, the governance verifies up-front the qualification of the proposer (line 348): `require(VOTER.getPriorVotes(msg.sender, block.number.sub(1))> proposalThreshold())`. Note that the number of prior votes is strictly higher than `proposalThreshold()`.

However, if we check the proposal cancellation logic, i.e., the `cancel()` function, a proposal can be canceled (line 437) if the number of prior votes (before current block) is strictly smaller than `proposalThreshold()`. The corner case of having an exact number prior votes as the threshold, though unlikely, is largely unattended. It is suggested to accommodate this particular corner case as well.

```
417     function cancel(uint proposalId) public {
418         ProposalState state = state(proposalId);
419         require(state != ProposalState.Executed, "Governance::cancel: cannot cancel executed proposal");

421         Proposal storage proposal = proposals[proposalId];
422         require(VOTER.getPriorVotes(proposal.proposer, block.number.sub(1)) < proposalThreshold(), "Governance::cancel: proposer above threshold");

424         proposal.canceled = true;
425         for (uint i = 0; i < proposal.targets.length; i++) {
426             cancelTransaction(proposal.targets[i], proposal.values[i], proposal.signatures[i], proposal.calldatas[i], proposal.eta);
427         }

429         emit ProposalCanceled(proposalId);
430     }
```

Listing 3.24: Keep3rGovernance.sol

Recommendation Accommodate the corner case by also allowing the proposal to be successfully submitted when the number of proposer's prior votes is exactly the same as the required threshold, i.e., `proposalThreshold()`.

```
  function propose(address[] memory targets, uint[] memory values, string[] memory signatures, bytes[] memory calldatas, string memory description) public returns (uint) {
      require(VOTER.getPriorVotes(msg.sender, block.number.sub(1)) >= proposalThreshold(), "Governance::propose: proposer votes below proposal threshold");
      require(targets.length == values.length && targets.length == signatures.length && targets.length == calldatas.length, "Governance::propose: proposal function information arity mismatch");
      require(targets.length != 0, "Governance::propose: must provide actions");
      require(targets.length <= proposalMaxOperations(), "Governance::propose: too many actions");

      uint latestProposalId = latestProposalIds[msg.sender];
      if (latestProposalId != 0) {
        ProposalState proposersLatestProposalState = state(latestProposalId);
        require(proposersLatestProposalState != ProposalState.Active, "Governance::propose: one live proposal per proposer, found an already active proposal");
        require(proposersLatestProposalState != ProposalState.Pending, "Governance::propose: one live proposal per proposer, found an already pending proposal");
      }

      return _propose(targets, values, signatures, calldatas, description);
  }
```

Listing 3.25: Keep3rGovernance.sol

Status This issue has been fixed in the commit: 1fe7f0cf3ddefef7276842038d77429d14d833f7.

3.15 Necessity Of Non-Public Functions For Timelocked Transactions

- ID: PVE-015
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: Keep3rGovernance
- Category: Coding Practices [10]
- CWE subcategory: CWE-287 [3]

Description

The Keep3rGovernance contracts provide a number of timelocked functions that are now re-designed to be called only for the contract itself. These timelocked functions are adapted from Compound, but merged into the Keep3rGovernance contract. These timelocked functions include queueTransaction(), cancelTransaction(), and executeTransaction().

The current merge of timelocked functions into Keep3rGovernance seems problematic. Using queueTransaction() as an example, this routine is used to queue the succeeded proposal for timelocked execution. To elaborate the execution logic, we show below the queue() function.

```
    function queue(uint proposalId) public {
        require(state(proposalId) == ProposalState.Succeeded, "Governance::queue:
            proposal can only be queued if it is succeeded");
        Proposal storage proposal = proposals[proposalId];
        uint eta = block.timestamp.add(delay);
        for (uint i = 0; i < proposal.targets.length; i++) {
            _queueOrRevert(proposal.targets[i], proposal.values[i], proposal.signatures[
                i], proposal.calldatas[i], eta);
        }
        proposal.eta = eta;
        emit ProposalQueued(proposalId, eta);
    }

    function _queueOrRevert(address target, uint value, string memory signature, bytes
        memory data, uint eta) internal {
        require(!queuedTransactions[keccak256(abi.encode(target, value, signature, data,
            eta))], "Governance::_queueOrRevert: proposal action already queued at eta"
            );
        queueTransaction(target, value, signature, data, eta);
    }
```

Listing 3.26: Keep3rGovernance.sol

The queue() function is defined as public and can be invoked by anyone. As mentioned earlier, the proposal needs to survive the voting phase and if so is queued for delayed execution. The delayed execution is enforced in queueTransaction() by specifying the Estimated Time of Arrival - eta for the execution. However, as part of the merge, the current `queueTransaction()` enforces the following requirement: `require(msg.sender == address(this))` (line 531). Note that `queue()` can be invoked by anyone, which is apparently blocked by this specific requirement.

```
530     function queueTransaction(address target, uint value, string memory signature, bytes
            memory data, uint eta) public returns (bytes32) {
531         require(msg.sender == address(this), "Timelock::queueTransaction: Call must come
                from admin.");
532         require(eta >= getBlockTimestamp().add(delay), "Timelock::queueTransaction:
                Estimated execution block must satisfy delay.");

534         bytes32 txHash = keccak256(abi.encode(target, value, signature, data, eta));
535         queuedTransactions[txHash] = true;

537         emit QueueTransaction(txHash, target, value, signature, data, eta);
538         return txHash;
539     }
```

Listing 3.27: Keep3rGovernance.sol

Other two timelocked functions, i.e., `cancelTransaction()` and `executeTransaction()`, share the same issue. As a solution, we may need to make these timelocked functions non-public and remove the requirement: `require(msg.sender == address(this))` (lines 531, 542, and 551).

Recommendation Revise the affected timelocked functions from being `public` to `private` or `internal`. In the meantime, remove their restriction on the calling user.

Status This issue has been fixed in the commit: 1fe7f0cf3ddefef7276842038d77429d14d833f7.

3.16 Full Charge of Proposal Execution Cost From Accompanying msg.value

- ID: PVE-016
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Keep3rGovernance`
- Category: Business Logics [11]
- CWE subcategory: CWE-770 [8]

Description

As mentioned earlier, the adopted governance implementation is adapted from `Compound` with necessary adjustments to its governance token and related parameters, e.g., `quorumVotes()` and `proposalThreshold()`. The original governance has been successfully audited by `OpenZeppelin`.

In the following, we would like to comment on a particular issue regarding the proposal execution cost. Note that the actual proposal execution is kicked off by invoking the governance's `execute()` function. This function is marked as `payable`, indicating the transaction sender is responsible for supplying required amount of `ETHs` as each inherent action (line 412) in the proposal may require accompanying certain `ETHs`, specified in `proposal.values[i]`, where i is the i^{th} action inside the proposal.

```
407     function execute(uint proposalId) public payable {
408         require(state(proposalId) == ProposalState.Queued, "Governance::execute:
                proposal can only be executed if it is queued");
409         Proposal storage proposal = proposals[proposalId];
410         proposal.executed = true;
411         for (uint i = 0; i < proposal.targets.length; i++) {
412             executeTransaction(proposal.targets[i], proposal.values[i], proposal.
                    signatures[i], proposal.calldatas[i], proposal.eta);
413         }
414         emit ProposalExecuted(proposalId);
415     }
```

Listing 3.28: Keep3rGovernance.sol

Though it is likely the case that a majority of these actions do not require any `ETHs`, i.e., `proposal.values[i] = 0`, we may be less concerned on the payment of required `ETHs` for the proposal execution. However, in the unlikely case of certain particular actions that do need `ETHs`, the issue of properly attributing the associated cost arises. With that, we need to better keep track of `ETHs` charged for each action and ensure that the transaction sender (who initiates the proposal execution) actually pays the cost. In other words, we do not rely on the governance's balance of `ETHs` for the payment.

Recommendation Properly charge the proposal execution cost by ensuring the amount of accompanying ETH deposit is sufficient. If necessary, we can also return possible leftover (in the amount of `msg.value - consumedEther`) back to the sender.

```
407     function execute(uint proposalId) public payable {
408         require(state(proposalId) == ProposalState.Queued, "Governance::execute:
                proposal can only be executed if it is queued");
409         Proposal storage proposal = proposals[proposalId];
410         proposal.executed = true;
411         uint consumedEther = 0;
412         for (uint i = 0; i < proposal.targets.length; i++) {
413             consumedEther = consumedEther + proposal.values[i];
414             executeTransaction(proposal.targets[i], proposal.values[i], proposal.
                    signatures[i], proposal.calldatas[i], proposal.eta);
415         }
416         if (msg.value > consumedEther) {
417             msg.sender.transfer(msg.value - consumedEther);
418         }
419         emit ProposalExecuted(proposalId);
```

```
    }
```

Listing 3.29: Keep3rGovernance.sol (revised)

Status This issue has been confirmed. Since possible proposal-related actions in Keep3r typically do not involve any ETHs, the team decides to keep it as is.

3.17 Improved Sanity Checks For System Parameters

- ID: PVE-017
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Keep3r, Keep3rGovernance
- Category: Status Codes [13]
- CWE subcategory: CWE-391 [4]

Description

As with other DeFi protocol, Keep3r is designed with a number of important system-wide reconfigurable parameters, e.g., _quorumVotes, _proposalThreshold, and governance. Note that _quorumVotes specifies the number of votes in support of a proposal required in order for a quorum to be reached and for a vote to succeed. The _proposalThreshold parameter controls the number of votes required in order for a voter to become a proposer. The governance parameter records the current governance contract.

However, we notice that the updates of these important system parameters do not have necessary sanity checks in place. As an example, we show here the two helper routines that adjust _quorumVotes and _proposalThreshold . Note that if _quorumVotes is not set properly, say more than 100%, every proposal attempt would fail.

```
    function setQuorum(uint quorum_) external {
        require(msg.sender == address(this), "Governance::setQuorum: timelock only");
        _quorumVotes = quorum_;
    }

    function quorumVotes() public view returns (uint) {
        return VOTER.totalBonded().mul(_quorumVotes).div(BASE);
    }

    function proposalThreshold() public view returns (uint) {
        return VOTER.totalBonded().mul(_proposalThreshold).div(BASE);
    }

    function setThreshold(uint threshold_) external {
        require(msg.sender == address(this), "Governance::setQuorum: timelock only");
        _proposalThreshold = threshold_;
```

```
218     }
```

Listing 3.30: Keep3rGovernance.sol

As these routines update these important parameters that may impact the overall operation and health, great care needs to be taken to ensure these parameters fall in an appropriate range. Currently, there is no sanity checks in place to ensure their correctness. In the meantime, as these parameters control various aspects of system operation, it is always helpful to emit related events when they are being updated.

Recommendation Apply necessary sanity checks to ensure these parameters always fall in an appropriate range. Also emit corresponding events when these risk parameters are being updated.

Status This issue has been fixed in the commit: 1fe7f0cf3ddefef7276842038d77429d14d833f7.

3.18 Incompatibility With Deflationary/Rebasing Tokens

- ID: PVE-018
- Severity: Informational
- Likelihood: N/A
- Impact: Medium
- Target: `Keep3r`
- Category: Business Logics [11]
- CWE subcategory: CWE-708 [7]

Description

Within the proposed keeper network, the `Keep3r` contract operates as the main entry for interaction with keepers and jobs. The registering keepers `bond()` certain assets into the contract. Later on, the registered keepers can `withdraw()` their own assets from the contract (after a proper exit process). With assets in the pool, keepers can fulfill various jobs and claim their rewards/credits.

Naturally, the above `bond()` function is involved in transferring users' assets into the pool (line 900). When transferring standard ERC20 tokens, these asset-transferring routines work as expected: namely the account's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contracts.

```
888     /**
889      * @notice begin the bonding process for a new keeper
890      * @param bonding the asset being bound
891      * @param amount the amount of bonding asset being bound
892      */
893     function bond(address bonding, uint amount) external {
894         require(pendingbonds[msg.sender][bonding] == 0, "bond: bonding");
895         require(!blacklist[msg.sender], "bond: blacklisted");
896         bondings[msg.sender][bonding] = now.add(BOND);
897         if (bonding == address(this)) {
```

```
            _transferTokens(msg.sender, address(this), amount);
        } else {
            ERC20(bonding).transferFrom(msg.sender, address(this), amount);
        }
        pendingbonds[msg.sender][bonding] = pendingbonds[msg.sender][bonding].add(amount
            );
        emit KeeperBonding(msg.sender, block.number, bondings[msg.sender][bonding],
            amount);
    }
```

Listing 3.31: Keep3r.sol

However, in the cases of deflationary tokens, as shown in the above code snippet, the input amount may not be equal to the received amount due to the charged (and burned) transaction fee. As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above `bond()` operation may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts in the cases of deflationary tokens. (And keep in mind that `USDT` may become deflationary if the control switch in its token contract is turned on.)

One mitigation is to query the asset change right before and after the asset-transferring routines. In other words, instead of automatically assuming the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `transfer()`/`transferFrom()` is expected and aligned well with the intended operation. Though these additional checks cost additional gas usage, we feel that they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary. Another mitigation is to regulate the set of ERC20 tokens that are permitted into Keep3r.

Recommendation Regulate the set of bond tokens supported in Keep3r and, if there is a need to support deflationary tokens, apply necessary mitigation mechanisms to keep track of accurate balances.

Status This issue has been fixed in the commit: a29de87be5d76f2cf74280bbdc4838dbfd4b281e.

4 Conclusion

In this audit, we have analyzed the design and implementation of Keep3r, 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.

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 [16, 17, 18, 19, 22].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [24] 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[4] MITRE. CWE-391: Unchecked Error Condition. https://cwe.mitre.org/data/definitions/391.html.

[5] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[6] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[7] MITRE. CWE-708: Incorrect Ownership Assignment. https://cwe.mitre.org/data/definitions/708.html.

[8] MITRE. CWE-770: Allocation of Resources Without Limits or Throttling. https://cwe.mitre.org/data/definitions/770.html.

[9] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

[10] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

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

[12] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[13] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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

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

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

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

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

[21] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

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

[23] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.

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