SMART CONTRACT AUDIT REPORT

for

KCOMPOUND

Prepared By: Shuxiao Wang

PeckShield

April 21, 2021

Document Properties

Client	KeeperDAO
Title	Smart Contract Audit Report
Target	KCompound
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	April 21, 2021	Xuxian Jiang	Final Release
1.0-rc1	March 30, 2021	Xuxian Jiang	Release Candidate #1
0.4	March 24, 2021	Xuxian Jiang	Add More Findings #3
0.3	March 22, 2021	Xuxian Jiang	Add More Findings #2
0.2	March 20, 2021	Xuxian Jiang	Add More Findings #1
0.1	March 15, 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 **KCompound** design document and related smart contract source code, 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 KCompound

`KCompound` is a wrapper around `Compound` and is designed to allow users to open positions on `Compound` and behaves identical to `Compound`. Moreover, users' positions are protected from the flash bots by the underwriters, who provide buffer to the positions that are close to being liquidated. The `KeeperDAO`-friendly keepers preempt liquidations on positions that are underwater (when buffer provided by the underwriters is not considered). When such a liquidation is preempted by the friendly keepers, a portion of it would be returned back to the user.

The basic information of KCompound is as follows:

Table 1.1: Basic Information of KCompound

Item	Description
Issuer	KeeperDAO
Website	https://keeperdao.com/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 21, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that KCompound assumes a trusted price oracle with timely market price feeds for

supported assets and the oracle itself is not part of this audit.

- https://github.com/keeperdao/kcompound (08d3590)

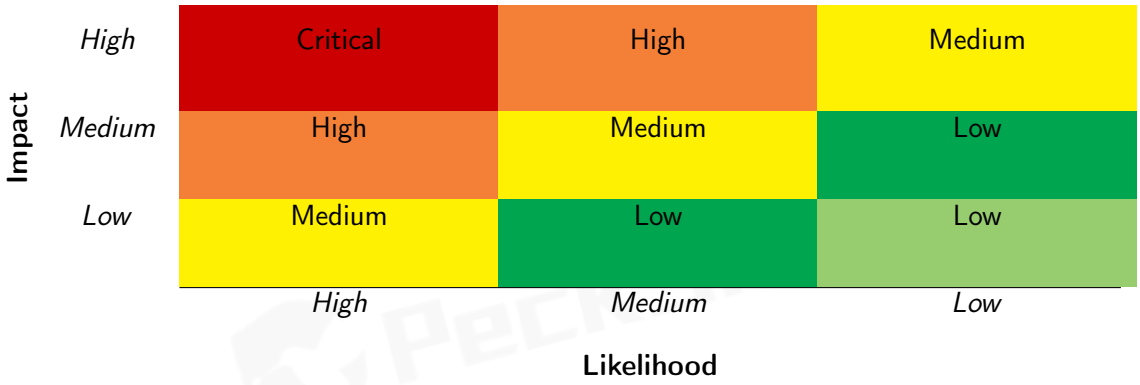
And here is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/keeperdao/kcompound (6ae6ca2)

1.2 About PeckShield

PeckShield Inc. [7] 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



1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [6]:

- 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.

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

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) [5], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

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

Table 1.4: 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 implementation of the KCompound protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business 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	3	■■■
Informational	3	■■■
Total	8	

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

2.2 Key Findings

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

Table 2.1: Key KCompound Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Timely accrueInterest() Refresh In seizeTokenAmount()	Business Logic	Fixed
PVE-002	Low	Duplicate Check In CompoundVars::addCERC20()	Coding Practices	Fixed
PVE-003	Informational	Inconsistency Between Document and Implementation	Coding Practices	Fixed
PVE-004	Medium	Orange vs. Apple in checkBufferValue()	Business Logic	Fixed
PVE-005	Informational	Naming Consistency In KComptroller	Coding Practices	Fixed
PVE-006	Low	Accommodation of approve() Idiosyncrasies	Business Logic	Confirmed
PVE-007	Informational	Incompatibility with Deflationary/Rebasing Tokens	Business Logic	Confirmed
PVE-008	Medium	Incorrect Calculation of KComptroller::clearBuffer()	Business Logic	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 Timely accrueInterest() Refresh In seizeTokenAmount()

- ID: PVE-001
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `KComptroller`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

By design, `KCompound` is a wrapper around `Compound` and is designed to allow users to open positions on `Compound` and behaves identical to `Compound`. Accordingly, there is a `KComptroller` contract that acts as a gateway to various functionality in `KCompoundPosition`. In the following, we examine one specific function `seizeTokenAmount()` from `KComptroller`.

To elaborate, we show below the function's full implementation. This `seizeTokenAmount()` function is designed to validate whether the position is liquidatable. If yes, the function further calculates the amount of tokens that can be seized.

```
72      function seizeTokenAmount(
73          address cTokenRepaid,
74          address cTokenSeized,
75          uint repayAmount
76      ) override external returns (uint) {
77          // accrue interest
78          requireNoError(CToken(cTokenRepaid).accrueInterest(), "KComptroller: failed to
                accrue interest");

80          // Check whether the markets are listed
81          (bool isBorrowTokenListed, ,) = compoundVars.comptroller().markets(cTokenRepaid)
                ;
82          (bool isCollateralTokenListed, ,) = compoundVars.comptroller().markets(
                cTokenSeized);
83          require(isBorrowTokenListed && isCollateralTokenListed, "KComptroller: market
                not listed");
```

```
85          // The borrower must have shortfall in order to be liquidatable
86          ( , uint shortfall) = getAccountLiquidity(msg.sender, 0);
87          require(shortfall != 0, "KComptroller: insufficient shortfall to liquidate");

89          // The liquidator may not repay more than what is allowed by the closeFactor
90          uint borrowBalance = CToken(cTokenRepaid).borrowBalanceStored(msg.sender);
91          (MathError mErr, uint maxClose) = mulScalarTruncate(Exp({mantissa: compoundVars.
                comptroller().closeFactorMantissa()}), borrowBalance);
92          require(mErr == MathError.NO_ERROR, "KComptroller: failed to calculate max close
                amount");
93          require(repayAmount <= maxClose, "KComptroller: repay amount cannot exceed the
                max close amount");

95          // Calculate the amount of tokens that can be seized
96          (uint errCode2, uint seizeTokens) = compoundVars.comptroller()
97              .liquidateCalculateSeizeTokens(cTokenRepaid, cTokenSeized, repayAmount);
98          requireNoError(errCode2, "KComptroller: failed to calculate seize token amount")
                ;

100         // Check that the amount of tokens being seized is less than the user's
101         // cToken balance
102         uint256 totalCollateral = CToken(cTokenSeized).balanceOf(msg.sender).sub(
103             buffer[msg.sender][cTokenSeized]
104         );
105         require(totalCollateral >= seizeTokens, "KComptroller: insufficient liquidity");

107         return seizeTokens;
108     }
```

Listing 3.1: KComptroller::seizeTokenAmount()

Our analysis with the above function shows that the calculation should be performed when both related `CTokens` or markets have timely accrued their interest. Otherwise, the function may operate on the stale principal and index. The current implementation has ensured only the freshness of the `cTokenRepaid` market, but not the `cTokenSeized` market.

Recommendation Properly ensure the freshness of `cTokenSeized` before calculating the token amounts to be seized.

Status The issue has been fixed by this commit: `32665ee`.

3.2 Duplicate Check In CompoundVars::addCERC20()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CompoundVars`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

Description

The KCompound protocol has a contract, i.e., `CompoundVars`, that plays the role of acting as the protocol-wide registry. Specifically, this contract holds the addresses and information related to all the relevant contracts, such as `Compound Comptroller`, `KComptroller`, `WETH`, `KeeperDAO's liquidityPool` contract, etc. While the contract properly provides related `setters` to dynamically configure them, we notice one specific routine, i.e., `addCERC20()`, that can be improved to ensure no duplicate will be added.

Specifically, this `addCERC20()` routine maintains internal mappings `cTokenMap` as well as arrays of `cERC20s` and `_cTokens`. To elaborate, we show below its current implementation. While it properly implements the intended functionality, it comes to our attention this routine does not check whether the given `_newCERC20` is already present in current arrays yet. The lack of this duplicate check may introduce unnecessary inconsistency and cause confusion in the managed states.

```
20      /// @notice allows the owner of add additional
21      /// CERC20 tokens.
22      function addCERC20(CErc20 _newCERC20) public onlyOwner {
23          require(_newCERC20 != CErc20(0), "CompoundVars: CErc20 cannot be 0x0");
24          require(_newCERC20.underlying() != address(0), "CompoundVars: CErc20's
                underlying address cannot be 0x0");
26          cERC20s.push(_newCERC20);
27          cTokenMap[_newCERC20.underlying()] = _newCERC20;
28          _cTokens.push(address(_newCERC20));
29      }
```

Listing 3.2: CompoundVars::addCERC20()

Recommendation Apply necessary validation on the given input and ensure no duplicate will be introduced.

Status The issue has been fixed by this commit: `32665ee`.

3.3 Inconsistency Between Document and Implementation

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

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 at line 198 of `KComptroller::provideBuffer()`. The preceding function summary indicates that this function is designed to *"remove tokens from the user's buffer."* However, the implementation logic (lines 205 – 207) indicates the opposite in providing additional buffer to the position of `msg.sender`.

```
460     /// @notice remove tokens from the user's buffer, this function is used to manage
461     ///         the total amount of buffer provided by a contract.
462     /// @dev this function only takes care of bookkeeping, not the actual buffer
463     ///      transfers.
464     ///
465     /// @param _cToken The address of the cToken contract.
466     /// @param _amount The amount of cTokens provided by JITU.
467     function provideBuffer(address _cToken, uint256 _amount) override external {
468         buffer[msg.sender][address(_cToken)] = buffer[msg.sender][address(_cToken)].add(
                _amount);
469     }
```

Listing 3.3: KComptroller:: provideBuffer ()

Also, we notice inconsistency at line 256 of `KComptroller::tokenBalance()` and line 303 of `KComptroller::collateralRemoveEffect()`. In particular, the function `KComptroller::tokenBalance()` returns the current, not outstanding, balance of the given `CToken`'s underlying token.

```
256     /// @notice calculate the given token's outstanding balance for the given account.
257     ///
258     /// @param _cToken The address of the cToken contract.
259     ///
260     /// @return Outstanding balance of the given token.
261     function tokenBalance(address _account, CToken _cToken) override external returns (
            uint256) {
262         Exp memory exchangeRate = Exp({mantissa: _cToken.exchangeRateCurrent()});
263         (MathError mErr, uint balance) = mulScalarTruncate(exchangeRate, _cToken.
                balanceOf(_account).sub(
264                     buffer[_account][address(_cToken)]
265                 )
266         );
```

```
267         require(mErr == MathError.NO_ERROR, "KComptroller: balance could not be
                calculated");
268         return balance;
269     }
```

Listing 3.4: KComptroller::tokenBalance()

The `KComptroller::collateralRemoveEffect()` function calculates the change in liquidity if the given amount of underlying tokens are removed, instead of the amount of the `cTokens`.

```
302     /** Internal Functions */
303     /// @notice calculate the change in liquidity if the given amount of cTokens are
            removed
304     function collateralRemoveEffect(CToken cToken, uint256 amount) internal view returns
            (uint256) {
305         (, uint collateralFactor, ) = compoundVars.comptroller().markets(address(cToken)
                );
306         (MathError mErr, uint256 colRemoveEffect) = mulScalarTruncate(Exp({mantissa:
                collateralFactor}), valueInUSD(cToken, amount));
307         require(mErr == MathError.NO_ERROR, "KComptroller: redeem effect could not be
                calculated");
308         return colRemoveEffect;
309     }
```

Listing 3.5: KComptroller:: collateralRemoveEffect ()

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

Status The issue has been fixed by this commit: `32665ee`.

3.4 Orange vs. Apple in checkBufferValue()

- ID: PVE-004
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `KComptroller`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

As mentioned in Section 3.1, there is a `KComptroller` contract that acts as a gateway to various functionality in `KCompoundPosition`. In this section, we examine a core helper routine `checkBufferValue()` that is designed to validate the given amount of additional buffer will not exceed the preset threshold, i.e., the total buffer should be no more than 25% of the total collateral. Note that the

worst case scenario is when the KeeperDAO keeper fails to preempt a liquidation while the account is liquidated by an external keeper.

To illustrate, we show below the checkBufferValue() helper routine. This helper routine implements a rather straightforward logic in deriving the current underlying position balance (line 246) with the provided buffer and verifying whether the additional buffer would result in an undesirably higher percentage of the total collateral, i.e., more than 25%.

```
232     /// @notice maximum amount of buffer that can be provided should
233     ///         be 25% of the total collateral, as the worst case
234     ///         scenario where the KeeperDAO keeper fails to preempt
235     ///         a liquidation and the account is liquidated by an
236     ///         external keeper. The buffer should still be reclaimed.
237     /// @dev this function would be reverted if the amount being
238     ///      provided as buffer exceeds max buffer value.
239     ///
240     /// @param _token the address of the ERC20 token, in which the
241     ///               buffer is being provided.
242     /// @param _amount the amount of ERC20 tokens that are being
243     ///                provided.
244     function checkBufferValue(address _token, uint256 _amount) override external {
245         CToken cToken = compoundVars.cTokenWrapper(_token);
246         uint256 walletTokenBalanace = cToken.balanceOfUnderlying(msg.sender);
247         uint256 bufferProvided = buffer[msg.sender][address(cToken)];
248         require((_amount + bufferProvided) * 3 <= (walletTokenBalanace - bufferProvided)
                ,
249             "buffer exceeds 25% of the user's ctoken balance");
250     }
```

Listing 3.6: KComptroller::checkBufferValue()

The analysis with the above helper routine indicates the given _amount represents the amount of underlying ERC20 tokens, not the amount of CTokens. However, the addition of _amount + bufferProvided (line 248) makes use of the bufferProvided amount of CTokens, leading to an orange vs apple situation! A proper validation requires the conversion of the CTokens amount to the corresponding amount of the underlying ERC20 token. Only after the conversion, we can then properly calculate the buffer share in the total collateral and require the buffer share does not exceed the allowed threshold of 25%.

Recommendation Apply the correct computation to ensure that the provided buffer does not exceed the maximum, permitted range in checkBufferValue().

Status The issue has been fixed by this commit: 32665ee.

3.5 Naming Consistency In KComptroller

- ID: PVE-005
- Severity: Informational
- Likelihood: Low
- Impact: N/A
- Target: `KComptroller`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

Description

With the introduction of a wrapper token, e.g., `CTokens`, of the underlying ERC20 tokens, there is a natural need to convert tokens from one to another. Also, a set of functions may take as arguments the `CTokens` amount and another set may take the amount of the underlying ERC20 tokens as input. To avoid unnecessary confusions, the `KComptroller` contract is designed to differentiate these two types of tokens. In particular, it is stated that "through out this contract we use xxxTokens for cTokens and xxxAmount for value in underlying token" as shown below with the `KComptroller` contract definition.

```
10 /// @dev this contract checks if a user can borrow/redeem without considering
11 ///      the buffer added by the KeeperDAO's underwriter.
12 /// @dev this contract allows KeeperDAO's friendly keepers to preempt
13 ///      liquidations, that are underwater after removing the buffer provided
14 ///      by the KeeperDAO's underwriter so that it does not have to go through
15 ///      PGAs.
16 /// @dev this contract keeps track of the buffer provided by the KeeperDAO
17 ///      underwriters.
18 /// @dev through out this contract we use xxxTokens for cTokens and xxxAmount
19 ///      for value in underlying token.
20 contract KComptroller is ComptrollerErrorReporter, Exponential, IKComptroller {
21     ...
22 }
```

Listing 3.7: The KComptroller Contract

We have analyzed the full set of functions defined in the `KComptroller` contract and realize that the above naming convention is not followed. As an example, we show below the `provideBuffer()` routine. This routine takes as input the amount of cTokens provided by `Just-In-Time-Underwriter` (`JITU`). The convention would suggest the use of `_tokens`, instead of `_amount`.

```
198     /// @notice remove tokens from the user's buffer, this function is used to manage
199     ///         the total amount of buffer provided by a contract.
200     /// @dev this function only takes care of bookkeeping, not the actual buffer
201     ///      transfers.
202     ///
203     /// @param _cToken The address of the cToken contract.
204     /// @param _amount The amount of cTokens provided by JITU.
205     function provideBuffer(address _cToken, uint256 _amount) override external {
```

```
206         buffer[msg.sender][address(_cToken)] = buffer[msg.sender][address(_cToken)].add(
                _amount);
207     }
```

Listing 3.8: KComptroller:: provideBuffer ()

Also, according to the function head of `collateralRemoveEffect()`, it is proposed to calculate the change in liquidity if the given amount of cTokens are removed. In this case, the second argument of `collateralRemoveEffect()` should be `_tokens` as well.

Recommendation Follow the convention of differentiating the `CToken` amount from the amount of the underlying ERC20 tokens.

Status The issue has been fixed by this commit: `32665ee`.

3.6 Accommodation of approve() Idiosyncrasies

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Business Logic [4]
- CWE subcategory: N/A

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 `approve()` 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. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
194     /**
195     * @dev Approve the passed address to spend the specified amount of tokens on behalf
            of msg.sender.
196     * @param _spender The address which will spend the funds.
197     * @param _value The amount of tokens to be spent.
198     */
199     function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

201         // To change the approve amount you first have to reduce the addresses`
```

```
202         //  allowance to zero by calling `approve(_spender, 0)` if it is not
203         //  already 0 to mitigate the race condition described here:
204         //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
205         require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

207         allowed[msg.sender][_spender] = _value;
208         Approval(msg.sender, _spender, _value);
209     }
```

Listing 3.9: USDT Token Contract

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. For example, the `KCompoundPosition::_deposit()` routine is designed to pull funds from the given `_from`, deposit into the `Compound` pool, and returns back the minted `CToken`-based pool tokens. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

```
263     /// @dev _deposit pulls ERC20 tokens from `_from` address
264     /// and deposits  them to compound and returns the change
265     /// in compound token balance.
266     function _deposit(address _from, address _token, uint256 _amount) internal returns (
            uint256) {
267         require(address(vars.cTokenWrapper(_token)) != address(0x0), "KCompoundPosition:
                token is not registered");
268         require(_amount != 0, "KCompoundPosition: non-zero amount required");

270         ERC20(_token).safeTransferFrom(_from, address(this), _amount);
271         uint256 initialBalance = vars.cTokenWrapper(_token).balanceOf(address(this));
272         if (vars.weth() == _token) {
273             // Convert WETH to ETH before depositing to compound.
274             Weth(vars.weth()).withdraw(_amount);
275             vars.cEther().mint{value: _amount}();
276         } else {
277             ERC20(_token).safeApprove(address(vars.cTokenMap(_token)), _amount);
278             requireNoError(vars.cTokenMap(_token).mint(_amount), "KCompoundPosition:
                    failed to deposit erc20 tokens");
279         }
280         uint256 finalBalance = vars.cTokenWrapper(_token).balanceOf(address(this));
281         return finalBalance - initialBalance;
282     }
```

Listing 3.10: KCompoundPosition::_deposit()

Meanwhile, it is important to highlight that the current implementation is safe as far as the one-time `safeApprove()` is always followed by the full transfer of the approved amount, which effectively reduces the approved amount back to zero. However, to accommodate various situations, it is always suggested to follow the convention of applying the `approve()` call twice to ensure the operation always runs smoothly.

Recommendation Accommodate the above-mentioned idiosyncrasy of `approve()`.

Status The issue has been confirmed. Considering the fact that every `approve()` is always consumed by the following `transferFrom()`, the team decides to leave it as is.

3.7 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-007
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `JITU`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In the KCompound protocol, the `JITU` contract is the gateway contract for users to manage compound positions. As the name of `JITU` indicates, it provides just-in-time underwriting functionality and the contract is designed to be the one a normal user has to interact with to create and manage their compound positions. Therefore, this contract has entry routines, i.e., `deposit()/withdraw()` to accept or withdraw user deposits of supported assets (e.g., `DAI`). Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the `JITU` contract. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
73      /// @notice Deposit funds to the user's Compound Wallet.
74      ///
75      /// @param token The address of the token contract.
76      /// @param amount The value of the amount being deposited.
77      function deposit(address token, uint256 amount) external override isInitialized  {
78          require(amount != 0, "KCompound: non-zero amount required");
79          require(address(compoundVars.cTokenWrapper(token)) != address(0), "KCompound:
                token is not registered");
80
81          compoundPositions[_msgSender()].deposit(token, amount);
82      }
83
84      /// @notice Repay funds to the user's Compound Wallet.
85      ///
86      /// @param token The address of the token contract.
87      /// @param amount The value of partial loan.
88      function repay(address token, uint256 amount) external override isInitialized {
89          require(amount != 0, "KCompound: non-zero amount required");
90          require(address(compoundVars.cTokenWrapper(token)) != address(0), "KCompound:
                token is not registered");
91
92          compoundPositions[_msgSender()].repay(token, amount);
```

```
93      }
94
95      /// @notice Withdraw funds from the user's Compound Wallet.
96      ///
97      /// @param to The address of the amount receiver.
98      /// @param token The address of the token contract.
99      /// @param amount The value of partial loan.
100     function withdraw(address to, address token, uint256 amount) external override
            isInitialized {
101         require(amount != 0, "KCompound: non-zero amount required");
102         require(address(compoundVars.cTokenWrapper(token)) != address(0), "KCompound:
                token is not registered");
103
104         compoundPositions[_msgSender()].withdraw(to, token, amount);
105     }
```

Listing 3.11: Related Functions In JITU

However, as mentioned in Section 3.6, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as YAM.) As a result, this may not meet the assumption behind these low-level asset-transferring routines.

One possible mitigation is to regulate the set of ERC20 tokens that are permitted into the JITU. In our case, it is indeed possible to effectively regulate the set of tokens that can be supported. Keep in mind that there exist certain assets (e.g., USDT) that may have control switches that can be dynamically exercised to suddenly become one.

Recommendation If current codebase needs to support possible deflationary tokens, it is better to check the balance before and after the `transfer()/transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is widely-adopted USDT.

Status This issue has been confirmed. However, considering the fact that this specific issue does not affect the normal operation, the team decides to address it when the need of supporting deflationary/rebasing tokens arises.

3.8 Incorrect Calculation of KComptroller::clearBuffer()

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `KComptroller`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

As mentioned in Section 3.1 and Section 3.4, the `KComptroller` contract acts as a gateway to various functionality in `KCompoundPosition`. In this section, we examine a specific routine `clearBuffer()` that is designed to manage the total amount of buffer provided by a contract when the buffer is removed. However, it should be mentioned that this function serves as the bookkeeping purposes, without actually removing the buffer. To elaborate, we show below the function's implementation.

```
209     /// @notice remove tokens from the user's buffer, this function is used to manage
210     ///         the total amount of buffer provided by a contract.
211     /// @dev this function only takes care of bookkeeping, not the actual buffer
212     ///      transfers.
213     ///
214     /// @param _cToken The address of the cToken contract.
215     ///
216     /// @return cTokenBalance that can be redeemed to repay the buffer.
217     function clearBuffer(address _cToken) override external returns (uint256) {
218         uint256 bufferAmount = buffer[msg.sender][address(_cToken)];
219         if (bufferAmount == 0) return 0;

221         // calculate the effect removing this buffer would create
222         uint256 removeBufferEffect = collateralRemoveEffect(CToken(_cToken),
                bufferAmount);

224         // check if removing this buffer would take the position under water
225         (, uint shortfall) = getAccountLiquidity(msg.sender, removeBufferEffect);
226         require(shortfall == 0, "KComptroller: insufficient liquidity to remove buffer")
                ;

228         buffer[msg.sender][address(_cToken)] = 0;
229         return bufferAmount;
230     }
```

Listing 3.12: KComptroller:: clearBuffer ()

We notice the function calls another helper routine `getAccountLiquidity()` to check whether the buffer removal will result in an underwater position. For that, it needs to calculate the effect that may be caused by removing this buffer. The calculation is performed as `collateralRemoveEffect (CToken(_cToken), bufferAmount)`. It comes to our attention that the `bufferAmount` is in terms of

CToken amount, not the underlying token amount. More importantly, there is no need to consider removeBufferEffect as the getAccountLiquidity() function properly takes into account the bufferAmount.

Recommendation Revise the above calculations to properly validate whether the buffer-associated position is under water after the buffer removal.

Status The issue has been fixed by this commit: 7d07aad.

4 Conclusion

In this audit, we have analyzed the KCompound design and implementation. The system presents a unique, robust offering as an on-chain liquidity underwriting engine that provides buffer to the Compound positions, which are close to being liquidated. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

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

References

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

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

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

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

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

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

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