SMART CONTRACT AUDIT REPORT

for

Drops NFT

Prepared By: Yiqun Chen

PeckShield

January 17, 2021

Document Properties

Client	Drops
Title	Smart Contract Audit Report
Target	Drops NFT
Version	1.0
Author	Jing Wang
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	January 17, 2021	Jing Wang	Final Release
1.0-rc	December 31, 2021	Jing Wang	Release Candidate

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 design document and related smart contract source code of the Drops NFT protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Drops NFT

Drops NFT is a lending and borrowing protocol with the goal of developing a NFT supported money market. The protocol designs are architected and inspired based on Compound with the extensions of supporting NFT for loans. By adding the permission-less NFT lending pools, Drops NFT enables users to put their NFT down as collateral and receive instant access to a trust-less loan without having to talk to the lender or wait to be approved.

The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Drops NFT Protocol

Item	Description
Name	Drops
Website	https://drops.co/
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 17, 2021

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

supported assets and the oracle itself is not part of this audit. Also, there is an implicit assumption of the project that all NFTs of the same collection are priced the same valued by their floor price.

- https://github.com/Dropsorg/drops-nft-loans-protocol.git (eb8a1e8)

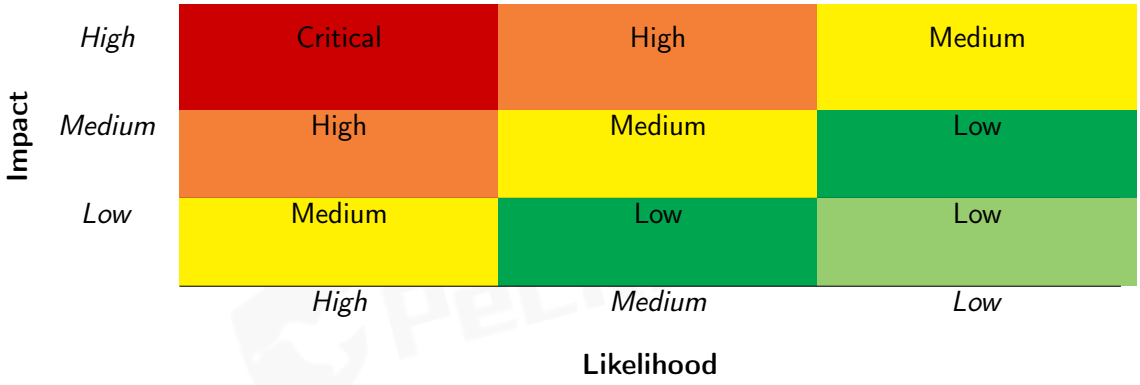
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/Dropsorg/drops-nft-loans-protocol.git (d5fc9c9)

1.2 About PeckShield

PeckShield Inc. [12] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification



1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [11]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

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

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [10], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.3: The Full List of Check Items

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

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

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `Drops NFT` implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	2	■■
Medium	1	■
Low	4	■■■■
Informational	1	■
Total	8	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that 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, 1 medium-severity vulnerability, 4 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Drops NFT Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Improper Handling of Underlying NFT Transfer in CtokenEx::seizeInternal()	Business Logic	Fixed
PVE-002	Informational	Improved Handling of Exchange Rate in CtokenEx	Business Logic	Fixed
PVE-003	High	Uninitialized State Index DoS From Reward Activation	Business Logic	Fixed
PVE-004	Medium	Non ERC20-Compliance Of CToken	Coding Practices	Confirmed
PVE-005	Low	Interface Inconsistency Between CErc20 And CEther	Coding Practice	Confirmed
PVE-006	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Confirmed
PVE-007	Low	Possible Front-Running For Unintended Payment In repayBorrowBehalf()	Time And State	Confirmed
PVE-008	Low	Recommended Explicit tokenIndex Validity Checks	Time And State	Fixed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Improper Handling of Underlying NFT Transfer in CtokenEx::seizeInternal()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `CtokenEx`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [6]

Description

As mentioned earlier, the `Drops NFT` protocol is heavily forked from `Compound` and shares the same architectural design, including the liquidation functionality. To support the `NFT`'s underlying tokens, the `Drops NFT` protocol extends the `Ctoken` contract by revising a number of routines, e.g., `transferTokens()`, `mintFresh()` and `redeemFresh()`. However, there is also a need to revise the implementation of the `CtokenEx::seizeInternal()` routine when transferring collateral `NFT` tokens (from this market) to the liquidator.

```
1128    function seizeInternal(address seizerToken, address liquidator, address borrower,
            uint seizeTokens) internal returns (uint) {
1129        /* Fail if seize not allowed */
1130        uint allowed = comptroller.seizeAllowed(address(this), seizerToken, liquidator,
                borrower, seizeTokens);
1131        if (allowed != 0) {
1132            return failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.
                    LIQUIDATE_SEIZE_COMPTROLLER_REJECTION, allowed);
1133        }
1134
1135        /* Fail if borrower = liquidator */
1136        if (borrower == liquidator) {
1137            return fail(Error.INVALID_ACCOUNT_PAIR, FailureInfo.
                    LIQUIDATE_SEIZE_LIQUIDATOR_IS_BORROWER);
1138        }
1139
```

```
1140        MathError mathErr;
1141        uint borrowerTokensNew;
1142        uint liquidatorTokensNew;
1143
1144        /*
1145         * We calculate the new borrower and liquidator token balances, failing on
                 underflow/overflow:
1146         *  borrowerTokensNew = accountTokens[borrower] - seizeTokens
1147         *  liquidatorTokensNew = accountTokens[liquidator] + seizeTokens
1148         */
1149        (mathErr, borrowerTokensNew) = subUInt(accountTokens[borrower], seizeTokens);
1150        if (mathErr != MathError.NO_ERROR) {
1151            return failOpaque(Error.MATH_ERROR, FailureInfo.
                    LIQUIDATE_SEIZE_BALANCE_DECREMENT_FAILED, uint(mathErr));
1152        }
1153
1154        (mathErr, liquidatorTokensNew) = addUInt(accountTokens[liquidator], seizeTokens)
                ;
1155        if (mathErr != MathError.NO_ERROR) {
1156            return failOpaque(Error.MATH_ERROR, FailureInfo.
                    LIQUIDATE_SEIZE_BALANCE_INCREMENT_FAILED, uint(mathErr));
1157        }
1158
1159        ////////////////////////
1160        // EFFECTS & INTERACTIONS
1161        // (No safe failures beyond this point)
1162
1163        /* We write the previously calculated values into storage */
1164        accountTokens[borrower] = borrowerTokensNew;
1165        accountTokens[liquidator] = liquidatorTokensNew;
1166
1167        /* Emit a Transfer event */
1168        emit Transfer(borrower, liquidator, seizeTokens);
1169
1170        /* We call the defense hook */
1171        comptroller.seizeVerify(address(this), seizerToken, liquidator, borrower,
                seizeTokens);
1172
1173        return uint(Error.NO_ERROR);
1174    }
```

Listing 3.1: `CtokenEx::seizeInternal()`

To elaborate, we show above the related implementation of the `seizeInternal()` routine. This routine calculates the new borrower and liquidator token balances after the liquidation writes the updated values into storage. However, our analysis shows that it only updates the balance of `CToken`, which could not result in a successful transfer for a `NFT-CToken`. In fact, without properly handling of the underlying `NFT` transfer, the user could not redeem any `NFT-CToken` even they have enough balance in `accountTokens[user]`.

Recommendation Properly handle the underlying NFT transfer in CtokenEx::seizeInternal().

Status The issue has been fixed by this commit: d44c29b.

3.2 Improved Handling of Exchange Rate in CtokenEx

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: CtokenEx
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [6]

Description

As mentioned in Section 3.1, the Drops NFT protocol is heavily forked from Compound and shares the same architectural design. The protocol extends the CToken contract to support NFTs and their transfers. However, as the CToken contract is originally designed to support ERC20 tokens, when the CtokenEx contract forks the design from CToken, the exchange rate is not properly handled by the Drops NFT protocol.

```
70      function transferTokens(address spender, address src, address dst, uint tokens)
            internal returns (uint) {
71          /* Fail if transfer not allowed */
72          uint allowed = comptroller.transferAllowed(address(this), src, dst, 1);
73          if (allowed != 0) {
74              return failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.
                    TRANSFER_COMPTROLLER_REJECTION, allowed);
75          }

77          /* Do not allow self-transfers */
78          if (src == dst) {
79              return fail(Error.BAD_INPUT, FailureInfo.TRANSFER_NOT_ALLOWED);
80          }

82          /* Get the allowance, infinite for the account owner */
83          uint startingAllowance = 0;
84          if (spender == src) {
85              startingAllowance = uint(-1);
86          } else {
87              startingAllowance = transferAllowances[src][spender];
88          }

90          /* Do the calculations, checking for {under,over}flow */
91          MathError mathErr;
92          uint allowanceNew;
93          uint srcTokensNew;
94          uint dstTokensNew;
```

```
96          (mathErr, allowanceNew) = subUInt(startingAllowance, 1);
97          if (mathErr != MathError.NO_ERROR) {
98              return fail(Error.MATH_ERROR, FailureInfo.TRANSFER_NOT_ALLOWED);
99          }

101         (mathErr, srcTokensNew) = subUInt(accountTokens[src], 1);
102         if (mathErr != MathError.NO_ERROR) {
103             return fail(Error.MATH_ERROR, FailureInfo.TRANSFER_NOT_ENOUGH);
104         }

106         (mathErr, dstTokensNew) = addUInt(accountTokens[dst], 1);
107         if (mathErr != MathError.NO_ERROR) {
108             return fail(Error.MATH_ERROR, FailureInfo.TRANSFER_TOO_MUCH);
109         }

111         ////////////////////////
112         // EFFECTS & INTERACTIONS
113         // (No safe failures beyond this point)

115         accountTokens[src] = srcTokensNew;
116         accountTokens[dst] = dstTokensNew;

118         /* Eat some of the allowance (if necessary) */
119         if (startingAllowance != uint(-1)) {
120             transferAllowances[src][spender] = allowanceNew;
121         }

123         doTransfer(src, dst, tokens);

125         /* We emit a Transfer event */
126         emit Transfer(src, dst, 1);

128         comptroller.transferVerify(address(this), src, dst, 1);

130         return uint(Error.NO_ERROR);
131     }
```

Listing 3.2: `CtokenEx::transferTokens()`

To elaborate, we show above the `transferTokens()` routine. We notice the routine is calling the `doTransfer()` routine to transfer the `NFT`'s underlying token. It is making an implicit assumption that the amount of underlying tokens to be transferred by the `transferTokens()` routine is 1. However, the amount of `CTokens` to be transferred in the `transferTokens()` routine is also hard coded as 1 (lines 72, 96, 101, and 106). Our analysis shows that the conversion from underlying token amounts to `CToken` amounts should remain the constant 1. In this case, we suggest to clean up the current logic to remove the conversion or make the $1 : 1$ conversion explicit!

Note another routine `redeemFreshCToken()` shares the same issue. Also, the `redeemFreshUnderlying()` routine is redundant with the `redeemFreshCToken()` routine because both of them are transferring

1 NFT underlying token with a specific tokenId from the contract to the user.

Recommendation Correct the above routine to properly handle the exchange rates in CtokenEx.

Status The issue has been fixed by this commit: 62a9b01.

3.3 Uninitialized State Index DoS From Reward Activation

- ID: PVE-003
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: ComptrollerG1
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [6]

Description

The Drops NFT protocol provides incentive mechanisms that reward the protocol users. Specifically, the reward mechanism follows the same approach as the COMP reward in Compound. Our analysis on the related COMP reward in Drops NFT shows the current logic needs to be improved.

To elaborate, we show below the initial logic of setCompSpeedInternal() that kicks off the actual minting of protocol tokens. It comes to our attention that the initial supply-side index is configured on the conditions of CompSupplyState[address(cToken)].index == 0 and CompSupplyState[address(cToken)].block == 0 (line 1076). However, for an already listed market with a current speed of 0, the first condition is indeed met while the second condition does not! The reason is that both supply-side state and borrow-side state have the associated block information updated, which is diligently performed via other helper pairs updateCompSupplyIndex()/updateCompBorrowIndex(). As a result, the setCompSpeedInternal() logic does not properly set up the default supply-side index and the default borrow-side index.

```
1064    function setCompSpeedInternal(CToken cToken, uint compSpeed) internal {
1065        uint currentCompSpeed = compSpeeds[address(cToken)];
1066        if (currentCompSpeed != 0) {
1067            // note that COMP speed could be set to 0 to halt liquidity rewards for a
                    market
1068            Exp memory borrowIndex = Exp({mantissa: cToken.borrowIndex()});
1069            updateCompSupplyIndex(address(cToken));
1070            updateCompBorrowIndex(address(cToken), borrowIndex);
1071        } else if (compSpeed != 0) {
1072            // Add the COMP market
1073            Market storage market = markets[address(cToken)];
1074            require(market.isListed == true, "comp market is not listed");

1076            if (compSupplyState[address(cToken)].index == 0 && compSupplyState[address(
                    cToken)].block == 0) {
```

```
1077                compSupplyState[address(cToken)] = CompMarketState({
1078                    index: compInitialIndex,
1079                    block: safe32(getBlockNumber(), "block number exceeds 32 bits")
1080                });
1081            }

1083            if (compBorrowState[address(cToken)].index == 0 && compBorrowState[address(
                    cToken)].block == 0) {
1084                compBorrowState[address(cToken)] = CompMarketState({
1085                    index: compInitialIndex,
1086                    block: safe32(getBlockNumber(), "block number exceeds 32 bits")
1087                });
1088            }
1089        }

1091        if (currentCompSpeed != compSpeed) {
1092            compSpeeds[address(cToken)] = compSpeed;
1093            emit CompSpeedUpdated(cToken, compSpeed);
1094        }
1095    }
```

Listing 3.3: `ComptrollerG1::setCompSpeedInternal()`

```
1101    function updateCompSupplyIndex(address cToken) internal {
1102        CompMarketState storage supplyState = compSupplyState[cToken];
1103        uint supplySpeed = compSpeeds[cToken];
1104        uint blockNumber = getBlockNumber();
1105        uint deltaBlocks = sub_(blockNumber, uint(supplyState.block));
1106        if (deltaBlocks > 0 && supplySpeed > 0) {
1107            uint supplyTokens = CToken(cToken).totalSupply();
1108            uint compAccrued = mul_(deltaBlocks, supplySpeed);
1109            Double memory ratio = supplyTokens > 0 ? fraction(compAccrued, supplyTokens)
                    : Double({mantissa: 0});
1110            Double memory index = add_(Double({mantissa: supplyState.index}), ratio);
1111            compSupplyState[cToken] = CompMarketState({
1112                index: safe224(index.mantissa, "new index exceeds 224 bits"),
1113                block: safe32(blockNumber, "block number exceeds 32 bits")
1114            });
1115        } else if (deltaBlocks > 0) {
1116            supplyState.block = safe32(blockNumber, "block number exceeds 32 bits");
1117        }
1118    }
```

Listing 3.4: `ComptrollerG1::updateCompSupplyIndex()`

When the reward speed is configured, since the supply-side and borrow-side state indexes are not initialized, any normal functionality such as `mint()` will be immediately reverted! This revert occurs inside the `distributeSupplierComp()/distributeBorrowerComp()` functions. Using the `distributeSupplierComp()` function as an example, the revert is caused from the arithmetic operation `sub_(supplyIndex, supplierIndex)` (line 1158). Since the `supplyIndex` is not properly initialized, it will be updated to a smaller number from an earlier invocation of `updateCompSupplyIndex()` (lines 1111-1114). However,

when the `distributeSupplierComp()` function is invoked, the `supplierIndex` is reset with `CompInitialIndex` (line 1155), which unfortunately reverts the arithmetic operation `sub_(supplyIndex, supplierIndex)`!

```
1148    function distributeSupplierComp(address cToken, address supplier) internal {
1149        CompMarketState storage supplyState = compSupplyState[cToken];
1150        Double memory supplyIndex = Double({mantissa: supplyState.index});
1151        Double memory supplierIndex = Double({mantissa: compSupplierIndex[cToken][
                supplier]});
1152        compSupplierIndex[cToken][supplier] = supplyIndex.mantissa;

1154        if (supplierIndex.mantissa == 0 && supplyIndex.mantissa > 0) {
1155            supplierIndex.mantissa = compInitialIndex;
1156        }

1158        Double memory deltaIndex = sub_(supplyIndex, supplierIndex);
1159        uint supplierTokens = CToken(cToken).balanceOf(supplier);
1160        uint supplierDelta = mul_(supplierTokens, deltaIndex);
1161        uint supplierAccrued = add_(compAccrued[supplier], supplierDelta);
1162        compAccrued[supplier] = supplierAccrued;
1163        emit DistributedSupplierComp(CToken(cToken), supplier, supplierDelta,
                supplyIndex.mantissa);
1164    }
```

Listing 3.5: `ComptrollerG1::distributeSupplierComp()`

Recommendation Properly initialize the reward state indexes in the above affected `setCompSpeedInternal()` function. An example revision is shown as follows:

```
1064    function setCompSpeedInternal(cToken cToken, uint CompSpeed) internal {
1065        uint currentCompSpeed = CompSpeeds[address(cToken)];
1066        if (currentCompSpeed != 0) {
1067            // note that Comp speed could be set to 0 to halt liquidity rewards for a
                    market
1068            Exp memory borrowIndex = Exp({mantissa: cToken.borrowIndex()});
1069            updateCompSupplyIndex(address(cToken));
1070            updateCompBorrowIndex(address(cToken), borrowIndex);
1071        } else if (CompSpeed != 0) {
1072            // Add the Comp market
1073            Market storage market = markets[address(cToken)];
1074            require(market.isListed == true, "Comp market is not listed");

1076            if (CompSupplyState[address(cToken)].index == 0) {
1077                CompSupplyState[address(cToken)].index = CompInitialIndex;
1078            }
1079            CompSupplyState[address(cToken)].block = safe32(getBlockNumber());

1081            if (CompBorrowState[address(cToken)].index == 0) {
1082                CompBorrowState[address(cToken)].index = CompInitialIndex;
1083            }
1084            CompBorrowState[address(cToken)].block = safe32(getBlockNumber());
1085        }
```

```
1087         if (currentCompSpeed != CompSpeed) {
1088             CompSpeeds[address(cToken)] = CompSpeed;
1089             emit CompSpeedUpdated(cToken, CompSpeed);
1090         }
1091     }
```

Listing 3.6: `Comptroller::setCompSpeedInternal()`

Status The issue has been fixed by this commit: `c2c59a3`.

3.4 Non ERC20-Compliance Of CToken

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `CToken`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [3]

Description

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	✓

Each asset supported by the Drops NFT 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 CToken as collateral. There are currently two types of CToken: CErc20 and CEther. In the following, we examine the ERC20 compliance of these cTokens.

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.

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.

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.

3.5 Interface Inconsistency Between CErc20 And CEther

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [2]

Description

As mentioned in Section 3.4, each asset supported by the Drops NFT 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 Drops NFT 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 ETH, 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().

```
80      /**
81       * @notice Sender repays their own borrow
82       * @param repayAmount The amount to repay
83       * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
84       */
85      function repayBorrow(uint repayAmount) external override returns (uint) {
86          (uint err,) = repayBorrowInternal(repayAmount);
87          return err;
```

```
88      }
```

Listing 3.7: CErc20::repayBorrow()

```
80      /**
81       * @notice Sender repays their own borrow
82       * @dev Reverts upon any failure
83       */
84      function repayBorrow() external payable {
85          (uint err,) = repayBorrowInternal(msg.value);
86          requireNoError(err, "repayBorrow failed");
87      }
```

Listing 3.8: CEther::repayBorrow()

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.

3.6 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Time and State [9]
- CWE subcategory: CWE-663 [5]

Description

A common coding best practice in Solidity is the adherence of checks-effects-interactions principle. This principle is effective in mitigating a serious attack vector known as re-entrancy. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the DAO [14] exploit, and the recent Uniswap/Lendf.Me hack [13].

We notice there are occasions where the checks-effects-interactions principle is violated. Using the CToken as an example, the repayBorrowFresh() function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above re-entrancy. For example, the interaction with the external

contract (line 898) start before effecting the update on internal states (lines 912 – 914), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
857   function repayBorrowFresh(address payer, address borrower, uint repayAmount) internal
          returns (uint, uint) {
858       /* Fail if repayBorrow not allowed */
859       uint allowed = comptroller.repayBorrowAllowed(address(this), payer, borrower,
              repayAmount);
860       if (allowed != 0) {
861           return (failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.
                  REPAY_BORROW_COMPTROLLER_REJECTION, allowed), 0);
862       }
863
864       /* Verify market's block number equals current block number */
865       if (accrualBlockNumber != getBlockNumber()) {
866           return (fail(Error.MARKET_NOT_FRESH, FailureInfo.
                  REPAY_BORROW_FRESHNESS_CHECK), 0);
867       }
868
869       RepayBorrowLocalVars memory vars;
870
871       /* We remember the original borrowerIndex for verification purposes */
872       vars.borrowerIndex = accountBorrows[borrower].interestIndex;
873
874       /* We fetch the amount the borrower owes, with accumulated interest */
875       (vars.mathErr, vars.accountBorrows) = borrowBalanceStoredInternal(borrower);
876       if (vars.mathErr != MathError.NO_ERROR) {
877           return (failOpaque(Error.MATH_ERROR, FailureInfo.
                  REPAY_BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED, uint(vars.mathErr))
                  , 0);
878       }
879
880       /* If repayAmount == -1, repayAmount = accountBorrows */
881       if (repayAmount == uint(-1)) {
882           vars.repayAmount = vars.accountBorrows;
883       } else {
884           vars.repayAmount = repayAmount;
885       }
886
887       ////////////////////////
888       // EFFECTS & INTERACTIONS
889       // (No safe failures beyond this point)
890
891       /*
892        * We call doTransferIn for the payer and the repayAmount
893        *  Note: The cToken must handle variations between ERC-20 and ETH underlying.
894        *  On success, the cToken holds an additional repayAmount of cash.
895        *  doTransferIn reverts if anything goes wrong, since we can't be sure if side
                effects occurred.
896        *   it returns the amount actually transferred, in case of a fee.
897        */
```

```
898         vars.actualRepayAmount = doTransferIn(payer, vars.repayAmount);
899
900         /*
901          * We calculate the new borrower and total borrow balances, failing on underflow
                :
902          *  accountBorrowsNew = accountBorrows - actualRepayAmount
903          *  totalBorrowsNew = totalBorrows - actualRepayAmount
904          */
905         (vars.mathErr, vars.accountBorrowsNew) = subUInt(vars.accountBorrows, vars.
                actualRepayAmount);
906         require(vars.mathErr == MathError.NO_ERROR, "
                REPAY_BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED");
907
908         (vars.mathErr, vars.totalBorrowsNew) = subUInt(totalBorrows, vars.
                actualRepayAmount);
909         require(vars.mathErr == MathError.NO_ERROR, "
                REPAY_BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED");
910
911         /* We write the previously calculated values into storage */
912         accountBorrows[borrower].principal = vars.accountBorrowsNew;
913         accountBorrows[borrower].interestIndex = borrowIndex;
914         totalBorrows = vars.totalBorrowsNew;
915
916         /* We emit a RepayBorrow event */
917         emit RepayBorrow(payer, borrower, vars.actualRepayAmount, vars.accountBorrowsNew
                , vars.totalBorrowsNew);
918
919         /* We call the defense hook */
920         comptroller.repayBorrowVerify(address(this), payer, borrower, vars.
                actualRepayAmount, vars.borrowerIndex);
921
922         return (uint(Error.NO_ERROR), vars.actualRepayAmount);
923     }
```

Listing 3.9: `cToken::repayBorrowFresh()`

While the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`, it is important to take precautions to thwart possible `re-entrancy`. The similar issue is also present in other functions, including `mintFresh()` and `repayBorrowFresh()` in other contracts, and the adherence of the `checks-effects-interactions` best practice is strongly recommended. We highlight that the very same issue has been exploited in a recent `Cream` incident [1] and therefore deserves special attention.

From another perspective, the current mitigation in applying money-market-level reentrancy protection can be strengthened by elevating the reentrancy protection at the `Comptroller`-level. In addition, each individual function can be self-strengthened by following the `checks-effects-interactions` principle

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-`

interactions principle and utilizing the necessary nonReentrant modifier to block possible re-entrancy. Also consider strengthening the reentrancy protection at the protocol-level instead of at the current money-market granularity.

Status The issue has been confirmed.

3.7 Possible Front-Running For Unintended Payment In repayBorrowBehalf()

- ID: PVE-007
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: CToken
- Category: Time and State [9]
- CWE subcategory: CWE-663 [5]

Description

As mentioned earlier, the Drops NFT 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 Drops NFT 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.

```
857     function repayBorrowFresh(address payer, address borrower, uint repayAmount)
            internal returns (uint, uint) {
858         /* Fail if repayBorrow not allowed */
859         uint allowed = comptroller.repayBorrowAllowed(address(this), payer, borrower,
                repayAmount);
860         if (allowed != 0) {
861             return (failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.
                    REPAY_BORROW_COMPTROLLER_REJECTION, allowed), 0);
862         }

864         /* Verify market's block number equals current block number */
865         if (accrualBlockNumber != getBlockNumber()) {
866             return (fail(Error.MARKET_NOT_FRESH, FailureInfo.
                    REPAY_BORROW_FRESHNESS_CHECK), 0);
867         }
```

```
869         RepayBorrowLocalVars memory vars;

871         /* We remember the original borrowerIndex for verification purposes */
872         vars.borrowerIndex = accountBorrows[borrower].interestIndex;

874         /* We fetch the amount the borrower owes, with accumulated interest */
875         (vars.mathErr, vars.accountBorrows) = borrowBalanceStoredInternal(borrower);
876         if (vars.mathErr != MathError.NO_ERROR) {
877             return (failOpaque(Error.MATH_ERROR, FailureInfo.
                    REPAY_BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED, uint(vars.mathErr))
                    , 0);
878         }

880         /* If repayAmount == -1, repayAmount = accountBorrows */
881         if (repayAmount == uint(-1)) {
882             vars.repayAmount = vars.accountBorrows;
883         } else {
884             vars.repayAmount = repayAmount;
885         }

887         ////////////////////////
888         // EFFECTS & INTERACTIONS
889         // (No safe failures beyond this point)

891         /*
892          * We call doTransferIn for the payer and the repayAmount
893          *  Note: The cToken must handle variations between ERC-20 and ETH underlying.
894          *  On success, the cToken holds an additional repayAmount of cash.
895          *  doTransferIn reverts if anything goes wrong, since we can't be sure if side
                effects occurred.
896          *   it returns the amount actually transferred, in case of a fee.
897          */
898         vars.actualRepayAmount = doTransferIn(payer, vars.repayAmount);

900         /*
901          * We calculate the new borrower and total borrow balances, failing on underflow
                :
902          *  accountBorrowsNew = accountBorrows - actualRepayAmount
903          *  totalBorrowsNew = totalBorrows - actualRepayAmount
904          */
905         (vars.mathErr, vars.accountBorrowsNew) = subUInt(vars.accountBorrows, vars.
                actualRepayAmount);
906         require(vars.mathErr == MathError.NO_ERROR, "
                REPAY_BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED");

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

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

```
912         accountBorrows[borrower].principal = vars.accountBorrowsNew;
913         accountBorrows[borrower].interestIndex = borrowIndex;
914         totalBorrows = vars.totalBorrowsNew;

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

919         /* We call the defense hook */
920         comptroller.repayBorrowVerify(address(this), payer, borrower, vars.
                actualRepayAmount, vars.borrowerIndex);

922         return (uint(Error.NO_ERROR), vars.actualRepayAmount);
923     }
```

Listing 3.10: `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.

3.8 Recommended Explicit tokenIndex Validity Checks

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CErc721`
- Category: Coding Practices [7]
- CWE subcategory: CWE-628 [4]

Description

In the `CErc721` contract, there are several helper routines which are used to transfer the `NFT` underlying tokens. Note all added `NFT` tokens are maintained in the `userTokens` data structure. To elaborate, we show below the related code snippet.

```
279     /**
280      * @dev User deposit tokens map
281      */
```

```
282     mapping (address => uint256[]) public userTokens;
```

Listing 3.11: The userTokens Data Structure in CErc721Storage

When there is a need to transfer a NFT token from one user to another user (to transfer a NFT token from the user to the money market, or to transfer a NFT token from the money market to the user), there is a constant need to perform sanity checks on the tokenIndex validity. The current implementation simply relies on the implicit, compiler-generated bound-checks of arrays to ensure the token index stays within the array range [0, userTokens[user].length - 1]. However, considering the importance of validating given tokenIndex and their numerous occasions, a better alternative is to make explicit the sanity checks by introducing a new modifier, say validateTokenIndex. This new modifier essentially ensures the given tokenIndex indeed points to a valid, NFT, and additionally give semantically meaningful information when it is not!

```
237     function doTransfer(address from, address to, uint tokenIndex) internal override {
238         // doTransferOut
239         uint tokenId = userTokens[from][tokenIndex];
240         uint newBalance = userTokens[from].length - 1;
241         userTokens[from][tokenIndex] = userTokens[from][newBalance];
242         userTokens[from].pop();
243
244         // doTransferIn
245         userTokens[to].push(tokenId);
246     }
247 }
```

Listing 3.12: CErc721::doTransfer()

We highlight that there are a number of functions that can be benefited from the new tokenIndex -validating modifier, including doTransferIn(), doTransferOut() and doTransfer().

Recommendation Apply necessary sanity checks to ensure the given tokenIndex is legitimate. Accordingly, a new modifier validateTokenIndex can be developed and appended to each function in the above list.

```
    modifier validateTokenIndex(address to, uint tokenIndex) {
        require(tokenIndex < userTokens[to].length, "invalid token index");
        _;
    }
```

Listing 3.13: The New validateTokenIndex() Modifier

Status The issue has been fixed by this commit: d56ea54.

4 | Conclusion

In this audit, we have analyzed the Drops NFT protocol design and implementation. The protocol is designed to be an money market that is inspired from Compound with the extensions of supporting NFT for loans. During the audit, we notice that the current code base is well organized.

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

References

[1] Aislinn Keely. Cream Finance Exploited in $18.8 million Flash Loan Attack. https://www.theblockcrypto.com/linked/116055/creamfinance-exploited-in-18-8-million-flash-loan-attack.

[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-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.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: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

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

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

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

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

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

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

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