SMART CONTRACT AUDIT REPORT

for

KINE PROTOCOL

Prepared By: Shuxiao Wang

PeckShield

February 26, 2021

Document Properties

Client	Kine Protocol
Title	Smart Contract Audit Report
Target	Kine
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Huaguo Shi, Jeff Liu
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 26, 2021	Xuxian Jiang	Final Release
1.0-rc1	February 25, 2021	Xuxian Jiang	Release Candidate #1
0.5	February 22, 2021	Xuxian Jiang	Add More Findings #4
0.4	February 21, 2021	Xuxian Jiang	Add More Findings #3
0.3	February 18, 2021	Xuxian Jiang	Add More Findings #2
0.2	February 16, 2021	Xuxian Jiang	Add More Findings #1
0.1	February 3, 2021	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 the **Kine** 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 Kine

`Kine` is a decentralized protocol that establishes general purpose liquidity pools backed by a customizable portfolio of digital assets. The liquidity pool allows traders to open and close derivatives positions according to trusted price feeds, avoiding the need of counterparties. `Kine` lifts the restriction on existing peer-to-pool (aka peer-to-contract) trading protocols, by expanding the collateral space to any Ethereum-based assets and allowing third-party liquidation. At its core, the Kine protocol is a collateralized lending system. While the collaterals are general ERC20 assets and the lending asset is a special purpose token `KUSD` representing a stake in a liquidity pool.

The basic information of Kine is as follows:

Table 1.1: Basic Information of Kine

Item	Description
Client	Kine Protocol
Website	https://kine.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 26, 2021

In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit. Note that Kine assumes a trusted entity to update timely and reliable market price feeds for supported assets.

- https://github.com/Kine-Technology/kine-protocol.git (a8c0a8)
- https://github.com/Kine-Technology/kine-oracle.git (eb64d29)

And here are the commit IDs after all fixes for the issues found in the audit have been checked in:

- https://github.com/Kine-Technology/kine-protocol.git (ac4036d)
- https://github.com/Kine-Technology/kine-oracle.git (752d5a9)

1.2 About PeckShield

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

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [17]:

- 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) [16], 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 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
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 Logic	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.

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 Kine implementation. 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■■
Low	7	■■■■■■■
Informational	3	■■■
Total	12	

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities, 7 low-severity vulnerabilities, and 3 informational recommendations.

Table 2.1: Key Kine Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Suggested Adherence of Checks-Effects-Interactions	Time and State	Fixed
PVE-002	Low	Improved Precision By Multiplication-Before-Division	Numeric Errors	Fixed
PVE-003	Medium	Potential Overflow Mitigation in notifyRewardAmount()	Numeric Errors	Fixed
PVE-004	Low	Maturity Miscalculation Across Multiple Release Periods	Numeric Errors	Confirmed
PVE-005	Informational	Same Controller Enforcement In liquidateBorrowAllowed()	Coding Practices	Fixed
PVE-006	Low	Improved Sanity Checks Of System/Function Parameters	Coding Practices	Fixed
PVE-007	Low	Safe-Version Replacement With safeApprove(), safeTransfer() And safeTransferFrom()	Coding Practices	Fixed
PVE-008	Informational	Improved Ether Transfers	Business Logics	Confirmed
PVE-009	Informational	Inconsistency Between Document and Implementation	Coding Practices	Fixed
PVE-010	Low	Inaccurate Error Reason in redeemAllowedInternal()	Business Logic	Fixed
PVE-011	Low	Possible Risk in Front-running repayBorrowBehalf()	Time and State	Fixed
PVE-012	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `KToken`
- Category: Time and State [14]
- CWE subcategory: CWE-663 [8]

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` [20] exploit, and the recent `Uniswap/Lendf.Me` hack [19].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `KToken` as an example, the `redeemFresh()` 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 313) starts before effecting the update on internal states (lines 316−317), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via another entry function.

```
283     function redeemFresh(address payable redeemer, uint redeemTokensIn) internal {
284         require(redeemTokensIn != 0, "redeemTokensIn must not be zero");
285 
286         RedeemLocalVars memory vars;
287 
```

```
        /* Fail if redeem not allowed */
        (bool allowed, string memory reason) = controller.redeemAllowed(address(this), redeemer, redeemTokensIn);
        require(allowed, reason);

        /*
         * We calculate the new total supply and redeemer balance, checking for underflow:
         *  totalSupplyNew = totalSupply - redeemTokens
         *  accountTokensNew = accountTokens[redeemer] - redeemTokens
         */
        vars.totalSupplyNew = totalSupply.sub(redeemTokensIn, REDEEM_NEW_TOTAL_SUPPLY_CALCULATION_FAILED);

        vars.accountTokensNew = accountTokens[redeemer].sub(redeemTokensIn, REDEEM_NEW_ACCOUNT_BALANCE_CALCULATION_FAILED);

        /* Fail gracefully if protocol has insufficient cash */
        require(getCashPrior() >= redeemTokensIn, TOKEN_INSUFFICIENT_CASH);

        /////////////////////////
        // EFFECTS & INTERACTIONS

        /*
         * We invoke doTransferOut for the redeemer and the redeemAmount.
         *  Note: The kToken must handle variations between ERC-20 and ETH underlying.
         *  On success, the kToken has redeemAmount less of cash.
         *  doTransferOut reverts if anything goes wrong, since we can't be sure if side effects occurred.
         */
        doTransferOut(redeemer, redeemTokensIn);

        /* We write previously calculated values into storage */
        totalSupply = vars.totalSupplyNew;
        accountTokens[redeemer] = vars.accountTokensNew;

        /* We emit a Transfer event, and a Redeem event */
        emit Transfer(redeemer, address(this), redeemTokensIn);
        emit Redeem(redeemer, redeemTokensIn);

        /* We call the defense hook */
        controller.redeemVerify(address(this), redeemer, redeemTokensIn);
    }
```

Listing 3.1: KToken::redeemFresh()

In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`. Moreover, the current implementation has taken precautions in making use of `nonReentrant` to block possible `re-entrancy`.

However, it is important to mention that the `Kine` protocol partitions various functionalities in two sets: `KToken` and `KMCD`. The first set handles the collateral-side while the second handles the borrow-side. These two sets are implemented in two different contracts. As a result, the `nonReentrant` protection on one contract does not prevent possible `re-entrancy` from another. With that, it is strongly suggested to adhere with the `checks-effects-interactions` principle.

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` best practice.

Status The issue has been fixed by this commit: `36dee57`.

3.2 Improved Precision By Multiplication-Before-Division

- ID: PVE-002
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `Multiple Contracts`
- Category: Numeric Errors [15]
- CWE subcategory: CWE-190 [5]

Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, the lack of `float` support in `Solidity` may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (`mul`) and division (`div`) are involved.

In particular, we use the `claimable()` (in `KUSDMinter` contract) as an example. This routine is used to calculate the claimable rewards so far.

```
    /**
     * @notice Calculate account's claimable rewards so far.
     * @param account Which account to be viewed.
     * @return Account's claimable rewards so far.
     */
    function claimable(address account) external view returns (uint) {
        uint accountNewAccruedReward = earned(account);
        uint pastTime = block.timestamp.sub(accountRewardDetails[account].lastClaimTime);
        uint maturedReward = accountNewAccruedReward.mul(1e18).div(rewardReleasePeriod).mul(pastTime).div(1e18);
        if (maturedReward > accountNewAccruedReward) {
            maturedReward = accountNewAccruedReward;
        }
        return maturedReward;
```

```
279     }
```

Listing 3.2: KUSDMinter::claimable()

We notice the calculation of the `maturedReward` (line 274) involves mixed multiplication and devision. For improved precision, it is better to calculate the multiplication before the division, i.e., `accountNewAccruedReward.mul(pastTime).div(rewardReleasePeriod)`. Also, the arithmetic operations with `mul(1e8)` and `div(1e8)` can be canceled out.

Similarly, the calculation of `liquidateCalculateSeizeTokens()` in `Controller` contract (lines 757 – 758) can be accordingly adjusted. Note that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible.

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status The issue has been fixed by this commit: `bb9a840`.

3.3 Potential Overflow Mitigation in notifyRewardAmount()

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `KUSDMinter`
- Category: Numeric Errors [15]
- CWE subcategory: CWE-190 [5]

Description

The `Kine` protocol is architecturally designed to incentivize users. By design, the contract `KUSDMinter` allows an entity i.e., `rewardDistribution`, to distribute rewards to protocol users. Specifically, there is a routine `notifyRewardAmount()` that is defined to apply new rewards for distribution.

To elaborate, we show below the full implementation of the `notifyRewardAmount()` routine. This is a protected function that can only be invoked by the configured `rewardDistribution` to specify the intended `reward`.

```
    function notifyRewardAmount(uint reward) external onlyRewardDistribution
        updateReward(address(0)) {
        if (block.timestamp > startTime) {
            if (block.timestamp >= periodFinish) {
                rewardRate = reward.div(rewardDuration);
            } else {
                uint remaining = periodFinish.sub(block.timestamp);
                uint leftover = remaining.mul(rewardRate);
                rewardRate = reward.add(leftover).div(rewardDuration);
            }
```

```
            lastUpdateTime = block.timestamp;
            periodFinish = block.timestamp.add(rewardDuration);
            emit RewardAdded(reward);
        } else {
            rewardRate = reward.div(rewardDuration);
            lastUpdateTime = startTime;
            periodFinish = startTime.add(rewardDuration);
            emit RewardAdded(reward);
        }
    }
```

Listing 3.3: KUSDMinter::notifyRewardAmount()

However, a further analysis of the logic shows another related routine `rewardPerToken()`, which is responsible for calculating the reward rate for each staked token and it is always invoked up-front for almost every public function to properly update and use the latest reward rate.

```
235     /**
236      * @notice Calculate new accrued reward per staked Kine MCD.
237      * @return Current accrued reward per staked Kine MCD.
238      */
239     function rewardPerToken() public view returns (uint) {
240         uint totalStakes = totalStakes();
241         if (totalStakes == 0) {
242             return rewardPerTokenStored;
243         }
244         return
245         rewardPerTokenStored.add(
246             lastTimeRewardApplicable()
247             .sub(lastUpdateTime)
248             .mul(rewardRate)
249             .mul(1e18)
250             .div(totalStakes)
251         );
252     }
```

Listing 3.4: KUSDMinter::rewardPerToken()

A potential issue may surface if an oversized `reward` is applied. In particular, with the multiplication of three `uint256` integer, it is possible for their multiplication to have an undesirable overflow (lines 245 – 250), especially when the `rewardRate` is largely controlled by an external entity, i.e., `rewardDistribution`. An overflowed computation may revert ongoing transactions and potentially disable the borrow functionality! Fortunately, the authentication check on the caller of `notifyRewardAmount()` greatly alleviates such concern. Currently, only the `rewardDistribution` address is able to call.

Recommendation Apply necessary measures to mitigate the potential overflow risk in the incentivizer mechanism.

Status The issue has been fixed by this commit: `980e452`.

3.4 Maturity Miscalculation Across Multiple Release Periods

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `KUSDMinter`
- Category: Numeric Errors [15]
- CWE subcategory: CWE-190 [5]

Description

As mentioned in Section 3.3, the `Kine` protocol has developed an incentivizer mechanism to attract and reward participating users. Users need to stake their assets to be eligible for rewards. The protocol supports the notion of `RewardReleasePeriod`, which indicates how long all earned rewards will be matured. With that, each staking user will be associated with a specific state `lastClaimTime` that keeps track of the last time the user claims the reward.

To elaborate, we show below the `getReward()` routine. This routine is designed to allow users to claim the matured rewards. Basically, it firstly examines the `accuredReward` and the computes the `maturedReward`. The computed `maturedReward` will then be transferred to the user.

```
    /**
     * @notice Claim the matured rewards of caller.
     * Claim will fail if hasn't reach start time.
     */
    function getReward() external checkStart updateReward(msg.sender) {
        uint reward = accountRewardDetails[msg.sender].accruedReward;
        if (reward > 0) {
            uint pastTime = block.timestamp.sub(accountRewardDetails[msg.sender].lastClaimTime);
            uint maturedReward = reward.mul(1e18).mul(pastTime).div(rewardReleasePeriod).div(1e18);
            if (maturedReward > reward) {
                maturedReward = reward;
            }

            accountRewardDetails[msg.sender].accruedReward = accountRewardDetails[msg.sender].accruedReward.sub(maturedReward);
            accountRewardDetails[msg.sender].lastClaimTime = block.timestamp;
            kine.safeTransfer(msg.sender, maturedReward);
            emit RewardPaid(msg.sender, maturedReward);
        }
    }
```

Listing 3.5: KUSDMinter::getReward()

It comes to our attention that the computation of `maturedReward` may be improved. Specifically, it take into account the `lastClaimTime` that keeps track of the last time the user claims the reward.

If `lastClaimTime` occurs at the last reward period, the time range between the end-time of last period and the start-time of current period will be considered part of maturity time! In other words, the idle time range that is not supposed to be part of maturity time has been unfortunately taken into account for maturity.

Recommendation Properly measure the maturity time across multiple release periods so that the correct rewards can be computed for claims.

Status This issue has been confirmed. This is a design choice to balance the user convenience and implementation complexity.

3.5 Same Controller Enforcement In liquidateBorrowAllowed()

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `KMCD`
- Category: Business Logic [13]
- CWE subcategory: CWE-841 [9]

Description

At its core, the `Kine` protocol is a collateralized lending system. While the collaterals are general ERC20 assets and the lending asset is a special purpose token representing a stake in a liquidity pool. Accordingly, the protocol has partitioned its functionality into two parts: `KToken` and `KMCD`. These two parts work closely as the first one provides required collaterals so that the second part is allowed to borrow. When the collaterals are insufficient, the borrow position can be liquidated. In the following, we examine the `liquidateBorrowAllowed()` that verifies whether a liquidation should be allowed to occur.

```
    function liquidateBorrowAllowed(
        address kTokenBorrowed,
        address kTokenCollateral,
        address liquidator,
        address borrower,
        uint repayAmount) external returns (bool allowed, string memory reason) {
        // Shh - currently unused
        liquidator;

        if (!markets[kTokenBorrowed].isListed  !markets[kTokenCollateral].isListed) {
            allowed = false;
            reason = MARKET_NOT_LISTED;
            return (allowed, reason);
        }

```

```
460         /* The borrower must have shortfall in order to be liquidatable */
461         (, uint shortfall) = getAccountLiquidityInternal(borrower);
462         if (shortfall == 0) {
463             allowed = false;
464             reason = INSUFFICIENT_SHORTFALL;
465             return (allowed, reason);
466         }
467
468         /* The liquidator may not repay more than what is allowed by the closeFactor */
469         /* Only KMCD has borrow related logics */
470         uint borrowBalance = KMCD(kTokenBorrowed).borrowBalance(borrower);
471         uint maxClose = mulScalarTruncate(Exp({mantissa: closeFactorMantissa}),
                borrowBalance);
472         if (repayAmount > maxClose) {
473             allowed = false;
474             reason = TOO_MUCH_REPAY;
475             return (allowed, reason);
476         }
477
478         allowed = true;
479         return (allowed, reason);
480     }
```

Listing 3.6: Controller::liquidateBorrowAllowed()

The validation logic works as follows: It firstly checks both markets, i.e., `kTokenBorrowed` and `kTokenCollateral`, are indeed listed (line 454), next validates the borrower must have shortfall (lines 461 – 466), and finally ensures the liquidated amount is within allowed range (lines 470 – 476).

The above logic can be improved by further verifying that these two markets share the same `controller`. In other words, these two markets should not be allowed to liquidate if they are managed by two different controllers. The reason is that two different controllers have different policies in governing various operations, e.g., `mint/redeem`, `borrow/repay`, `liquidate/seize`, and `transfer`. We notice that the suggested same controller enforcement have been validated in `seizeAllowed()`. However, this enforcement is better performed at the very beginning when the liquidation action is intended.

Recommendation Enforce the controller consistency between the two involved markets: `kTokenBorrowed` and `kTokenCollateral`.

Status The issue has been fixed by this commit: `47a346d`.

3.6 Improved Sanity Checks For System/Function Parameters

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Controller`
- Category: Coding Practices [12]
- CWE subcategory: CWE-1126 [4]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Kine` protocol is no exception. Specifically, if we examine the `Controller` contract, it has defined a number of protocol-wide risk parameters: `closeFactorMantissa` and `liquidationIncentiveMantissa`. In the following, we show the corresponding routines that allow for their changes.

```
    /**
      * @notice Sets the closeFactor used when liquidating borrows
      * @dev Admin function to set closeFactor
      * @param newCloseFactorMantissa New close factor, scaled by 1e18
      */
    function _setCloseFactor(uint newCloseFactorMantissa) external onlyAdmin() {
        uint oldCloseFactorMantissa = closeFactorMantissa;
        closeFactorMantissa = newCloseFactorMantissa;
        emit NewCloseFactor(oldCloseFactorMantissa, closeFactorMantissa);
    }
```

Listing 3.7: Controller :: _setCloseFactor()

```
    /**
      * @notice Sets liquidationIncentive
      * @dev Admin function to set liquidationIncentive
      * @param newLiquidationIncentiveMantissa New liquidationIncentive scaled by 1e18
      */
    function _setLiquidationIncentive(uint newLiquidationIncentiveMantissa) external
        onlyAdmin() {
        uint oldLiquidationIncentiveMantissa = liquidationIncentiveMantissa;
        liquidationIncentiveMantissa = newLiquidationIncentiveMantissa;
        emit NewLiquidationIncentive(oldLiquidationIncentiveMantissa,
            newLiquidationIncentiveMantissa);
    }
```

Listing 3.8: Controller :: _setLiquidationIncentive ()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `liquidationIncentiveMantissa` may charge unreasonably high fee in the `liquidate` operation, hence incurring cost to keepers or hurting the adoption of the protocol.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. If necessary, also consider emitting relevant events for their changes.

Status The issue has been fixed by this commit: `bfa889c`.

3.7 Safe-Version Replacement With safeApprove(), safeTransfer() And safeTransferFrom()

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `KineTreasury`, `KUSDVault`
- Category: Coding Practices [12]
- CWE subcategory: CWE-1126 [4]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transfer()` routine and possible idiosyncrasies from current widely-used token contracts. In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below.

```
/**
* @dev transfer token for a specified address
* @param _to The address to transfer to.
* @param _value The amount to be transferred.
*/
function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
    uint fee = (_value.mul(basisPointsRate)).div(10000);
    if (fee > maximumFee) {
        fee = maximumFee;
    }
    uint sendAmount = _value.sub(fee);
    balances[msg.sender] = balances[msg.sender].sub(_value);
    balances[_to] = balances[_to].add(sendAmount);
    if (fee > 0) {
        balances[owner] = balances[owner].add(fee);
        Transfer(msg.sender, owner, fee);
    }
    Transfer(msg.sender, _to, sendAmount);
```

```
139     }
```

Listing 3.9: USDT Token Contract

It is important to note the `transfer()` function does not have a return value. However, the `IERC20` interface has defined the following `transfer()` interface with a `bool` return value: `function transfer(address recipient, uint256 amount)external returns (bool)`. As a result, the call to `transfer()` may expect a return value. With the lack of return value of `USDT`'s `transfer()`, the call will be unfortunately reverted.

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. To use this library you can add a using SafeERC20 for IERC20. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `transferErc20()` routine in the `KineTreasury` contract. If the `USDT` token is given as the routine's argument, i.e., `erc20Addr`, the unsafe version of `erc20.transfer(target, amount)` (line 64) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)!

```
219     // @notice Only admin can call
220     function transferErc20(address erc20Addr, address target, uint amount) external
            onlyAdmin {
221         // check balance;
222         IERC20 erc20 = IERC20(erc20Addr);
223         uint balance = erc20.balanceOf(address(this));
224         require(balance >= amount, "not enough erc20 balance");
225         // transfer token
226         erc20.transfer(target, amount);

228         emit TransferErc20(erc20Addr, target, amount);
229     }
```

Listing 3.10: KineTreasury::transferErc20()

Note that the same issue exists in the `_transferErc20()` routine from the `KUSDVault` contract, which reverts related liquidation additions.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status The issue has been fixed by this commit: `0120eb4`.

3.8 Improved Ether Transfers

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `KEther`
- Category: Business Logic [13]
- CWE subcategory: CWE-841 [9]

Description

As described in Section 3.7, assets are transferred in or out with a number of helper routines such as `doTransferIn()` and `doTransferOut()`. While dealing with ERC20 tokens, we have examined related helper routines in their handling of non-standard ERC20 implementations. As for the case of transferring `ETH`, the Solidity function `transfer()` is used (line 108 in the code snippet below). However, as described in [2], when the recipient happens to be a contract that implements a callback function containing EVM instructions such as `SLOAD`, the 2300 gas supplied with `transfer()` might not be sufficient, leading to an out-of-gas error.

```
106     function doTransferOut(address payable to, uint amount) internal {
107         /* Send the Ether, with minimal gas and revert on failure */
108         to.transfer(amount);
109     }
```

Listing 3.11: KEther::doTransferOut()

As suggested in [2], we may consider avoiding the direct use of Solidity's `transfer()` as well. Note that we need to exercise extra caution during the use of `call()` as it may lead to side effects such as `re-entrancy` and gas token vulnerabilities. In other words, we need to specify the maximum allowed gas amount when making the (untrusted) external `call()`.

Recommendation When transferring `ETH`, it is suggested to replace the Solidity function `transfer()` with `call()`.

Status This issue has been confirmed.

3.9 Inconsistency Between Document and Implementation

- ID: PVE-009
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [12]
- CWE subcategory: CWE-1041 [3]

Description

There are a few misleading comments embedded among lines of solidity code, which bring unnecessary hurdles to understand and/or maintain the software. An example comment can be found in line 460 of `KUSDMinter::notifyRewardAmount()`. The preceding function summary indicates that *"Notify will fail if hasn't reach start time."* However, the implementation logic (lines 479 – 484) indicates if the start time has not been reached, this routine simply updates the states `rewardRate`, `lastUpdateTime`, and `periodFinish`, without failing the transaction.

```
460     /**
461      * @notice Notify rewards has been added, trigger a new round of reward period,
            recalculate reward rate and duration end time.
462      * If distributor notify rewards before this round duration end time, then the
            leftover rewards of this round will roll over to
463      * next round and will be distributed together with new rewards in next round of
            reward period.
464      * Notify will fail if hasn't reach start time.
465      * @param reward How many of rewards has been added for new round of reward period.
466      */
467     function notifyRewardAmount(uint reward) external onlyRewardDistribution
            updateReward(address(0)) {
468         if (block.timestamp > startTime) {
469             if (block.timestamp >= periodFinish) {
470                 rewardRate = reward.div(rewardDuration);
471             } else {
472                 uint remaining = periodFinish.sub(block.timestamp);
473                 uint leftover = remaining.mul(rewardRate);
474                 rewardRate = reward.add(leftover).div(rewardDuration);
475             }
476             lastUpdateTime = block.timestamp;
477             periodFinish = block.timestamp.add(rewardDuration);
478             emit RewardAdded(reward);
479         } else {
480             rewardRate = reward.div(rewardDuration);
481             lastUpdateTime = startTime;
482             periodFinish = startTime.add(rewardDuration);
483             emit RewardAdded(reward);
484         }
```

```
    }
```

Listing 3.12: KUSDMinter::notifyRewardAmount()

Also, we notice inconsistency in the design document and current implementation. In particular, it is stated from the Section 3.3.4 of the design document: *User's accrued rewards in KUSD-Minter will gradually mature in a release period. Every time user claim rewards, the release timer will be updated. The matured reward of total accrued rewards is calculated as* $Reward_{matured} = max(1, \frac{Time_{curernt} - Time_{lastClaim}}{ReleasePeriod}) * Reward_{accrued}$.

```
    /**
     * @notice Claim the matured rewards of caller.
     * Claim will fail if hasn't reach start time.
     */
    function getReward() external checkStart updateReward(msg.sender) {
        uint reward = accountRewardDetails[msg.sender].accruedReward;
        if (reward > 0) {
            uint pastTime = block.timestamp.sub(accountRewardDetails[msg.sender].
                lastClaimTime);
            uint maturedReward = reward.mul(1e18).mul(pastTime).div(rewardReleasePeriod)
                .div(1e18);
            if (maturedReward > reward) {
                maturedReward = reward;
            }

            accountRewardDetails[msg.sender].accruedReward = accountRewardDetails[msg.
                sender].accruedReward.sub(maturedReward);
            accountRewardDetails[msg.sender].lastClaimTime = block.timestamp;
            kine.safeTransfer(msg.sender, maturedReward);
            emit RewardPaid(msg.sender, maturedReward);
        }
    }
```

Listing 3.13: KUSDMinter::getReward()

The actual implementation (as shown above) shows the total accrued rewards is calculated as $Reward_{matured} = min(1, \frac{Time_{curernt} - Time_{lastClaim}}{ReleasePeriod}) * Reward_{accrued}$.

Last, there are additional contracts in the `kine-oracle` repository, which may not needed and can be safely removed. Specifically, the contracts `UniswapConfig`, and `UniswapAnchorView` are not currently used.

Recommendation Ensure the consistency between documents (including embedded comments) and implementation.

Status The issue has been fixed by the following commits: `980e452`, and `ac4036d`.

3.10 Inaccurate Error Reason in redeemAllowedInternal()

- ID: PVE-010
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: AToken
- Category: Business Logic [13]
- CWE subcategory: CWE-841 [9]

Description

In Ethereum, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the Controller contract as an example. This contract is designed to enforce various policies when a number of protocol operations are performed. During our analysis, we notice the redeemAllowedInternal() enforcement (line 275) contains incorrect information. Specifically, if the KToken-mapped market is not listed, the returned failure should be MARKET_NOT_LISTED. not EXIT_MARKET_REJECTION.

```
266     /**
267      * @param kToken The market to verify the redeem against
268      * @param redeemer The account which would redeem the tokens
269      * @param redeemTokens The number of kTokens to exchange for the underlying asset in
             the market
270      * @return false and reason if redeem not allowed, otherwise return true and empty
           string
271      */
272     function redeemAllowedInternal(address kToken, address redeemer, uint redeemTokens)
            internal view returns (bool allowed, string memory reason) {
273         if (!markets[kToken].isListed) {
274             allowed = false;
275             reason = EXIT_MARKET_REJECTION;
276             return (allowed, reason);
277         }
278
279         /* If the redeemer is not 'in' the market, then we can bypass the liquidity
                check */
280         if (!markets[kToken].accountMembership[redeemer]) {
281             allowed = true;
282             return (allowed, reason);
283         }
284
285         /* Otherwise, perform a hypothetical liquidity check to guard against shortfall
                */
```

```
        (, uint shortfall) = getHypotheticalAccountLiquidityInternal(redeemer, KToken(
            kToken), redeemTokens, 0);
        if (shortfall > 0) {
            allowed = false;
            reason = INSUFFICIENT_LIQUIDITY;
            return (allowed, reason);
        }

        allowed = true;
        return (allowed, reason);
    }
```

Listing 3.14: Controller::redeemAllowedInternal()

Recommendation Properly report the correct reason when an enforced policy is violated. The failure reason will be returned to the caller or emitted to better reflect the true logic and is very helpful for external analytics and reporting tools.

Status The issue has been fixed by this commit: 2a6c0d9.

3.11 Possible Risk in Front-running repayBorrowBehalf()

- ID: PVE-011
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: KMCD
- Category: Time and State [11]
- CWE subcategory: CWE-362 [7]

Description

At its core, the Kine protocol is a collateralized lending system that supports basic borrow and repay operations. In the following, we examine the logic behind the repay operation.

To elaborate, we show below the code snippet of repayBorrowFresh(). This function implements a rather straightforward logic in firstly performing sanity checks of this repay operation, next fetching the amount the borrower owes, and calculating the new borrower and total borrow balances. It comes to our attention that when repayAmount == -1, current implementation logic considers the purpose of performing a full repayment with the amount of accountBorrows (line 217).

```
198     /**
199      * @notice Borrows are repaid by another user, should be the minter.
200      * @param payer the account paying off the MCD
201      * @param borrower the account with the MCD being payed off
202      * @param repayAmount the amount of MCD being returned
203      * @return the actual repayment amount.
204      */
```

```
    function repayBorrowFresh(address payer, address borrower, uint repayAmount)
        internal returns (uint) {
        /* Fail if repayBorrow not allowed */
        (bool allowed, string memory reason) = controller.repayBorrowAllowed(address(
            this), payer, borrower, repayAmount);
        require(allowed, reason);

        RepayBorrowLocalVars memory vars;

        /* We fetch the amount the borrower owes */
        vars.accountBorrows = accountBorrows[borrower];

        /* If repayAmount == -1, repayAmount = accountBorrows */
        if (repayAmount == uint(-1)) {
            vars.repayAmount = vars.accountBorrows;
        } else {
            vars.repayAmount = repayAmount;
        }

        /*
         * We calculate the new borrower and total borrow balances, failing on underflow
            :
         *  accountBorrowsNew = accountBorrows - actualRepayAmount
         *  totalBorrowsNew = totalBorrows - actualRepayAmount
         */
        vars.accountBorrowsNew = vars.accountBorrows.sub(vars.repayAmount,
            REPAY_BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED);
        vars.totalBorrowsNew = totalBorrows.sub(vars.repayAmount,
            REPAY_BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED);

        /* We write the previously calculated values into storage */
        accountBorrows[borrower] = vars.accountBorrowsNew;
        totalBorrows = vars.totalBorrowsNew;

        /* We emit a RepayBorrow event */
        emit RepayBorrow(payer, borrower, vars.repayAmount, vars.accountBorrowsNew, vars
            .totalBorrowsNew);

        /* We call the defense hook */
        controller.repayBorrowVerify(address(this), payer, borrower, vars.repayAmount);

        return vars.repayAmount;
    }
```

Listing 3.15: KMCD::repayBorrowFresh()

However, the full repayment logic exposes a possible race condition issue [1]. Specifically, when a user intends to fully repay the current borrow amount, the borrower may race to further borrow up to a large amount. This breaks the user's intention of restricting the full repayment of current borrow amount, not including the new borrow amount. In other words, the user may **not** intend to repay the sum of old borrow amount and new borrow amount.

Recommendation Instead of using `-1` to indicate the full repay amount, it is suggested to require a specific repay amount. By doing so, we can eliminate the risk behind the race condition.

Status The issue has been fixed by this commit: `8014c40`.

3.12 Trust Issue of Admin Keys

- ID: PVE-012
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `KUSDMinter`
- Category: Security Features [10]
- CWE subcategory: CWE-287 [6]

Description

In Kine, the privileged account plays a critical role in governing and regulating the system-wide operations (e.g., oracle management, reward adjustment, and parameter setting). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    /**
     * @notice Mint KUSD to treasury account to keep on-chain KUSD consist with off-
         chain trading system
     * @param amount The amount of KUSD to mint to treasury
     */
    function treasuryMint(uint amount) external onlyTreasury {
        kUSD.mint(vault, amount);
        emit TreasuryMint(amount);
    }

    /**
     * @notice Burn KUSD from treasury account to keep on-chain KUSD consist with off-
         chain trading system
     * @param amount The amount of KUSD to burn from treasury
     */
    function treasuryBurn(uint amount) external onlyTreasury {
        kUSD.burn(vault, amount);
        emit TreasuryBurn(amount);
    }

    /**
     * @notice Change treasury account to a new one
     * @param newTreasury New treasury account address
     */
    function _setTreasury(address newTreasury) external onlyOwner {
        address oldTreasury = treasury;
```

```
        treasury = newTreasury;
        emit NewTreasury(oldTreasury, newTreasury);
    }
```

Listing 3.16: Various Privileged Routines in KUSDMinter

Specifically, we examine the privileged functions `treasuryMint()/treasuryBurn()` in `KUSDMinter`. Notice that the privileged account is able to mint/burn `KUSD` to/from specified `treasury` account. Note that the `treasury` account can be dynamically configured from the privileged `owner`. We point out that a compromised privileged account would allow the attacker to add a malicious `treasury` to mint arbitrary `KUSD` tokens. It can also be configured to burn `KUSD` tokens from a specified user account.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed. The team confirmed the plan to hold the admin key in a multi-sig account. All changed to privileged operations will be mitigated with necessary timelocks.

4 Conclusion

In this audit, we have analyzed the Kine design and implementation. The system presents a unique, robust offering as a decentralized non-custodial money market protocol that establishes general purpose liquidity pools backed by a customizable portfolio of digital assets. The current code base is well structured and neatly organized. 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] HaleTom. Resolution on the EIP20 API Approve / TransferFrom multiple withdrawal attack. https://github.com/ethereum/EIPs/issues/738.

[2] Steve Marx. Stop Using Solidity's transfer() Now. https://diligence.consensys.net/blog/2019/09/stop-using-soliditys-transfer-now/.

[3] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[4] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[5] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

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

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

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

[10] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

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

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

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

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

[15] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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

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

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