SMART CONTRACT AUDIT REPORT

for

COZY FINANCE

Prepared By: Yiqun Chen

PeckShield

July 2, 2021

Document Properties

Client	Cozy Finance
Title	Smart Contract Audit Report
Target	Cozy Protocol
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 2, 2021	Xuxian Jiang	Final Release
1.0-rc2	June 18, 2021	Xuxian Jiang	Release Candidate #2
1.0-rc1	May 28, 2021	Xuxian Jiang	Release Candidate #1
0.3	May 27, 2021	Xuxian Jiang	Add More Findings #2
0.2	May 14, 2021	Xuxian Jiang	Add More Findings #1
0.1	May 10, 2021	Xuxian Jiang	Initial Draft

Contact

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

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

Contents

1 Introduction

Given the opportunity to review the Cozy protocol 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 Cozy Protocol

The Cozy protocol allows for permissionless, peer-to-peer protection from smart contract failure by enabling an algorithmic money market protocol, which extends Compound with the support of protection markets. Users may supply funds to protection markets to earn interest, which provides protection to other users. And funds borrowed from protection markets are protected because they do not need to be repaid when a trigger event occurs. Protocol users operate the same as Compound by supplying funds to money markets and then use those funds as collateral to borrow from protection markets, in addition to current choice of borrowing regular, unprotected funds from money markets if desired.

The basic information of Cozy Protocol is as follows:

Table 1.1: Basic Information of Cozy Protocol

Item	Description
Issuer	Cozy Finance
Website	https://cozy.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 2, 2021

In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit. Note that Cozy Protocol 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/Cozy-Finance/cozy-protocol.git (a8a68b7)
- https://github.com/Cozy-Finance/cozy-invest-contracts.git (a210d97)

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

- https://github.com/Cozy-Finance/cozy-protocol.git (0c951aa)
- https://github.com/Cozy-Finance/cozy-invest-contracts.git (d5f79267)

1.2 About PeckShield

PeckShield Inc. [14] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

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

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;

- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

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

To evaluate the risk, we go through a 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) [12], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. 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.

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.

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 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 Cozy Protocol 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	1	■
Medium	4	■■■■
Low	3	■■■
Informational	1	■
Total	9	

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 4 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Cozy Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Associated Risks For Protection Market Suppliers	Business Logic	Confirmed
PVE-002	Medium	Sandwich Attacks For Maximum Contributor Rewards	Time And State	Fixed
PVE-003	Medium	Non ERC20-Compliance Of CToken	Coding Practices	Confirmed
PVE-004	Low	Possible Front-running For Unintended Payment In repayBorrowBehalf()	Time And State	Confirmed
PVE-005	Low	Interface Inconsistency Between CErc20 And CEther	Coding Practice	Confirmed
PVE-006	Informational	Redundant State/Code Removal	Coding Practice	Confirmed
PVE-007	Low	Suggested Improvement Of _setCollateralFactor()	Business Logics	Confirmed
PVE-008	Medium	Possible DoS Against Liquidation	Business Logic	Fixed
PVE-009	High	Possible Risks From Evil interestRateModel of Protection Market	Business Logic	Confirmed

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.

3 Detailed Results

3.1 Associated Risks For Protection Market Suppliers

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `CToken`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [7]

Description

The `Cozy` protocol has a unique feature in supporting the so-called protection markets. And by design, any one is allowed to deploy a new protection market with any trigger and the protection market is open to all users to supply the liquidity. A protection market is distinguished from an ordinary money market with the presence of a non-zero value for the `trigger` address property in the `CToken` contract

Due to the permissionless nature of protection market deployment and activation, we need to highlight certain risks that are associated with supplying assets into an untrusted protection market. In particular, if a malicious actor deploys a protection market and a user supplies assets into the protection market, the malicious actor (or an accomplice) may borrow all possible funds from the protection market. After that, the malicious actor may declare the trigger event has occurred, which exempts current borrowers from repaying any borrowed funds, resulting in loss for the supplying users.

Recommendation Properly vet deployed protection markets before their activation.

Status This issue has been confirmed.

3.2 Sandwich Attacks For Maximum Contributor Rewards

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `LinearCappedCozySpeedAugmentor`
- Category: Time and State [11]
- CWE subcategory: CWE-682 [6]

Description

The `Cozy` protocol is forked from `Compound` and shares a central core contract, i.e., `Comptroller`. While this contract acts as a policy enforcement supervisor, it can also be used to configure and disseminate the protocol tokens `COZY`. In the following, we examine the logic behind the specific contributor rewards.

To elaborate, we show the `Comptroller::updateContributorRewards()` routine, which is used to compute additional accrued `COZY` rewards for a contributor since the last accrual. The computation is delegated to a so-called `cozySpeedAugmentor` to calculate the final COZY speed used for that contributor, and the address of the augmentor used can be updated via governance.

```
1310  function updateContributorRewards(address contributor) public {
1311    // Get COZY speed based on base rate stored in Comptroller and augmented value
          computed externally
1312    uint256 baseSpeed = cozyContributorSpeeds[contributor];
1313    CToken[] memory markets = protectionMarkets[contributor];
1314    bool hasAugmentor = address(cozySpeedAugmentor) != address(0);
1315    uint256 cozySpeed = hasAugmentor ? cozySpeedAugmentor.getCozySpeed(contributor,
          baseSpeed, markets) : baseSpeed;
1316
1317    // Update rewards
1318    uint256 blockNumber = getBlockNumber();
1319    uint256 deltaBlocks = sub_(blockNumber, lastContributorBlock[contributor]);
1320    if (deltaBlocks > 0 && cozySpeed > 0) {
1321      uint256 newAccrued = mul_(deltaBlocks, cozySpeed);
1322      uint256 contributorAccrued = add_(cozyAccrued[contributor], newAccrued);
1323
1324      cozyAccrued[contributor] = contributorAccrued;
1325      lastContributorBlock[contributor] = blockNumber;
1326    }
1327  }
```

Listing 3.1: Comptroller::updateContributorRewards()

Our analysis with the given `LinearCappedCozySpeedAugmentor` may be exploited to always return the maximum contributor rewards. Specifically, the computed `COZY` speed depends on current total borrows of the related protection market. However, this number can be manipulated via borrowing the

borrowAmountForMaxCozySpeed right before calling updateContributorRewards() and then immediately returning the borrowed amount within the same transaction, with zero interest!

```
76  function getCozySpeed(
77    address _contributor,
78    uint256 _baseSpeed,
79    CToken[] calldata _markets
80  ) external view returns (uint256) {
81    // Add COZY for each deployed protection market
82    uint256 _additionalSpeed = 0;
83
84    // Get the current oracle
85    PriceOracle _oracle = ComptrollerOracleInterface(comptroller).oracle();
86
87    for (uint256 i = 0; i < _markets.length; i++) {
88      // Skip if this market has been triggered
89      CToken _market = _markets[i];
90      if (_market.isTriggered()) {
91        continue;
92      }
93
94      // Get the total USD borrows in the market
95      uint256 _price = _oracle.getUnderlyingPrice(_market);
96      uint256 _totalBorrows = _market.totalBorrows().mul(_price); // total borrows in
          USD
97
98      // Compute amount that should be added. When calculating _speedToAdd we don't need
           SafeMath. This is because
99      // the multiplciation only happens if _totalBorrows < _ceiling, so as long as the
          `borrowAmountForMaxCozySpeed`
100     // and `maxCozySpeed` values are set correctly and reasonably, that multiplication
           will never overflow.
101     uint256 _maxSpeed = maxCozySpeed[address(_market)]; // max COZY speed for the
          market
102     uint256 _ceiling = borrowAmountForMaxCozySpeed[(address(_market))]; // USD borrow
          amount at which maxSpeed is given
103     uint256 _speedToAdd = _totalBorrows >= _ceiling ? _maxSpeed : (_maxSpeed *
          _totalBorrows) / _ceiling;
104     _additionalSpeed = _additionalSpeed.add(_speedToAdd);
105   }
106
107   // Return computed COZY speed
108   return _baseSpeed.add(_additionalSpeed);
109 }
```

Listing 3.2: LinearCappedCozySpeedAugmentor::getCozySpeed()

Recommendation Develop an effective mitigation to the above sandwich attack to ensure the proper computation and dissemination of protocol tokens.

Status The issue has been fixed by the following merge request: 53.

3.3 Non ERC20-Compliance Of CToken

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `CToken`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [3]

Description

Each asset supported by the `Cozy` protocol is integrated through a so-called `cToken` contract, which is an ERC20 compliant representation of balances supplied to the protocol. By minting `cTokens`, users can earn interest through the `cToken`'s exchange rate, which increases in value relative to the underlying asset, and further gain the ability to use `cTokens` as collateral. There are currently two types of `cTokens`: `CErc20` and `CEther`. In the following, we examine the ERC20 compliance of these `cTokens`.

Table 3.1: Basic `View-Only` Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as part of our audit, we

examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Table 3.2: Key `State-Changing` Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	×
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	×
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	×
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	✓
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to $address(0x0)$ when new tokens are generated	✓
Approval() event	Is emitted on any successful call to approve()	✓

Our analysis shows that there are several ERC20 inconsistency or incompatibility issues found in the `cToken` contract. Specifically, the current `transfer()` function simply returns the related error code if the sender does not have sufficient balance to spend. A similar issue is also present in the `transferFrom()` function that does not revert when the sender does not have the sufficient balance or the message sender does not have the enough allowance.

Moreover, the ERC20 token standard also specifies that "a token contract which creates new tokens SHOULD trigger a `Transfer` event with the `_from` address set to `0x0` when tokens are created." [1] However, current `mint()` logic emits the `Transfer` event by specifying the `CErc20/CEther` contract itself as the `_from`. For better ERC20 compliance, it is also suggested to strictly follow the

ERC20 token standard.

In the surrounding two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` functions (Table 3.2) according to the widely-adopted ERC20 specification. In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777/ERC2222), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Table 3.3: Additional `Opt-in` Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on transfer()/transferFrom() calls	—
Rebasing	The balanceOf() function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
Pausable	The token contract allows the owner or privileged users to pause the token transfers and other operations	✓
Blacklistable	The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited	—
Mintable	The token contract allows the owner or privileged users to mint tokens to a specific address	✓
Burnable	The token contract allows the owner or privileged users to burn tokens of a specific address	✓

Recommendation Revise the `CToken` implementation to ensure its ERC20-compliance.

Status This issue has been confirmed. Considering that this is part of the original `Compound` code base, the team decides to leave it as is to minimize the difference from the original `Compound` and reduce the risk of introducing bugs as a result of changing the behavior.

3.4 Possible Front-running For Unintended Payment In repayBorrowBehalf()

- ID: PVE-004
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: CToken
- Category: Time and State [10]
- CWE subcategory: CWE-663 [5]

Description

The Cozy protocol is in essence an over-collateralized lending pool that has the lending functionality and supports a number of normal lending functionalities for supplying and borrowing users, i.e., mint()/redeem() and borrow()/repay(). In the following, we examine one specific functionality, i.e., repay().

To elaborate, we show below the core routine repayBorrowFresh() that actually implements the main logic behind the repay() routine. This routine allows for repaying partial or full current borrowing balance. It is interesting to note that the Cozy protocol supports the payment on behalf of another borrowing user (via repayBorrowBehalf()). And the repayBorrowFresh() routine supports the corner case when the given amount is larger than the current borrowing balance. In this corner case, the protocol assumes the intention for a full repayment.

```
1047  function repayBorrowFresh(
1048    address payer,
1049    address borrower,
1050    uint256 repayAmount
1051  ) internal whenNotTriggered returns (uint256, uint256) {
1052    /* Fail if repayBorrow not allowed */
1053    uint256 allowed = comptroller.repayBorrowAllowed(address(this), payer, borrower,
          repayAmount);    uint256 allowed = comptroller.repayBorrowAllowed(address(this),
           payer, borrower, repayAmount);
1054    if (allowed != 0) {
1055      return (failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.
            REPAY_BORROW_COMPTROLLER_REJECTION, allowed), 0);
1056    }

1058    /* Verify market's block number equals current block number */
1059    if (accrualBlockNumber != getBlockNumber()) {
1060      return (fail(Error.MARKET_NOT_FRESH, FailureInfo.REPAY_BORROW_FRESHNESS_CHECK), 0)
            ;
1061    }

1063    RepayBorrowLocalVars memory vars;

1065    /* We remember the original borrowerIndex for verification purposes */
```

```
1066      vars.borrowerIndex = accountBorrows[borrower].interestIndex;

1068      /* We fetch the amount the borrower owes, with accumulated interest */
1069      (vars.mathErr, vars.accountBorrows) = borrowBalanceStoredInternal(borrower);
1070      if (vars.mathErr != MathError.NO_ERROR) {
1071        return (
1072          failOpaque(
1073            Error.MATH_ERROR,
1074            FailureInfo.REPAY_BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED,
1075            uint256(vars.mathErr)
1076          ),
1077          0
1078        );
1079      }

1081      /* If repayAmount == -1, repayAmount = accountBorrows */
1082      if (repayAmount == uint256(-1)) {
1083        vars.repayAmount = vars.accountBorrows;
1084      } else {
1085        vars.repayAmount = repayAmount;
1086      }

1088      ////////////////////////
1089      // EFFECTS & INTERACTIONS
1090      // (No safe failures beyond this point)

1092      /*
1093       * We call doTransferIn for the payer and the repayAmount
1094       *  Note: The cToken must handle variations between ERC-20 and ETH underlying.
1095       *  On success, the cToken holds an additional repayAmount of cash.
1096       *  doTransferIn reverts if anything goes wrong, since we can't be sure if side
              effects occurred.
1097       *   it returns the amount actually transferred, in case of a fee.
1098       */
1099      vars.actualRepayAmount = doTransferIn(payer, vars.repayAmount);

1101      /*
1102       * We calculate the new borrower and total borrow balances, failing on underflow:
1103       *  accountBorrowsNew = accountBorrows - actualRepayAmount
1104       *  totalBorrowsNew = totalBorrows - actualRepayAmount
1105       */
1106      (vars.mathErr, vars.accountBorrowsNew) = subUInt(vars.accountBorrows, vars.
          actualRepayAmount);
1107      require(vars.mathErr == MathError.NO_ERROR, "
          REPAY_BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED");

1109      (vars.mathErr, vars.totalBorrowsNew) = subUInt(totalBorrows, vars.actualRepayAmount)
          ;
1110      require(vars.mathErr == MathError.NO_ERROR, "
          REPAY_BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED");

1112      /* We write the previously calculated values into storage */
```

```
1113        accountBorrows[borrower].principal = vars.accountBorrowsNew;
1114        accountBorrows[borrower].interestIndex = borrowIndex;
1115        totalBorrows = vars.totalBorrowsNew;

1117        /* We emit a RepayBorrow event */
1118        emit RepayBorrow(payer, borrower, vars.actualRepayAmount, vars.accountBorrowsNew,
                vars.totalBorrowsNew);

1120        return (uint256(Error.NO_ERROR), vars.actualRepayAmount);
1121    }
```

Listing 3.3: CToken::repayBorrowFresh()

This is a reasonable assumption, but our analysis shows this assumption may be taken advantage of to launch a front-running `borrow()` operation, resulting in a higher borrowing balance for repayment. To avoid this situation, it is suggested to disallow the repayment amount of -1 to imply the full repayment. In fact, it is always suggested to use the exact payment amount in the `repayBorrowBehalf()` case.

Recommendation Revisit the generous assumption of using repayment amount of -1 as the indication of full repayment.

Status This issue has been confirmed. Considering the given amount is the choice from the repayer, the team decides to leave it as is.

3.5 Interface Inconsistency Between CErc20 And CEther

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1041 [2]

Description

As mentioned in Section 3.3, each asset supported by the `Cozy` protocol is integrated through a so-called `cToken` contract, which is an ERC20 compliant representation of balances supplied to the protocol. And `cTokens` are the primary means of interacting with the `Cozy` protocol when a user wants to `mint()`, `redeem()`, `borrow()`, `repay()`, `liquidate()`, or `transfer()`. Moreover, there are currently two types of `cTokens`: `CErc20` and `CEther`. Both types expose the ERC20 interface and they wrap an underlying `ERC20` asset and `Ether`, respectively.

While examining these two types, we notice their interfaces are surprisingly different. Using the `replayBorrow()` function as an example, the `CErc20` type returns an error code while the `CEther` type

simply reverts upon any failure. The similar inconsistency is also present in other routines, including `repayBorrowBehalf()`, `mint()`, and `liquidateBorrow()`.

```
88  /**
89   * @notice Sender repays their own borrow
90   * @param repayAmount The amount to repay
91   * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
92   */
93  function repayBorrow(uint256 repayAmount) external returns (uint256) {
94    (uint256 err, ) = repayBorrowInternal(repayAmount);
95    return err;
96  }
```

Listing 3.4: CErc20::repayBorrow()

```
82  /**
83   * @notice Sender repays their own borrow
84   * @dev Reverts upon any failure
85   */
86  function repayBorrow() external payable {
87    (uint256 err, ) = repayBorrowInternal(msg.value);
88    requireNoError(err, "repayBorrow failed");
89  }
```

Listing 3.5: CEther::repayBorrow()

It is also worth mentioning that the `CErc20` type supports `_addReserves` while the `CEther` type does not.

Recommendation Ensure the consistency between these two types: `CErc20` and `CEther`.

Status This issue has been confirmed. Considering that this is part of the original `Compound` code base, the team decides to leave it as is to minimize the difference from the original `Compound` and reduce the risk of introducing bugs as a result of changing the behavior.

3.6 Redundant State/Code Removal

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [4]

Description

The `Cozy` protocol makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `SafeMath`, and `Address`, to facilitate its code implementation and organization. For example, the

Comptroller smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the cToken contract, there are a number of local variables that are defined, but not used. Examples include the err field in the defined MintLocalVars and RedeemLocalVars structures. Note a similar structure also named MintLocalVars in VAIToken and VAIController shares the same issue.

```
480     struct MintLocalVars {
481         Error err;
482         MathError mathErr;
483         uint exchangeRateMantissa;
484         uint mintTokens;
485         uint totalSupplyNew;
486         uint accountTokensNew;
487         uint actualMintAmount;
488     }
```

Listing 3.6: cToken::MintLocalVars

Moreover, the _acceptAdmin() routine in both Unitroller and cToken can be improved by removing the following redundant condition validation: msg.sender == address(0) (at lines 108 and 1405 respectively)

Recommendation Consider the removal of the redundant state (or code) with a simplified, consistent implementation.

Status This issue has been confirmed. Considering that this is part of the original Compound code base, the team decides to leave it as is to minimize the difference from the original Compound and reduce the risk of introducing bugs as a result of changing the behavior.

3.7 Suggested Improvement Of _setCollateralFactor()

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Comptroller
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [7]

Description

The introduction of protection markets in the Cozy protocol provides certain level of protection against borrowing users. However, there are inherent invariants that need to be upheld at all times. One

specific invariant is the need of zero collateral ratio (`collateralFactorMantissa`) of protection markets. If a protection market has a non-zero collateral ratio, it is possible for a malicious actor to deploy an evil protection market and use it to borrow funds from other money markets.

To dynamically configure the collateral ratio, the `Cozy` protocol provides an administrative function `_setCollateralFactor()`. This function can be improved to ensure the collateral ratio for currently deployed protection markets will not be changed.

```
940   function _setCollateralFactor(CToken cToken, uint256 newCollateralFactorMantissa)
          external returns (uint256) {
941     // Check caller is admin
942     if (msg.sender != admin) {
943       return fail(Error.UNAUTHORIZED, FailureInfo.SET_COLLATERAL_FACTOR_OWNER_CHECK);
944     }

946     // Verify market is listed
947     Market storage market = markets[address(cToken)];
948     if (!market.isListed) {
949       return fail(Error.MARKET_NOT_LISTED, FailureInfo.SET_COLLATERAL_FACTOR_NO_EXISTS);
950     }

952     Exp memory newCollateralFactorExp = Exp({mantissa: newCollateralFactorMantissa});

954     // Check collateral factor <= 0.9
955     Exp memory highLimit = Exp({mantissa: collateralFactorMaxMantissa});
956     if (lessThanExp(highLimit, newCollateralFactorExp)) {
957       return fail(Error.INVALID_COLLATERAL_FACTOR, FailureInfo.
              SET_COLLATERAL_FACTOR_VALIDATION);
958     }

960     // If collateral factor != 0, fail if price == 0
961     if (newCollateralFactorMantissa != 0 && oracle.getUnderlyingPrice(cToken) == 0) {
962       return fail(Error.PRICE_ERROR, FailureInfo.SET_COLLATERAL_FACTOR_WITHOUT_PRICE);
963     }

965     // Set market's collateral factor to new collateral factor, remember old value
966     uint256 oldCollateralFactorMantissa = market.collateralFactorMantissa;
967     market.collateralFactorMantissa = newCollateralFactorMantissa;

969     // Emit event with asset, old collateral factor, and new collateral factor
970     emit NewCollateralFactor(cToken, oldCollateralFactorMantissa,
            newCollateralFactorMantissa);

972     return uint256(Error.NO_ERROR);
973   }
```

Listing 3.7: Comptroller :: _setCollateralFactor ()

Recommendation Ensure that the `collateralFactorMantissa` of deployed protection markets stays at zero.

Status This issue has been confirmed. Considering that the setting of a collateral factor on protection markets can only be done by governance, the team assumes it is trusted to provide the right collateral factor of protection markets.

3.8 Possible DoS Against Liquidation

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Comptroller`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [7]

Description

The `Cozy` protocol has a core function `getHypotheticalAccountLiquidityInternal()` that is used to determine what the account liquidity would be if the given amounts were redeemed or borrowed. And this function is in the critical path behind the `borrow()` and `redeem()` logic. With the introduction of protection markets, the design of their permission-less deployment may be exploited to create a denial-of-service situation to prevent certain accounts from being liquidated.

To elaborate, we show below the `getHypotheticalAccountLiquidityInternal()` function. This function is rather complex by iterating all involved money markets and computing the overall safeness of the given account. With the introduction of protection markets, any one is able to create and deploy a protection market, which may make the iteration of involved money markets out-of-gas!

```
643   function getHypotheticalAccountLiquidityInternal(
644     address account,
645     CToken cTokenModify,
646     uint256 redeemTokens,
647     uint256 borrowAmount
648   )
649     internal
650     view
651     returns (
652       Error,
653       uint256,
654       uint256
655     )
656   {
657     AccountLiquidityLocalVars memory vars; // Holds all our calculation results
658     uint256 oErr;

660     // For each asset the account is in
661     CToken[] memory assets = accountAssets[account];
662     for (uint256 i = 0; i < assets.length; i++) {
663       CToken asset = assets[i];
```

```
665       // Read the balances and exchange rate from the cToken
666       (oErr, vars.cTokenBalance, vars.borrowBalance, vars.exchangeRateMantissa) = asset.
              getAccountSnapshot(account);
667       if (oErr != 0) {
668         // semi-opaque error code, we assume NO_ERROR == 0 is invariant between upgrades
669         return (Error.SNAPSHOT_ERROR, 0, 0);
670       }
671       vars.collateralFactor = Exp({mantissa: markets[address(asset)].
              collateralFactorMantissa});
672       vars.exchangeRate = Exp({mantissa: vars.exchangeRateMantissa});

674       // Get the normalized price of the asset
675       vars.oraclePriceMantissa = oracle.getUnderlyingPrice(asset);
676       if (vars.oraclePriceMantissa == 0) {
677         return (Error.PRICE_ERROR, 0, 0);
678       }
679       vars.oraclePrice = Exp({mantissa: vars.oraclePriceMantissa});

681       // Pre-compute a conversion factor from tokens -> ether (normalized price value)
682       vars.tokensToDenom = mul_(mul_(vars.collateralFactor, vars.exchangeRate), vars.
              oraclePrice);

684       // sumCollateral += tokensToDenom * cTokenBalance
685       vars.sumCollateral = mul_ScalarTruncateAddUInt(vars.tokensToDenom, vars.
              cTokenBalance, vars.sumCollateral);

687       // sumBorrowPlusEffects += oraclePrice * borrowBalance
688       vars.sumBorrowPlusEffects = mul_ScalarTruncateAddUInt(
689         vars.oraclePrice,
690         vars.borrowBalance,
691         vars.sumBorrowPlusEffects
692       );

694       // Calculate effects of interacting with cTokenModify
695       if (asset == cTokenModify) {
696         // redeem effect
697         // sumBorrowPlusEffects += tokensToDenom * redeemTokens
698         vars.sumBorrowPlusEffects = mul_ScalarTruncateAddUInt(
699           vars.tokensToDenom,
700           redeemTokens,
701           vars.sumBorrowPlusEffects
702         );

704         // borrow effect
705         // sumBorrowPlusEffects += oraclePrice * borrowAmount
706         vars.sumBorrowPlusEffects = mul_ScalarTruncateAddUInt(
707           vars.oraclePrice,
708           borrowAmount,
709           vars.sumBorrowPlusEffects
710         );
711       }
```

```
712     }
714     // These are safe, as the underflow condition is checked first
715     if (vars.sumCollateral > vars.sumBorrowPlusEffects) {
716       return (Error.NO_ERROR, vars.sumCollateral - vars.sumBorrowPlusEffects, 0);
717     } else {
718       return (Error.NO_ERROR, 0, vars.sumBorrowPlusEffects - vars.sumCollateral);
719     }
720   }
```

Listing 3.8: Comptroller:: getHypotheticalAccountLiquidityInternal ()

Specifically, a malicious actor may choose to deploy many protection markets (via `deployProtectionMarket()`), next borrow other assets via (`borrow()`), and finally enter these protection markets (via `enterMarkets()`). As far as the list of `accountAssets[actor]` is sufficiently long, which will eventually lead to the out-of-gas execution of `getHypotheticalAccountLiquidityInternal()`, hence preventing the malicious actor from being liquidated.

Recommendation Limit the total number of markets (`maxAssets`) that will be allowed for any account to enter.

Status The issue has been fixed by the following merge request: `50`.

3.9 Possible Risks From Evil interestRateModel of Protection Market

- ID: PVE-009
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `CToken`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [7]

Description

In Section 3.8, we have examined a possible denial-of-service issue that exploits the permissionless nature of protection market deployment. In this section, we further examine the protection market feature and report possible risks from its interest rate model, which may not be trusted.

In particular, our analysis shows that the protection market can be deployed with an arbitrary interest rate model. In other words, the interest rate model may subject to the deployer's full control. With that, if we revisit the way how the interest is accrued, a malicious interest model may suddenly change its behavior by charging an exorbitant borrow rate. One possible consequence will be that the borrowing user may suffer from immediate follow-up liquidation!

```
515   function accrueInterest() public returns (uint256) {
516     /* Short-circuit if trigger event occured */
517     if (isTriggered) {
518       accrualBlockNumber = getBlockNumber(); // required to allow redemptions
519       return uint256(Error.NO_ERROR);
520     }
521
522     /* Remember the initial block number */
523     uint256 currentBlockNumber = getBlockNumber();
524     uint256 accrualBlockNumberPrior = accrualBlockNumber;
525
526     /* Short-circuit accumulating 0 interest */
527     if (accrualBlockNumberPrior == currentBlockNumber) {
528       return uint256(Error.NO_ERROR);
529     }
530
531     /* Read the previous values out of storage */
532     uint256 cashPrior = getCashPrior();
533     uint256 borrowsPrior = totalBorrows;
534     uint256 reservesPrior = totalReserves;
535     uint256 borrowIndexPrior = borrowIndex;
536
537     /* Calculate the current borrow interest rate */
538     uint256 borrowRateMantissa = interestRateModel.getBorrowRate(cashPrior, borrowsPrior
          , reservesPrior);
539     require(borrowRateMantissa <= borrowRateMaxMantissa, "borrow rate too high");
540     ...
541   }
```

Listing 3.9: CToken::accrueInterest()

Recommendation Regulate the interest rate models that may be introduced for protection markets.

Status This issue has been confirmed.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the Cozy protocol. The protocol presents a unique, robust offering by providing a permissionless, peer-to-peer protection from smart contract failure. It enables an algorithmic money market protocol, which extends Compound with the support of protection markets. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, 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] Fabian Vogelsteller And Vitalik Buterin. EIP-20: ERC-20 Token Standard. https://eips.ethereum.org/EIPS/eip-20.

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

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

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

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

[6] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

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

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

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

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

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

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