SMART CONTRACT AUDIT REPORT

for

WING FINANCE

Prepared By: Shuxiao Wang

PeckShield

April 2, 2021

Document Properties

Client	Wing Finance
Title	Smart Contract Audit Report
Target	Wing Finance
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Huaguo Shi
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 2, 2021	Xuxian Jiang	Final Release
1.0-rc1	March 29, 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 **Wing Finance** 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 Wing Finance

The `Wing Finance` is a decentralized finance (DeFi) platform to support cross-chain collaborative interaction between various DeFi products. Combining the platform's decentralized governance model, the risk control mechanism introduced by `Wing` promotes a healthy, benign relationship between borrowers, creditors and guarantors, allowing the implementation of a wider range of DeFi plans on the platform and thus providing users with more DeFi products of premium quality. The audited system is an over-collateralized lending pool that is heavily inspired from the popular `Compound` protocol but with unique contributions by supporting a dedicated insurance pool.

The basic information of `Wing Finance` is as follows:

Table 1.1: Basic Information of Wing Finance

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

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

this audit. Note that Wing Finance 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/wing-groups/flash-pool-eth.git (7206689)

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

- https://github.com/wing-groups/flash-pool-eth.git (fb8be01)

1.2 About PeckShield

PeckShield Inc. [13] 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 [12]:

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

1.4 Disclaimer

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

Table 1.3: The Full Audit Checklist

Category	Checklist Items
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

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

Category	Summary
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 `Wing Finance` 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	2	■■
Medium	2	■■
Low	3	■■■
Informational	3	■■■
Total	10	

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

2.2 Key Findings

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

Table 2.1: Key Wing Finance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possible Front-running For Over Repay In repayBorrow()	Time And State	Confirmed
PVE-002	Low	Timely Refresh Of CompSpeeds In setInsuranceCompRateRatio()	Coding Practices	Fixed
PVE-003	Low	Gas Optimization Of Comptroller::borrowAllowed()	Coding Practices	Fixed
PVE-004	Medium	Recalculation Of Repay Amount Under Insufficient Insurance Balance	Business Logic	Fixed
PVE-005	Informational	Inconsistency Between Document and Implementation	Coding Practices	Fixed
PVE-006	High	Unintended Insurance Coverage Before Collateral Liquidation	Business Logic	Fixed
PVE-007	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-008	Informational	Redundant Code Removal	Coding Practices	Fixed
PVE-009	Informational	Improved Sanity Checks Of System Parameters	Coding Practices	Confirmed
PVE-010	High	Unintentional WING Collateral Transfer-In	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 Possible Front-running For Over Repay In repayBorrow()

- ID: PVE-001
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `CToken`
- Category: Time and State [10]
- CWE subcategory: CWE-663 [5]

Description

The `Wing Finance` protocol is in essence an over-collateralized lending pool that has the lending functionality similar to `Compound` 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 `Wing Finance` 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.

```
896     function repayBorrowFresh(address payer, address borrower, uint repayAmount)
            internal returns (uint, uint) {
897         /* Fail if repayBorrow not allowed */
898         uint allowed = comptroller.repayBorrowAllowed(address(this), borrower);
899         if (allowed != 0) {
900             return (failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.
                    REPAY_BORROW_COMPTROLLER_REJECTION, allowed), 0);
901         }
902         /* Verify market's block number equals current block number */
903         if (accrualBlockNumber != getBlockNumber()) {
904             return (fail(Error.MARKET_NOT_FRESH, FailureInfo.
                    REPAY_BORROW_FRESHNESS_CHECK), 0);
```

```
905         }
907         RepayBorrowLocalVars memory vars;
909         /* We remember the original borrowerIndex for verification purposes */
910         vars.borrowerIndex = accountBorrows[borrower].interestIndex;
912         /* We fetch the amount the borrower owes, with accumulated interest */
913         (vars.mathErr, vars.accountBorrows, vars.accountBorrowsValid) =
                borrowBalanceStoredInternal(borrower);
914         if (vars.mathErr != MathError.NO_ERROR) {
915             return (failOpaque(Error.MATH_ERROR, FailureInfo.
                    REPAY_BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED, uint(vars.mathErr))
                    , 0);
916         }
917         /* If repayAmount == -1, repayAmount = accountBorrows */
918         if (repayAmount == uint(-1)  repayAmount > vars.accountBorrows) {
919             vars.repayAmount = vars.accountBorrows;
920         } else {
921             vars.repayAmount = repayAmount;
922         }
923         ////////////////////////
924         // EFFECTS & INTERACTIONS
925         // (No safe failures beyond this point)
927         /*
928          * We call doTransferIn for the payer and the repayAmount
929          *  Note: The cToken must handle variations between ERC-20 and ETH underlying.
930          *  On success, the cToken holds an additional repayAmount of cash.
931          *  doTransferIn reverts if anything goes wrong, since we can't be sure if side
                effects occurred.
932          *   it returns the amount actually transferred, in case of a fee.
933          */
934         vars.actualRepayAmount = doTransferIn(payer, vars.repayAmount);
935         /*
936          * We calculate the new borrower and total borrow balances, failing on underflow
                :
937          *  accountBorrowsNew = accountBorrows - actualRepayAmount
938          *  totalBorrowsNew = totalBorrows - actualRepayAmount
939          */
940         (vars.mathErr, vars.accountBorrowsNew) = subUInt(vars.accountBorrows, vars.
                actualRepayAmount);
941         require(vars.mathErr == MathError.NO_ERROR, "
                REPAY_BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED");
942         if (vars.accountBorrowsValid <= vars.actualRepayAmount) {
943             vars.accountBorrowsValidNew = 0;
944             vars.actualRepayAmountValid = vars.accountBorrowsValid;
945         } else {
946             vars.actualRepayAmountValid = vars.actualRepayAmount;
947             (vars.mathErr, vars.accountBorrowsValidNew) = subUInt(vars.
                    accountBorrowsValid, vars.actualRepayAmount);
948             require(vars.mathErr == MathError.NO_ERROR, "
```

```
                REPAY_BORROW_NEW_ACCOUNT_VALID_BORROW_BALANCE_CALCULATION_FAILED");
949         }
950         if (totalValidBorrows < vars.actualRepayAmountValid) {
951             vars.totalBorrowsValidNew = 0;
952         } else {
953             (vars.mathErr, vars.totalBorrowsValidNew) = subUInt(totalValidBorrows, vars.
                    actualRepayAmountValid);
954             require(vars.mathErr == MathError.NO_ERROR, "
                    REPAY_BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED");
955         }
956         if (totalBorrows < vars.actualRepayAmount) {
957             vars.totalBorrowsNew = 0;
958         } else {
959             (vars.mathErr, vars.totalBorrowsNew) = subUInt(totalBorrows, vars.
                    actualRepayAmount);
960             require(vars.mathErr == MathError.NO_ERROR, "
                    REPAY_BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED");
961         }

963         /* We write the previously calculated values into storage */
964         accountBorrows[borrower].principal = vars.accountBorrowsNew;
965         accountBorrows[borrower].interestIndex = borrowIndex;
966         totalBorrows = vars.totalBorrowsNew;
967         totalValidBorrows = vars.totalBorrowsValidNew;
968         accountBorrows[borrower].validBorrow = vars.accountBorrowsValidNew;
969         /* We emit a RepayBorrow event */
970         emit RepayBorrow(payer, borrower, vars.actualRepayAmount, vars.accountBorrowsNew
                , vars.totalBorrowsNew);

972         comptroller.repayBorrowVerify(address(this), borrower);
973         return (uint(Error.NO_ERROR), vars.actualRepayAmount);
974     }
```

Listing 3.1: 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.

3.2 Timely Refresh Of CompSpeeds In setInsuranceCompRateRatio()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Comptroller`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The `Wing Finance` protocol has a built-in incentive mechanism that distributes the protocol token to both suppliers and borrowers. In the meantime, the protocol supports dynamic reconfiguration of a system-wide risk parameter `insuranceCompRateRatio`, which, as the name indicates, denotes the dissemination rate for the unique insurance pool. In the following, we show the corresponding `setter` function that allows for runtime configuration of `insuranceCompRateRatio`.

Specifically, this `setInsuranceCompRateRatio()` routine properly validates the given parameter and then delegates the call to an internal handler `refreshCompSpeedsInternal()`. The internal handler iterates each current market, next updates its supply index and borrow index, and then applies the insurance index. It comes to our attention that the `setInsuranceCompRateRatio()` routine invokes the internal handler (line 900) before saving the new `insuranceCompRateRatio` (line 901). In other words, the `refreshCompSpeedsInternal()` handler does not timely apply the new insurance index.

```
897     function setInsuranceCompRateRatio(uint ratio) public {
898         require(msg.sender == admin || msg.sender == operator, "only admin or operator
                can change comp rate");
899         require(ratio < expScale);
900         refreshCompSpeedsInternal();
901         insuranceCompRateRatio = ratio;
902     }
```

Listing 3.2: Comptroller :: setInsuranceCompRateRatio()

```
868     function refreshCompSpeedsInternal() internal {
869         CToken[] memory allMarkets_ = allMarkets;
870         for (uint i = 0; i < allMarkets_.length; i++) {
871             CToken cToken = allMarkets_[i];
872             Exp memory borrowIndex = Exp({mantissa : cToken.borrowIndex()});
873             updateCompSupplyIndex(address(cToken));
874             updateCompBorrowIndex(address(cToken), borrowIndex);
875         }
876         insurance.updateCompInsuranceIndexEx();
877         Exp memory totalUtility = Exp({mantissa : 0});
878         Exp[] memory utilities = new Exp[](allMarkets_.length);
879         for (uint i = 0; i < allMarkets_.length; i++) {
```

```
880             CToken cToken = allMarkets_[i];
881             if (markets[address(cToken)].isComped) {
882                 Exp memory assetPrice = Exp({mantissa : oracle.getUnderlyingPrice(cToken
                        .underlying())});
883                 Exp memory utility = mul_(assetPrice, cToken.totalValidBorrows());
884                 utilities[i] = mul_(utility, cToken.borrowRatePerBlock());
885                 totalUtility = add_(totalUtility, utilities[i]);
886             }
887         }
888         uint rate = compRate - getInsuranceCompRate();
889         for (uint i = 0; i < allMarkets_.length; i++) {
890             CToken cToken = allMarkets[i];
891             uint newSpeed = totalUtility.mantissa > 0 ? mul_(rate, div_(utilities[i],
                    totalUtility)) : 0;
892             compSpeeds[address(cToken)] = newSpeed / 2;
893         }
894     }
```

Listing 3.3: Comptroller::refreshCompSpeedsInternal()

Recommendation Revise the `setInsuranceCompRateRatio()` routine by updating the insurance index and then invoking the `refreshCompSpeedsInternal()` handler. An example revision is shown below.

```
897     function setInsuranceCompRateRatio(uint ratio) public {
898         require(msg.sender == admin || msg.sender == operator, "only admin or operator
                can change comp rate");
899         require(ratio < expScale);
900         insuranceCompRateRatio = ratio;
901         refreshCompSpeedsInternal();
902     }
```

Listing 3.4: Comptroller::setInsuranceCompRateRatio()

Status The issue has been fixed by this commit: `bfd6f07`.

3.3 Gas Optimization Of Comptroller::borrowAllowed()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Comptroller`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1041 [1]

Description

As mentioned in Section 3.1, the `Wing Finance` protocol is initially forked from the `Compound` protocol and follows the same architectural design in separating the policy enforcement in a standard-alone

contract, i.e., `Comptroller`. This `Comptroller` contract effectively acts as the gateway to various functionality provided in `CToken`. In the following, we examine one specific gateway function, i.e., `borrowAllowed()`.

As the name indicates, this `borrowAllowed()` function validates the incoming borrow request and determines whether the account should be allowed to borrow the underlying asset of the given market. It comes to our attention that the `Wing Finance` protocol supports a third `boolean` parameter `useWing`, which does not exist in the original `Compound` protocol. And it is interesting to notice that the statements at lines 291 – 303 can be effectively relocated into the `if`-branch when the boolean parameter is `true`. By doing so, we can at least save two external cross-contract calls querying for oracle prices (line 293 and 297) when the given parameter of `useWing` is `false`.

```
254     function borrowAllowed(address cToken, address borrower, uint borrowAmount, bool
            useWing) external returns (uint) {
255         // Pausing is a very serious situation - we revert to sound the alarms
256         require(!borrowGuardianPaused[cToken], "borrow is paused");
257
258         if (!markets[cToken].isListed) {
259             return uint(Error.MARKET_NOT_LISTED);
260         }
261
262         if (!markets[cToken].accountMembership[borrower]) {
263             // only cTokens may call borrowAllowed if borrower not in market
264             require(msg.sender == cToken, "sender must be cToken");
265
266             // attempt to add borrower to the market
267             Error err = addToMarketInternal(CToken(msg.sender), borrower);
268             if (err != Error.NO_ERROR) {
269                 return uint(err);
270             }
271             // it should be impossible to break the important invariant
272             assert(markets[cToken].accountMembership[borrower]);
273         }
274
275         if (oracle.getUnderlyingPrice(CToken(cToken).underlying()) == 0) {
276             return uint(Error.PRICE_ERROR);
277         }
278
279         (Error err, , uint shortfall,) = getHypotheticalAccountLiquidityInternal(
                borrower, CToken(cToken), 0, borrowAmount);
280         if (err != Error.NO_ERROR) {
281             return uint(err);
282         }
283         if (shortfall > 0) {
284             return uint(Error.INSUFFICIENT_LIQUIDITY);
285         }
286
287         // Keep the flywheel moving
288         Exp memory borrowIndex = Exp({mantissa : CToken(cToken).borrowIndex()});
289         updateCompBorrowIndex(cToken, borrowIndex);
```

```
290         distributeBorrowerComp(cToken, borrower, borrowIndex, false);
291         BorrowAllowedStruct memory vars;
292         // lock wing
293         vars.tokenPrice = oracle.getUnderlyingPrice(CToken(cToken).underlying());
294         if (vars.tokenPrice == 0) {
295             return uint(Error.PRICE_ERROR);
296         }
297         vars.wingPrice = oracle.getUnderlyingPrice(getCompAddress());
298         if (vars.wingPrice == 0) {
299             return uint(Error.PRICE_ERROR);
300         }
301         if (lockWingFactorMantissa == 0) {
302             return uint(Error.LOCK_WING_FACTOR_MANTISSA);
303         }
304         if (useWing) {
305             vars.token = mul_(Exp({mantissa : vars.tokenPrice}), Exp({mantissa :
                    borrowAmount}));
306             vars.lockAmt = mul_(Exp({mantissa : lockWingFactorMantissa}), vars.token);
307             vars.wingAmt = div_(vars.lockAmt, Exp({mantissa : vars.wingPrice}));
308             transferWingCollateralIn(address(cToken), borrower, vars.wingAmt.mantissa);
309         }
310         return uint(Error.NO_ERROR);
311     }
```

Listing 3.5: Comptroller::borrowAllowed()

Recommendation Apply the aforementioned gas optimization by avoiding unnecessary calls when the given parameter of `useWing` is `false`.

Status The issue has been fixed by this commit: `00cca98`.

3.4 Recalculation Of Repay Amount Under Insufficient Insurance Balance

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `KComptroller`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The `Wing Finance` protocol has a number of innovative enhancements beyond `Compound`. One enhancement is the introduction of an insurance pool. In particular, for an underwater position, the protocol allows for possible payment from the insurance pool. In the following, we examine this particular feature.

To illustrate, we show below the `IToken::_payInsuranceInternal()` routine that contains the main logic of insurance-based payment. It implements a rather straightforward logic in firstly repaying with the user's `WING` collateral (line 516) and then spending the insurance pool to cover a certain percentage of the remaining debt (line 518).

```
489     function _payInsuranceInternal(address market, address borrower, uint lockWingAmt)
            internal returns (uint, uint, uint, uint) {
490         require(msg.sender == address(comptroller));
491         uint oraclePriceMantissa = oracle.getUnderlyingPrice(CToken(market).underlying()
                );
492         if (oraclePriceMantissa == 0) {
493             return (uint(Error.PRICE_ERROR), 0, 0, 0);
494         }
495         RepayByInsuranceStruct memory vars;
496         vars.tokenPrice = Exp({mantissa : oraclePriceMantissa});
497
498         uint wingPriceMantissa = oracle.getUnderlyingPrice(getUnderlying());
499         if (wingPriceMantissa == 0) {
500             return (uint(Error.PRICE_ERROR), 0, 0, 0);
501         }
502         vars.wingPrice = Exp({mantissa : wingPriceMantissa});
503         (uint borrowBalance,) = CToken(market).borrowBalanceStored(borrower);
504         if (borrowBalance == 0) {
505             return (uint(Error.PRICE_ERROR), 0, 0, 0);
506         }
507         vars.temp = mul_(vars.tokenPrice, Exp({mantissa : borrowBalance}));
508         Exp memory debt_wing = div_(vars.temp, vars.wingPrice);
509         if (lockWingAmt >= debt_wing.mantissa) {
510             vars.borrowerPaidWing = debt_wing.mantissa;
511             vars.wholePaidWing = debt_wing.mantissa;
512             vars.repayAmount = borrowBalance;
513         } else {
514             // 1. use user whole wing collateral firstly
515             // 2. insurance pool repay the 70% remains debt
516             vars.borrowerPaidWing = lockWingAmt;
517             Exp memory temp = sub_(debt_wing, Exp({mantissa : vars.borrowerPaidWing}));
518             vars.insurancePaidWing = mul_(temp, Exp({mantissa : insuranceRepayFactor})).
                    mantissa;
519             vars.wholePaidWing = vars.insurancePaidWing + vars.borrowerPaidWing;
520             // calculate repay_amount
521             uint wingValue = mul_(vars.wholePaidWing, vars.wingPrice);
522             vars.repayAmount = div_(Exp({mantissa : wingValue}), vars.tokenPrice).
                    mantissa;
523             EIP20NonStandardInterface token = EIP20NonStandardInterface(getUnderlying())
                    ;
524             uint balance = token.balanceOf(address(this));
525             if (vars.insurancePaidWing > balance) {
526                 vars.insurancePaidWing = balance;
527             }
528             if (vars.insurancePaidWing > 0) {
529                 _payInsuranceInternal(vars.insurancePaidWing);
530             }
```

```
531         }
532         return (uint(Error.NO_ERROR), vars.borrowerPaidWing, vars.insurancePaidWing,
                vars.repayAmount);
533     }
```

Listing 3.6: IToken::_payInsuranceInternal()

However, when spending the insurance pool to cover the remaining pool, it is possible that there is no enough funds in the insurance pool. In this case, there is a need to recalculate the payment from the insurance pool as well as the resulting `repayAmount` (line 522). If the resulting `repayAmount` is not recalculated, it returns a wrong amount that will be executed in `repayByInsuranceInternal()` to incorrectly reduce the borrower's debt.

Recommendation Recalculate and return the actual `repayAmount` in all possible cases that need to be handled in `_payInsuranceInternal()`.

Status The issue has been fixed by this commit: `bfd2669`.

3.5 Inconsistency Between Document and Implementation

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [8]
- 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 199 of `CToken::getAccountSnapshot()`. The preceding function summary indicates that this function is supposed to return *"possible error, token balance, borrow balance, exchange rate mantissa."* However, the implementation logic (line 214) indicates there is no error code returned.

```
199     /**
200      * @notice Get a snapshot of the account's balances, and the cached exchange rate
201      * @dev This is used by comptroller to more efficiently perform liquidity checks.
202      * @param account Address of the account to snapshot
203      * @return (possible error, token balance, borrow balance, exchange rate mantissa)
204      */
205     function getAccountSnapshot(address account) external view returns (uint, uint, uint
            , uint) {
206         uint cTokenBalance = accountTokens[account];
207         uint borrowBalance;
208         uint exchangeRateMantissa;
```

```
209         uint validBorrow;
210         MathError mErr;
211
212         (mErr, borrowBalance, validBorrow) = borrowBalanceStoredInternal(account);
213         require(mErr == MathError.NO_ERROR);
214
215         (mErr, exchangeRateMantissa) = exchangeRateStoredInternal();
216         require(mErr == MathError.NO_ERROR);
217
218         return (cTokenBalance, borrowBalance, validBorrow, exchangeRateMantissa);
219     }
```

Listing 3.7: CToken::getAccountSnapshot()

Also, there is another inconsistency at line 291 of `CToken::borrowBalanceStoredInternal()`.

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

Status The issue has been fixed by this commit: `bfd2669`.

3.6 Unintended Insurance Coverage Before Collateral Liquidation

- ID: PVE-006
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `Comptroller`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

As mentioned earlier, the `Wing Finance` protocol is in essence an over-collateralized lending pool that behaves identical to `Compound` and supports a number of normal lending functionalities for supplying and borrowing users, i.e., `mint()/redeem()` and `borrow()/repay()`. With the introduction of an insurance pool, the protocol strengthens the security protection to cover underwater borrows. By design, for an underwater position, the protocol typically liquidates the borrower's collateral before covering the loss from the insurance pool. However, our analysis shows that the current implementation allows for covering the loss first from the insurance pool.

The issue comes from the fact that the coverage from the insurance pool is implemented in a gateway function `liquidateBorrowVerify()`. To elaborate, we show below the implementation. In particular, as shown in line 449, it calls `repayByInsuranceInternal()` to cover the current loss. However, we notice that this gateway function is a public one, which means any one can call it.

In other words, it allows to cover the loss from an underwater position by directly paying from the insurance pool without actually liquidating the user's collateral first.

```
194     /**
195      * @notice Validates liquidateBorrow and reverts on rejection. May emit logs.
196      * @param cTokenBorrowed Asset which was borrowed by the borrower
197      * @param borrower The address of the borrower
198      */
199     function liquidateBorrowVerify(
200         address cTokenBorrowed,
201         address borrower) external returns (uint) {
202         (Error err, uint liquidity,,uint sumCollateral) =
                getHypotheticalAccountLiquidityInternal(borrower, CToken(0), 0, 0);
203         if (err != Error.NO_ERROR) {
204             return uint(err);
205         }

207         if (liquidity > 0) {
208             return uint(Error.LIQUIDITY_NOT_0);
209         }
210         if (sumCollateral > insuranceUsageThreshold) {
211             return uint(Error.SUM_COLLATERAL_BIGGER);
212         }
213         err = repayByInsuranceInternal(cTokenBorrowed, borrower);
214         if (err != Error.NO_ERROR) {
215             return fail(err, FailureInfo.REPAY_BY_INSURANCE);
216         }
217         return uint(Error.NO_ERROR);
218     }
```

Listing 3.8: Comptroller:: liquidateBorrowVerify ()

This is certainly against the intention of introducing the insurance pool and needs to properly guarded to block it.

Recommendation Maintain the intended order of covering the loss by handling the underwater borrows. In particular, the insurance pool should be paid out as the last resort!

Status The issue has been fixed by this commit: `8b1e2c2`.

3.7 Trust Issue of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [3]

Description

In the `Wing Finance` protocol, the privileged owner account plays a critical role in governing and regulating the system-wide operations (e.g., settings of risk parameters). It also has the privilege to regulate or govern the flow of assets for borrowing and lending among the involved components. In the following, we show a representative privileged operation in the `CToken` protocol. This routine essentially allows the pool owner to collect all funds in current pool.

```
254     //Add emergency interface
255     function withdrawAllToken(uint ethBalance) public returns (bool){
256         require(msg.sender == admin, "only admin can invoke this method");
257         if (underlying == address(0)) {
258             doTransferOut(address(admin), ethBalance);
259         } else {
260             EIP20NonStandardInterface token = EIP20NonStandardInterface(underlying);
261             uint balance = token.balanceOf(address(this));
262             doTransferOut(address(admin), balance);
263         }
264         return true;
265     }
```

Listing 3.9: CToken::withdrawAllToken()

Also, if we examine the `IToken::withdrawAllToken()`, which also allows the privileged owner to transfer all funds held in the contract.

```
56      //Add emergency interface
57      function withdrawAllToken() public returns (bool) {
58          require(msg.sender == admin, "only admin can invoke this method");
59          EIP20NonStandardInterface token = EIP20NonStandardInterface(getUnderlying());
60          uint balance = token.balanceOf(address(this));
61          doTransferOut(address(admin), balance);
62          return true;
63      }
```

Listing 3.10: IToken::withdrawAllToken()

We emphasize that the privilege assignment with certain accounts is necessary and required for proper protocol operations. However, it is worrisome if the above two routines are managed by an EOA account. In fact, there is a clear need for a proper governance to regulate and restrict such

operations. The discussion with the team has confirmed that this owner account will be managed by a multi-sig account.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changes to privileged operations may 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 has informed this is part of the design for use in emergence situations.

3.8 Redundant Code Removal

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [4]

Description

The Wing Finance protocol makes good use of a number of reference contracts, such as ERC20, SafeERC20, SafeMath, and Pausable, to facilitate its code implementation and organization. For example, the CToken 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::_acceptAdmin() routine, the statement of verifying of msg.sender == address(0) (line 1225) is unnecessary as it always yields false. The similar redundancy also occurs in the IToken::_acceptAdmin() routine.

```
1218     /**
1219       * @notice Accepts transfer of admin rights. msg.sender must be pendingAdmin
1220       * @dev Admin function for pending admin to accept role and update admin
1221       * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
1222       */
1223     function _acceptAdmin() external returns (uint) {
1224         // Check caller is pendingAdmin and pendingAdmin != address(0)
1225         if (msg.sender != pendingAdmin  msg.sender == address(0)) {
1226             return fail(Error.UNAUTHORIZED, FailureInfo.ACCEPT_ADMIN_PENDING_ADMIN_CHECK
                    );
1227         }
1228
1229         // Save current values for inclusion in log
1230         address oldAdmin = admin;
```

```
1231         address oldPendingAdmin = pendingAdmin;
1232
1233         // Store admin with value pendingAdmin
1234         admin = pendingAdmin;
1235
1236         // Clear the pending value
1237         pendingAdmin = address(0);
1238
1239         emit NewAdmin(oldAdmin, admin);
1240         emit NewPendingAdmin(oldPendingAdmin, pendingAdmin);
1241
1242         return uint(Error.NO_ERROR);
1243     }
```

Listing 3.11: CToken::_acceptAdmin()

More, the following two functions `IToken::transferComp()` and `Comptroller::adminOrInitializing()` are not used and can be safely removed as well.

Recommendation Consider the removal of the redundant code with a simplified, consistent implementation.

Status The issue has been fixed by this commit: `bfd2669`.

3.9 Improved Sanity Checks Of System Parameters

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Comptroller`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [2]

Description

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

```
    function _setLiquidationIncentive(address[] calldata cTokens, uint[] calldata
        incentives) external returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.
                SET_LIQUIDATION_INCENTIVE_OWNER_CHECK);
```

```
        }
        require(cTokens.length == incentives.length, "cToken.length !=
            newLiquidationIncentive.length");
        for (uint i = 0; i < cTokens.length; i ++) {
            // Set liquidation incentive to new incentive
            liquidationIncentiveMantissa[cTokens[i]] = incentives[i];
        }
        return uint(Error.NO_ERROR);
    }

    function _setLockWingFactor(uint newLockWingFactorMantissa) external returns (uint)
        {
        // Check caller is admin
        require(msg.sender == admin, "only admin can set close factor");
        lockWingFactorMantissa = newLockWingFactorMantissa;
        return uint(Error.NO_ERROR);
    }
```

Listing 3.12: Comptroller::_setLiquidationIncentive() and Comptroller::_setBorrowSafeRatio()

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 low percentage as the incentive for the liquidator, hence reducing their gains or hurting their participation 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 in related `setters`, including `_setTransferPaused()` and `_setSeizePaused()`.

Status The issue has been confirmed and the team decides to exercise extra caution in reconfiguring the protocol-wide risk parameters.

3.10 Unintentional WING Collateral Transfer-In

- ID: PVE-010
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `Comptroller`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

In Section 3.6, we have examined a gateway function that undermines the protocol security by paying out possible loss directly from the insurance pool without liquidating the user's collateral. In this section, we examine another gateway function also from the `Comptroller` contract that allows for unintended transfer-in of trusting users' `WING` collateral into the pool.

The issue comes from the unguarded `borrowAllowed()` function. To elaborate, we show below its full implementation. It comes to our attention that this gateway has the logic of transferring in users' `WING` collateral at the end of its function (lines 304 – 309).

```
254     function borrowAllowed(address cToken, address borrower, uint borrowAmount, bool
            useWing) external returns (uint) {
255         // Pausing is a very serious situation - we revert to sound the alarms
256         require(!borrowGuardianPaused[cToken], "borrow is paused");

258         if (!markets[cToken].isListed) {
259             return uint(Error.MARKET_NOT_LISTED);
260         }

262         if (!markets[cToken].accountMembership[borrower]) {
263             // only cTokens may call borrowAllowed if borrower not in market
264             require(msg.sender == cToken, "sender must be cToken");

266             // attempt to add borrower to the market
267             Error err = addToMarketInternal(CToken(msg.sender), borrower);
268             if (err != Error.NO_ERROR) {
269                 return uint(err);
270             }
271             // it should be impossible to break the important invariant
272             assert(markets[cToken].accountMembership[borrower]);
273         }

275         if (oracle.getUnderlyingPrice(CToken(cToken).underlying()) == 0) {
276             return uint(Error.PRICE_ERROR);
277         }

279         (Error err, , uint shortfall,) = getHypotheticalAccountLiquidityInternal(
                borrower, CToken(cToken), 0, borrowAmount);
280         if (err != Error.NO_ERROR) {
```

```
281             return uint(err);
282         }
283         if (shortfall > 0) {
284             return uint(Error.INSUFFICIENT_LIQUIDITY);
285         }

287         // Keep the flywheel moving
288         Exp memory borrowIndex = Exp({mantissa : CToken(cToken).borrowIndex()});
289         updateCompBorrowIndex(cToken, borrowIndex);
290         distributeBorrowerComp(cToken, borrower, borrowIndex, false);
291         BorrowAllowedStruct memory vars;
292         // lock wing
293         vars.tokenPrice = oracle.getUnderlyingPrice(CToken(cToken).underlying());
294         if (vars.tokenPrice == 0) {
295             return uint(Error.PRICE_ERROR);
296         }
297         vars.wingPrice = oracle.getUnderlyingPrice(getCompAddress());
298         if (vars.wingPrice == 0) {
299             return uint(Error.PRICE_ERROR);
300         }
301         if (lockWingFactorMantissa == 0) {
302             return uint(Error.LOCK_WING_FACTOR_MANTISSA);
303         }
304         if (useWing) {
305             vars.token = mul_(Exp({mantissa : vars.tokenPrice}), Exp({mantissa :
                    borrowAmount}));
306             vars.lockAmt = mul_(Exp({mantissa : lockWingFactorMantissa}), vars.token);
307             vars.wingAmt = div_(vars.lockAmt, Exp({mantissa : vars.wingPrice}));
308             transferWingCollateralIn(address(cToken), borrower, vars.wingAmt.mantissa);
309         }
310         return uint(Error.NO_ERROR);
311     }
```

Listing 3.13: Comptroller::borrowAllowed()

This vulnerability is in the same nature as the previously reported issue as elaborated in PVE-006. It is strongly suggested not to piggy-back any business logic in these gateway functions. In other words, these gateway functions are purely for the purpose of validations, not actual business logic.

Recommendation Revise the affected `borrowAllowed()` by avoiding `WING` collateral operations.

Status The issue has been fixed by this commit: `bfd2669`.

4 Conclusion

In this audit, we have analyzed the Wing Finance design and implementation. Inspired from the popular Compound protocol, the system presents a unique, robust offering as an over-collateralized lending pool with the insurance pool support. 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[7] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.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 VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

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

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