SMART CONTRACT AUDIT REPORT

for

Chai Money Market

Prepared By: Patrick Lou

PeckShield

March 14, 2022

Document Properties

Client	Chai Money Market
Title	Smart Contract Audit Report
Target	Chai Money Market
Version	1.0
Author	Patrick Lou
Auditors	Patrick Lou, Yiqun Chen, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 14, 2022	Patrick Lou	Final Release
1.0-rc	February 24, 2022	Patrick Lou	Release Candidate

Contact

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

Name	Patrick Lou
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 Chai Money Market 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 Chai Money Market

The Chai Money Market is a lending and borrowing platform, which is architected and inspired based on Compound with customized changes. Through the mToken contracts as decentralized non-custodial money markets, users may supply capital (ERC20-compliant or native tokens) to receive mTokens or borrow assets from the protocol (while holding other assets as collateral). The associated mToken contracts track these balances and algorithmically set interest rates for borrowers. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Mojito Protocol

Item	Description
Name	Chai Money Market
Website	https://chai.xyz/
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 14, 2022

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

- https://github.com/ChaiMoneyMarket/mojito-contracts (a63d815)

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

- https://github.com/ChaiMoneyMarket/mojito-contracts (f068be9)

1.2 About PeckShield

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

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

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

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

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

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

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

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 1.4: 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 `Chai Money Market` protocol 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	1	■
Medium	2	■ ■
Low	6	■ ■ ■ ■ ■ ■
Informational	0	
Undetermined	1	■
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 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 1 high-severity vulnerabilities, 2 medium-severity vulnerability, 6 low-severity vulnerabilities, and 1 undetermined issue.

Table 2.1: Key Chai Money Market Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Uninitialized State Index DoS From Reward Activation	Business Logic	Fixed
PVE-002	Medium	Non ERC20-Compliance Of MToken	Coding Practices	Fixed
PVE-003	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Fixed
PVE-004	Low	Interface Inconsistency Between MErc20 And MEther	Coding Practices	Confirmed
PVE-005	Low	Duplicate Pool Detection and Prevention	Business Logic	Fixed
PVE-006	Low	Timely massUpdatePools During Pool Weight Changes	Business Logic	Fixed
PVE-007	Undetermined	Staking Incompatibility With Deflationary Tokens	Business Logic	Confirmed
PVE-008	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-009	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-010	Low	Proper Staking Token Initialization in MultiFeeDistribution	Business Logic	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 Uninitialized State Index DoS From Reward Activation

- ID: PVE-001
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Comptroller`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

The `Chai Money Market` protocol is in essence a lending market 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 the rewarding logic of the protocol token, i.e., `Mojito (MOJI)`.

To elaborate, we show below the initial logic of `setRewardSpeedInternal()` 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 `rewardSupplyState[address(mToken)].index == 0` and `rewardSupplyState[address(mToken)].block == 0` (lines 1491-1492). 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 `updateRewardSupplyIndex()/updateRewardBorrowIndex()`. As a result, the `setRewardSpeedInternal()` logic does not properly set up the default supply-side index and the default borrow-side index.

```
1476 function setRewardSpeedInternal(MToken mToken, uint256 rewardSpeed)
1477     internal
1478 {
1479     uint256 currentRewardSpeed = rewardSpeeds[address(mToken)];
1480     if (currentRewardSpeed != 0) {
1481         // note that REWARD speed could be set to 0 to halt liquidity rewards for a
             market
1482         Exp memory borrowIndex = Exp({mantissa: mToken.borrowIndex()});
```

```
        updateRewardSupplyIndex(address(mToken));
        updateRewardBorrowIndex(address(mToken), borrowIndex);
    } else if (rewardSpeed != 0) {
        // Add the REWARD market
        Market storage market = markets[address(mToken)];
        require(market.isListed == true, "market is not listed");

        if (
            rewardSupplyState[address(mToken)].index == 0 &&
            rewardSupplyState[address(mToken)].block == 0
        ) {
            rewardSupplyState[address(mToken)] = RewardMarketState({
                index: rewardInitialIndex,
                block: safe32(
                    getBlockNumber(),
                    "block number exceeds 32 bits"
                )
            });
        }

        if (
            rewardBorrowState[address(mToken)].index == 0 &&
            rewardBorrowState[address(mToken)].block == 0
        ) {
            rewardBorrowState[address(mToken)] = RewardMarketState({
                index: rewardInitialIndex,
                block: safe32(
                    getBlockNumber(),
                    "block number exceeds 32 bits"
                )
            });
        }
    }

    if (currentRewardSpeed != rewardSpeed) {
        rewardSpeeds[address(mToken)] = rewardSpeed;
        emit RewardSpeedUpdated(mToken, rewardSpeed);
    }
}
```

Listing 3.1: `Comptroller::setRewardSpeedInternal()`

```
 function updateRewardSupplyIndex(address mToken) internal {
   RewardMarketState storage supplyState = rewardSupplyState[mToken];
   uint256 supplySpeed = rewardSpeeds[mToken];
   uint256 blockNumber = getBlockNumber();
   uint256 deltaBlocks = sub_(blockNumber, uint256(supplyState.block));
   if (deltaBlocks > 0 && supplySpeed > 0) {
       uint256 supplyTokens = MToken(mToken).totalSupply();
       uint256 rewardAccrued = mul_(deltaBlocks, supplySpeed);
       Double memory ratio = supplyTokens > 0
           ? fraction(rewardAccrued, supplyTokens)
           : Double({mantissa: 0});
```

```
1538         Double memory index = add_(
1539             Double({mantissa: supplyState.index}),
1540             ratio
1541         );
1542         rewardSupplyState[mToken] = RewardMarketState({
1543             index: safe224(index.mantissa, "new index exceeds 224 bits"),
1544             block: safe32(blockNumber, "block number exceeds 32 bits")
1545         });
1546     } else if (deltaBlocks > 0) {
1547         supplyState.block = safe32(
1548             blockNumber,
1549             "block number exceeds 32 bits"
1550         );
1551     }
1552 }
```

Listing 3.2: `Comptroller::updateRewardSupplyIndex()`

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 `distributeSupplierReward()/distributeBorrowerReward()` functions. Using the `distributeSupplierReward()` function as an example, the revert is caused from the arithmetic operation `sub_(supplyIndex, supplierIndex)` (line 1610). Since the `supplyIndex` is not properly initialized, it will be updated to a smaller number from an earlier invocation of `updateRewardSupplyIndex()` (lines 1542-1545). However, when the `distributeSupplierReward()` function is invoked, the `supplierIndex` is reset with `rewardInitialIndex` (line 1607), which unfortunately reverts the arithmetic operation `sub_(supplyIndex, supplierIndex)`!

```
1596   function distributeSupplierReward(address mToken, address supplier)
1597   internal
1598 {
1599   RewardMarketState storage supplyState = rewardSupplyState[mToken];
1600   Double memory supplyIndex = Double({mantissa: supplyState.index});
1601   Double memory supplierIndex = Double({
1602       mantissa: rewardSupplierIndex[mToken][supplier]
1603   });
1604   rewardSupplierIndex[mToken][supplier] = supplyIndex.mantissa;

1606   if (supplierIndex.mantissa == 0 && supplyIndex.mantissa > 0) {
1607       supplierIndex.mantissa = rewardInitialIndex;
1608   }

1610   Double memory deltaIndex = sub_(supplyIndex, supplierIndex);
1611   uint256 supplieMTokens = MToken(mToken).balanceOf(supplier);
1612   uint256 supplierDelta = mul_(supplieMTokens, deltaIndex);
1613   uint256 supplierAccrued = add_(rewardAccrued[supplier], supplierDelta);
1614   rewardAccrued[supplier] = supplierAccrued;
1615   emit DistributedSupplierReward(
1616       MToken(mToken),
1617       supplier,
```

```
        supplierDelta,
        supplyIndex.mantissa
    );
}
```

Listing 3.3: `Comptroller::distributeSupplierReward()`

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

```
    function setRewardSpeedInternal(MToken mToken, uint256 rewardSpeed)
    internal
{
    uint256 currentRewardSpeed = rewardSpeeds[address(mToken)];
    if (currentRewardSpeed != 0) {
        // note that REWARD speed could be set to 0 to halt liquidity rewards for a market
        Exp memory borrowIndex = Exp({mantissa: mToken.borrowIndex()});
        updateRewardSupplyIndex(address(mToken));
        updateRewardBorrowIndex(address(mToken), borrowIndex);
    } else if (rewardSpeed != 0) {
        // Add the REWARD market
        Market storage market = markets[address(mToken)];
        require(market.isListed == true, "market is not listed");

        if (rewardSupplyState[address(mToken)].index == 0 ) {
            rewardSupplyState[address(mToken)] = RewardMarketState({
                index: rewardInitialIndex,
                block: safe32(
                    getBlockNumber(),
                    "block number exceeds 32 bits"
                )
            });
        }

        if (rewardBorrowState[address(mToken)].index == 0 ) {
            rewardBorrowState[address(mToken)] = RewardMarketState({
                index: rewardInitialIndex,
                block: safe32(
                    getBlockNumber(),
                    "block number exceeds 32 bits"
                )
            });
        }
    }
```

Listing 3.4: `Comptroller::setRewardSpeedInternal()`

Status The issue has been fixed by this commit: `768f3d8`.

3.2 Non ERC20-Compliance Of MToken

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: MToken
- Category: Coding Practices [9]
- 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 `Chai Money Market` is integrated through a so-called `MToken` contract, which is an ERC20 compliant representation of balances supplied to the protocol. By minting `MTokens` , users can earn interest through the `MToken`'s exchange rate, which increases in value relative to the underlying asset, and further gain the ability to use `MTokens` as collateral.

In the following, we examine the ERC20 compliance of these `MTokens`. Note 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 `MToken` 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. In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777/ERC2222), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

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

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

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

Status This issue has been confirmed. Considering that this is part of the original `Compound` code base, the team decides to leave it as is to keep it consistent with `Compound`'s version so other applications can use `MToken` contracts in a similar approach.

3.3 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Time and State [11]
- CWE subcategory: CWE-663 [6]

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` [16] exploit, and the recent `Uniswap/Lendf.Me` hack [15].

We notice there are occasions where the `checks-effects-interactions` principle is violated. Using the `MToken` as an example, the `borrowFresh()` 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 1142) start before effecting the update on internal states (lines 1145-1147), 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.

```
1049 function borrowFresh(address payable borrower, uint256 borrowAmount)
1050 internal
1051 returns (uint256)
1052 {
1053 /* Fail if borrow not allowed */
1054 uint256 allowed = comptroller.borrowAllowed(
1055     address(this),
1056     borrower,
1057     borrowAmount
1058 );
1059 if (allowed != 0) {
1060     return
1061         failOpaque(
1062             Error.COMPTROLLER_REJECTION,
1063             FailureInfo.BORROW_COMPTROLLER_REJECTION,
1064             allowed
1065         );
1066 }
1067
1068 /* Verify market's block number equals current block number */
1069 if (accrualBlockNumber != getBlockNumber()) {
1070     return
1071         fail(
1072             Error.MARKET_NOT_FRESH,
1073             FailureInfo.BORROW_FRESHNESS_CHECK
1074         );
1075 }
1076
1077 /* Fail gracefully if protocol has insufficient underlying cash */
1078 if (getCashPrior() < borrowAmount) {
1079     return
1080         fail(
1081             Error.TOKEN_INSUFFICIENT_CASH,
1082             FailureInfo.BORROW_CASH_NOT_AVAILABLE
1083         );
1084 }
1085
1086 BorrowLocalVars memory vars;
1087
```

```
/*
 * We calculate the new borrower and total borrow balances, failing on overflow:
 *  accountBorrowsNew = accountBorrows + borrowAmount
 *  totalBorrowsNew = totalBorrows + borrowAmount
 */
(vars.mathErr, vars.accountBorrows) = borrowBalanceStoredInternal(
    borrower
);
if (vars.mathErr != MathError.NO_ERROR) {
    return
        failOpaque(
            Error.MATH_ERROR,
            FailureInfo.BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED,
            uint256(vars.mathErr)
        );
}

(vars.mathErr, vars.accountBorrowsNew) = addUInt(
    vars.accountBorrows,
    borrowAmount
);
if (vars.mathErr != MathError.NO_ERROR) {
    return
        failOpaque(
            Error.MATH_ERROR,
            FailureInfo
                .BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED,
            uint256(vars.mathErr)
        );
}

(vars.mathErr, vars.totalBorrowsNew) = addUInt(
    totalBorrows,
    borrowAmount
);
if (vars.mathErr != MathError.NO_ERROR) {
    return
        failOpaque(
            Error.MATH_ERROR,
            FailureInfo.BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED,
            uint256(vars.mathErr)
        );
}

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

/*
 * We invoke doTransferOut for the borrower and the borrowAmount.
 *  Note: The MToken must handle variations between ERC-20 and ETH underlying.
 *  On success, the MToken borrowAmount less of cash.
```

```
     *  doTransferOut reverts if anything goes wrong, since we can't be sure if side effects occurred.
     */
    doTransferOut(borrower, borrowAmount);

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

    /* We emit a Borrow event */
    emit Borrow(
        borrower,
        borrowAmount,
        vars.accountBorrowsNew,
        vars.totalBorrowsNew
    );

    /* We call the defense hook */
    // unused function
    // comptroller.borrowVerify(address(this), borrower, borrowAmount);

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

Listing 3.5: `MToken::borrowFresh()`

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 `redeemFresh()` function, 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.

Note the contract `MojitoChef` shares the same issue, especially in its `deposit()` and `emergencyWithdraw()` functions.

```
    // Withdraw without caring about rewards. EMERGENCY ONLY.
    function emergencyWithdraw(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        pool.lpToken.safeTransfer(address(msg.sender), user.amount);
        emit EmergencyWithdraw(msg.sender, _pid, user.amount);
        user.amount = 0;
        user.rewardDebt = 0;
```

```
    }
```

Listing 3.6: MojitoChef::emergencyWithdraw()

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 This issue has been confirmed. And the team decides to fix it by following checks-effects-interactions principle. The issue has been fixed by this commit: 415162d.

3.4 Interface Inconsistency Between MErc20 And MEther

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [9]
- CWE subcategory: CWE-1041 [2]

Description

As mentioned in Section 3.2, each asset supported by the Chai Money Market is integrated through a so-called MToken contract, which is an ERC20 compliant representation of balances supplied to the protocol. And MTokens are the primary means of interacting with the Mojito protocol when a user wants to mint(), redeem(), borrow(), repay(), liquidate(), or transfer(). Moreover, there are currently two types of MTokens: MErc20 and MEther. 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 MErc20 type returns an error code while the MEther type simply reverts upon any failure. The similar inconsistency is also present in other routines, including repayBorrowBehalf(), mint(), and liquidateBorrow().

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

Listing 3.7: MErc20::repayBorrow()

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

Listing 3.8: `MEther::repayBorrow()`

Recommendation Ensure the consistency between these two types: `MErc20` and `MEther`.

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

3.5 Duplicate Pool Detection and Prevention

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Staking`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

The `Chai Money Market` provides incentive mechanisms that reward the staking of supported assets with certain reward tokens. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. Each pool has its `allocPoint*100%/totalAllocPoint` share of scheduled rewards and the rewards for stakers are proportional to their share of LP tokens in the pool.

In current implementation, there are a number of concurrent pools that share the rewarded tokens and more can be scheduled for addition (via a proper governance procedure). To accommodate these new pools, the design has the necessary mechanism in place that allows for dynamic additions of new staking pools that can participate in being incentivized as well.

The addition of a new pool is implemented in `add()`, whose code logic is shown below. It turns out it did not perform necessary sanity checks in preventing a new pool but with a duplicate token from being added. Though it is a privileged interface (protected with the modifier `onlyOwner`), it is still desirable to enforce it at the smart contract code level, eliminating the concern of wrong pool introduction from human omissions.

```
    // Add a new lp to the pool. Can only be called by the owner.
    // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
    function add(
        uint256 _allocPoint,
        IERC20 _lpToken,
        bool _withUpdate
    ) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(PoolInfo({lpToken: _lpToken, allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock, accRewardPerShare: 0}));
    }
```

Listing 3.9: `MojitoChef::add()`

Recommendation Detect whether the given pool for addition is a duplicate of an existing pool. The pool addition is only successful when there is no duplicate.

```
    function checkPoolDuplicate(IERC20 _lpToken) public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            require(poolInfo[_pid].lpToken != _lpToken, "add: existing pool?");
        }
    }

    // Add a new lp to the pool. Can only be called by the owner.
    // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
    function add(uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate) external
        onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        checkPoolDuplicate(_lpToken);
        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(PoolInfo({
            lpToken: _lpToken,
            allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock,
            accPerShare: 0
        }));
    }
```

Listing 3.10: Revised `MojitoChef::add()`

We point out that if a new pool with a duplicate LP token can be added, it will likely cause a havoc in the distribution of rewards to the pools and the stakers.

Status This issue has been fixed in the following commit: bfbc5c5.

3.6 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: MojitoChef
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

As mentioned earlier, the Chai Money Market provides incentive mechanisms that reward the staking of supported assets. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

The reward pools can be dynamically added via `add()` and the weights of supported pools can be adjusted via `set()`. When analyzing the pool weight update routine `set()`, we notice the need of timely invoking `massUpdatePools()` to update the reward distribution before the new pool weight becomes effective.

```
    function set(
        uint256 _pid,
        uint256 _allocPoint,
        bool _withUpdate
    ) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
        poolInfo[_pid].allocPoint = _allocPoint;
    }
```

Listing 3.11: MojitoChef::set()

If the call to `massUpdatePools()` is not immediately invoked before updating the pool weights, certain situations may be crafted to create an unfair reward distribution. Moreover, a hidden pool without any weight can suddenly surface to claim unreasonable share of rewarded tokens. Fortunately, this interface is restricted to the owner (via the `onlyOwner` modifier), which greatly alleviates the concern.

Recommendation Timely invoke `massUpdatePools()` when any pool's weight has been updated. In fact, the third parameter (`_withUpdate`) to the `set()` routine can be simply ignored or removed.

```
    function set(
        uint256 _pid,
```

```
        uint256 _allocPoint,
        bool _withUpdate
    ) public onlyOwner {
        massUpdatePools();
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint
            );
        poolInfo[_pid].allocPoint = _allocPoint;
    }
```

Listing 3.12: Revised MojitoChef::set()

Status This issue has been fixed in the following commit: bfbc5c5.

3.7 Staking Incompatibility With Deflationary Tokens

- ID: PVE-007
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: `MojitoChef`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

In the `Chai Money Market` protocol, the `MojitoChef` contract is designed to take users' assets and deliver rewards depending on their share. In particular, one interface, i.e., `deposit()`, accepts asset transfer-in and records the depositor's balance. Another interface, i.e, `withdraw()`, allows the user to withdraw the asset with necessary bookkeeping under the hood. For the above two operations, i.e., `deposit()` and `withdraw()`, the contract using the `safeTransfer()/safeTransferFrom()` routines to transfer assets into or out of its pool. This routine works as expected with standard ERC20 tokens: namely the pool's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    // Deposit LP tokens to MasterChef for reward token allocation.
    function deposit(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending = user.amount.mul(pool.accRewardPerShare).div(1e12).sub(user
                .rewardDebt);
            safeRewardTransfer(msg.sender, pending);
        }
        pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
        user.amount = user.amount.add(_amount);
        user.rewardDebt = user.amount.mul(pool.accRewardPerShare).div(1e12);
```

```
        emit Deposit(msg.sender, _pid, _amount);
    }
```

Listing 3.13: `MojitoChef::deposit())`

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every transfer. As a result, this may not meet the assumption behind asset-transferring routines. In other words, the above operations, such as `deposit()` and `withdraw()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of the pool and affects protocol-wide operation and maintenance.

Specially, if we take a look at the `updatePool()` routine. This routine calculates `pool.accPerShare` via dividing `tokenReward` by `lpSupply`, where the `lpSupply` is derived from `pool.lpToken.balanceOf(address(this))` (line 132). Because the balance inconsistencies of the pool, the `lpSupply` could be 1 `Wei` and thus may yield a huge `pool.accPerShare` as the final result, which dramatically inflates the pool's reward.

```
126     // Update reward variables of the given pool to be up-to-date.
127     function updatePool(uint256 _pid) public {
128         PoolInfo storage pool = poolInfo[_pid];
129         if (block.number <= pool.lastRewardBlock) {
130             return;
131         }
132         uint256 lpSupply = pool.lpToken.balanceOf(address(this));
133         if (lpSupply == 0) {
134             pool.lastRewardBlock = block.number;
135             return;
136         }
137         uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
138         uint256 reward = multiplier.mul(rewardPerBlock).mul(pool.allocPoint).div(
                totalAllocPoint);
139         pool.accRewardPerShare = pool.accRewardPerShare.add(reward.mul(1e12).div(
                lpSupply));
140         pool.lastRewardBlock = block.number;
141     }
```

Listing 3.14: `MojitoChef::updatePool()`

One mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `safeTransfer()` or `safeTransferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `safeTransfer()` or `safeTransferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into `MojitoChef` for support. However, certain existing stable coins may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation Check the balance before and after the `safeTransfer()` or `safeTransferFrom()` call to ensure the book-keeping amount is accurate. An alternative solution is using non-deflationary tokens as collateral but some tokens (e.g., USDT) allow the admin to have the deflationary-like features kicked in later, which should be verified carefully.

Status This issue has been confirmed and team will not use `MojitoChef` with deflationary token.

3.8 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MErc20`
- Category: Coding Practices [9]
- CWE subcategory: CWE-628 [5]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    /**
     * @notice A public function to sweep accidental ERC-20 transfers to this contract.
         Tokens are sent to admin (timelock)
     * @param token The address of the ERC-20 token to sweep
     */
    function sweepToken(EIP20NonStandardInterface token) external {
      require(address(token) != underlying, "MErc20::sweepToken: can not sweep
          underlying token");
      uint256 balance = token.balanceOf(address(this));
```

```
        token.transfer(admin, balance);
    }
```

Listing 3.15: ZRX.sol

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

In the following, we show the `sweepToken()` routine in the `MErc20` contract. If the `USDT` token is supported as `token`, the unsafe version of `IERC20(token).transfer(to, amount)` (line 128) may revert as there is no return value in the `USDT` token contract's `transfer()/transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
  /**
  * @notice A public function to sweep accidental ERC-20 transfers to this contract.
      Tokens are sent to admin (timelock)
  * @param token The address of the ERC-20 token to sweep
  */
 function sweepToken(EIP20NonStandardInterface token) external {
   require(address(token) != underlying, "MErc20::sweepToken: can not sweep underlying
       token");
   uint256 balance = token.balanceOf(address(this));
   token.transfer(admin, balance);
 }
```

Listing 3.16: MErc20::sweepToken()

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

Status This issue has been fixed in the following commit: bfbc5c5.

3.9 Trust Issue of Admin Keys

- ID: PVE-009
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [8]
- CWE subcategory: CWE-287 [4]

Description

The Chai Money Market protocol has a privileged owner account that plays a critical role in governing and regulating the protocol-wide operations (e.g., market addition, reward adjustment, and parameter setting). In the following, we show representative privileged operations in the protocol's core MojitoChef contract.

```
// Update the given pool's reward allocation point. Can only be called by the owner.
function set(
    uint256 _pid,
    uint256 _allocPoint,
    bool _withUpdate
) public onlyOwner {
    if (_withUpdate) {
        massUpdatePools();
    }
    totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
    poolInfo[_pid].allocPoint = _allocPoint;
}

function setfund(address _fund) external onlyOwner {
    require(_fund != address(0), "Invalid zero address");
    fund = _fund;
}
...
function setRewardPerBlock(uint256 _rewardPerBlock) external onlyOwner {
   massUpdatePools();
   rewardPerBlock = _rewardPerBlock;
}
```

Listing 3.17: MojitoChef::set()/setfund()/setRewardPerBlock()

Also we notice that the MojitoDollar contract has the privileged owner/minter accounts that play a critical role in governing and regulating the protocol-wide operations (e.g., set minter, mint/burn tokens). In the following, we show representative privileged operations in the protocol's MojitoDollar contract.

```
function mintTo(address _to, uint256 _amount) public onlyMinter {
    require(totalSupply().add(_amount) <= maxCap, "> maxCap");
    _mint(_to, _amount);
```

```
}

function burnFrom(address _from, uint256 _amount) public onlyMinter {
    _burn(_from, _amount);
}

//========================================
//  RESTRICTED FUNCTIONS
//========================================

function setMinter(address _newMinter) public onlyOwner {
    require(_newMinter != address(0), "invalid address");
    minter = _newMinter;
    emit MinterUpdated(_newMinter);
}

function setMaxCap(uint256 _maxCap) public onlyOwner {
    maxCap = _maxCap;
    emit MaxCapChanged(_maxCap);
}
}
```

Listing 3.18: `MojitoDollar::mintTo()/burnFrom()/setMinter()/setMaxCap()`

We emphasize that the privilege assignment may be necessary and consistent with the token design. However, it is worrisome if the privileged account is not governed by a DAO-like structure. Meanwhile, we point out that a compromised privileged account would allow the attacker to add a malicious strategy or change other settings, which directly undermines the assumption of the `Chai Money Market` protocol. Also notice that if the main protocol logic is deployed behind a proxy, which can be upgradeable via the authorized admin account, the trust of this admin account is also paramount to the entire protocol.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed 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 will have the owner transferred to a timelock contract or a multisig contract.

3.10 Proper Staking Token Initialization in MultiFeeDistribution

- ID: PVE-010
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `MultiFeeDistribution`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

The `Chai Money Market` protocol has a `MultiFeeDistribution` contract that allows users to stake tokens to earn rewards in various tokens. Naturally, this contract provides regular staking and unstaking functionality. While examining its current initialization logic, we notice the current implementation needs to be improved.

To elaborate, we show below the related `constructor()` function. This function configures the `stakingToken` and initializes the related `rewardData`. It comes to our attention that `rewardData` initialization is incomplete as it only configures the `lastUpdateTime` field (line 80). An improved approach also needs to set up the `periodFinish` field. An uninitialized `periodFinish` may return the wrong value of `lastTimeRewardApplicable()`, which simply returns `Math.min(block.timestamp, rewardData[_rewardsToken].periodFinish)`.

```
74     constructor(address _stakingToken, address _stakingTokenFund) Ownable() {
75         stakingToken = IERC20(_stakingToken);
76         stakingTokenFund = IFund(_stakingTokenFund);
77         // First reward MUST be the staking token or things will break
78         // related to the 50% penalty and distribution to locked balances
79         rewardTokens.push(_stakingToken);
80         rewardData[_stakingToken].lastUpdateTime = block.timestamp;
81     }
```

Listing 3.19: `MultiFeeDistribution::constructor()`

Recommendation Improve the above routine by initializing the `periodFinish` field for `stakingToken`-related `rewardData`.

Status This issue has been fixed in the following commit: f068be9.

4 Conclusion

In this audit, we have analyzed the `Chai Money Market` protocol design and implementation. The protocol is designed to be an money market that is inspired from `Compound`. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

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

References

[1] 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[5] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

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

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

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

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

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

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

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

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

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

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

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