SMART CONTRACT AUDIT REPORT

for

Paxo Protocol

Prepared By: Xiaomi Huang

PeckShield

February 16, 2023

Document Properties

Client	Paxo
Title	Smart Contract Audit Report
Target	Paxo
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Patrick Lou, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 16, 2023	Xuxian Jiang	Final Release
1.0-rc1	February 8, 2023	Xuxian Jiang	Release Candidate #1
0.8	February 1, 2023	Xuxian Jiang	Initial Draft

Contact

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

Name	Xiaomi Huang
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 Paxo 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 Paxo

Paxo is a decentralized money market protocol that opens up investment loan options in DeFi for global users to invest in digital assets via multi-pool borrowing. Its investment loans are used to invest in other digital assets from a connected DEX via the Paxo protocol. Afterward, the acquired asset (investment) is securely locked in Paxo until the loan is repaid with interest. If not, the asset is liquidated in an auction when it reaches the liquidation point. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Paxo

Item	Description
Name	Paxo
Website	https://paxo.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 16, 2023

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

- https://github.com/paxo-finance/paxo-protocol.git (6ac0079)

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

- https://github.com/paxo-finance/paxo-protocol.git (d6c2efe)

1.2 About PeckShield

PeckShield Inc. [8] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [7]:

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

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

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract

Table 1.3: The Full Audit Checklist

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

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

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered 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 Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

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

Severity	# of Findings	
Critical	1	■
High	1	■
Medium	1	■
Low	2	■ ■
Informational	0	
Total	5	

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 critical-severity vulnerability, 1 high-severity vulnerability, 1 medium-severity vulnerability, and 2 low-severity vulnerabilities.

Table 2.1: Key Paxo Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Critical	Improper Approval to Drain Market Funds in CErc20	Business Logic	Fixed
PVE-002	High	Uninitialized State Index DoS From Reward Activation	Coding Practices	Fixed
PVE-003	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Fixed
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Fixed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Besides the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Improper Approval to Drain Market Funds in CErc20

- ID: PVE-001
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `CErc20`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

Each asset supported by the `Paxo` protocol is integrated through a so-called `CToken` contract, which is an ERC20 compliant representation of balances supplied to the protocol. By minting `CTokens`, users can earn interest through the `CToken`'s exchange rate, which increases in value relative to the underlying asset, and further gain the ability to use `CTokens` as collateral. While examining the exposed functions in the `CToken`-inherited `CErc20`, we notice one particular one puts the pool funds at risk.

To elaborate, we show below the related function – `increaseMarketAllowance()`. As the name indicates, this function is used to increase the market allowance. However, it comes to our attention that the given input asset (line 304) is not validated, which may be exploited to grant spend allowance to the given input asset. And the given input asset address may not be trusted!

```
300     function getCTokenUnderlying(address cToken) view internal returns (address) {
301         return CTokenU(cToken).underlying();
302     }
303
304     function increaseMarketAllowance(address asset) external {
305         EIP20Interface(getCTokenUnderlying(asset)).approve(asset, type(uint).max);
306     }
```

Listing 3.1: `CErc20::increaseMarketAllowance()`

Recommendation Validate the input asset in the above `increaseMarketAllowance()` routine.

Status This issue has been fixed in the following commit: 8847888.

3.2 Uninitialized State Index DoS From Reward Activation

- ID: PVE-002
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: Comptroller
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The Paxo 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 the rewarding logic of the protocol token coded as COMP.

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 1090). 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.

```
1078     function setCompSpeedInternal(CToken cToken, uint compSpeed) internal {
1079         uint currentCompSpeed = compSpeeds[address(cToken)];
1080         if (currentCompSpeed != 0) {
1081             // note that COMP speed could be set to 0 to halt liquidity rewards for a
                     market
1082             Exp memory borrowIndex = Exp({mantissa: cToken.borrowIndex()});
1083             updateCompSupplyIndex(address(cToken));
1084             updateCompBorrowIndex(address(cToken), borrowIndex);
1085         } else if (compSpeed != 0) {
1086             // Add the COMP market
1087             Market storage market = markets[address(cToken)];
1088             require(market.isListed == true, "CMktNtL"); //comp market is not listed

1090             if (compSupplyState[address(cToken)].index == 0 && compSupplyState[address(
                     cToken)].block == 0) {
1091                 compSupplyState[address(cToken)] = CompMarketState({
1092                     index: compInitialIndex,
```

```
                block: safe32(getBlockNumber(), "B#Ex") //block number exceeds 32 bits
            });
        }

        if (compBorrowState[address(cToken)].index == 0 && compBorrowState[address(cToken)].block == 0) {
            compBorrowState[address(cToken)] = CompMarketState({
                index: compInitialIndex,
                block: safe32(getBlockNumber(), "B#Ex") //block number exceeds 32 bits
            });
        }
    }

    if (currentCompSpeed != compSpeed) {
        compSpeeds[address(cToken)] = compSpeed;
        emit CompSpeedUpdated(cToken, compSpeed);
    }
}
```

Listing 3.2: `Comptroller::setCompSpeedInternal()`

```
function updateCompSupplyIndex(address cToken) internal {
    CompMarketState storage supplyState = compSupplyState[cToken];
    uint supplySpeed = compSpeeds[cToken];
    uint blockNumber = getBlockNumber();
    uint deltaBlocks = sub_(blockNumber, uint(supplyState.block));
    if (deltaBlocks > 0 && supplySpeed > 0) {
        uint supplyTokens = CToken(cToken).totalSupply();
        uint compAccrued = mul_(deltaBlocks, supplySpeed);
        Double memory ratio = supplyTokens > 0 ? fraction(compAccrued, supplyTokens) : Double({mantissa: 0});
        Double memory index = add_(Double({mantissa: supplyState.index}), ratio);
        compSupplyState[cToken] = CompMarketState({
            index: safe224(index.mantissa, "NIdxEx"), //new index exceeds 224 bits
            block: safe32(blockNumber, "B#Ex") //block number exceeds 32 bits
        });
    } else if (deltaBlocks > 0) {
        supplyState.block = safe32(blockNumber, "B#Ex"); //block number exceeds 32 bits
    }
}
```

Listing 3.3: `Comptroller::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 1172). Since the `supplyIndex` is not properly initialized, it will be updated to a smaller number from an earlier invocation of `updateCompSupplyIndex()` (lines 1126-1127). However, when the `distributeSupplierComp()` function is invoked, the `supplierIndex` is reset with `compInitialIndex` (line 1169), which unfortunately reverts the arithmetic operation `sub_(supplyIndex, supplierIndex)`!

```
1162    function distributeSupplierComp(address cToken, address supplier) internal {
1163        CompMarketState storage supplyState = compSupplyState[cToken];
1164        Double memory supplyIndex = Double({mantissa: supplyState.index});
1165        Double memory supplierIndex = Double({mantissa: compSupplierIndex[cToken][
                supplier]});
1166        compSupplierIndex[cToken][supplier] = supplyIndex.mantissa;

1168        if (supplierIndex.mantissa == 0 && supplyIndex.mantissa > 0) {
1169            supplierIndex.mantissa = compInitialIndex;
1170        }

1172        Double memory deltaIndex = sub_(supplyIndex, supplierIndex);
1173        uint supplierTokens = CToken(cToken).balanceOf(supplier);
1174        uint supplierDelta = mul_(supplierTokens, deltaIndex);
1175        uint supplierAccrued = add_(compAccrued[supplier], supplierDelta);
1176        compAccrued[supplier] = supplierAccrued;
1177        emit DistributedSupplierComp(CToken(cToken), supplier, supplierDelta,
                supplyIndex.mantissa);
1178    }
```

Listing 3.4: `Comptroller::distributeSupplierComp()`

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

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

            if (compSupplyState[address(cToken)].index == 0) {
                compSupplyState[address(cToken)] = CompMarketState({
                    index: compInitialIndex,
                    block: safe32(getBlockNumber(), "B#Ex") //block number exceeds 32
                        bits
                });
            }

            if (compBorrowState[address(cToken)].index == 0) {
                compBorrowState[address(cToken)] = CompMarketState({
```

```
                    index: compInitialIndex,
                    block: safe32(getBlockNumber(), "B#Ex") //block number exceeds 32
                        bits
                });
            }
        }

        if (currentCompSpeed != compSpeed) {
            compSpeeds[address(cToken)] = compSpeed;
            emit CompSpeedUpdated(cToken, compSpeed);
        }
    }
```

Listing 3.5: Revised `Comptroller::setCompSpeedInternal()`

Status This issue has been fixed in the following commit: `8847888`.

3.3 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

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` [10] exploit, and the recent `Uniswap/Lendf.Me` hack [9].

We notice there are occasions where the `checks-effects-interactions` principle is violated. Using the `CToken` 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 1022) start before effecting the update on internal states (lines 1027 – 1029), 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.

```
    function borrowFresh(address payable borrower, uint borrowAmount, bool credited)
        internal returns (uint) {
        /* Fail if borrow not allowed */
        uint allowed = comptroller.borrowAllowed(address(this), borrower, borrowAmount);
        if (allowed != 0) {
            return failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.
                BORROW_COMPTROLLER_REJECTION, allowed);
        }

        /* Verify market's block number equals current block number */
        if (accrualBlockNumber != getBlockNumber()) {
            return fail(Error.MARKET_NOT_FRESH, FailureInfo.BORROW_FRESHNESS_CHECK);
        }

        /* Fail gracefully if protocol has insufficient underlying cash */
        if (getCashPrior() < borrowAmount) {
            return fail(Error.TOKEN_INSUFFICIENT_CASH, FailureInfo.
                BORROW_CASH_NOT_AVAILABLE);
        }

        BorrowLocalVars memory vars;

        /*
         * 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, uint(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, uint(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, uint(vars.mathErr));
        }

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

        /*
```

```
         * We invoke doTransferOut for the borrower and the borrowAmount.
         *  Note: The cToken must handle variations between ERC-20 and ETH underlying.
         *  On success, the cToken borrowAmount less of cash.
         *  If doTransferOut fails despite the fact we checked pre-conditions,
         *   we revert because we can't be sure if side effects occurred.
         */

        if(!credited){
            vars.err = doTransferOut(borrower, borrowAmount);
            require(vars.err == Error.NO_ERROR, "F"); //borrow transfer out failed
        }

        /* 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 */

        // comptroller.borrowVerify(address(this), borrower, borrowAmount);

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

Listing 3.6: `CToken::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 other functions, including `redeemFresh()` 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 the `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 This issue has been fixed in the following commit: `8847888`.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

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."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.7: 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 `setDexAggregator()` routine in the `CErc20` contract. If the `USDT` token is supported as one of `underlying`, the unsafe version of `EIP20Interface(underlying).approve(uniswapV2, type(uint).max)` (line 390) 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)! Also, the previous approve on `uniswapV2` needs to be cancelled.

```
397     function setDexAggregator(address newDexAggregator) external {
398         require(msg.sender == admin, "0");//unauthorized
399         uniswapV2 = newDexAggregator;
400         EIP20Interface(underlying).approve(uniswapV2, type(uint).max);
401     }
```

Listing 3.8: `CErc20::setDexAggregator()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`. This issue affects a number of routines in various contracts, including `Comptroller::_supportMarket()`, `CErc20::increaseMarketAllowance()`, and `CErc20::swapAndSettle()`.

Status This issue has been fixed in the following commit: `8847888`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Paxo` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and marketing adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    function _supportBuy(CToken cToken) external returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_COLLATERAL_FACTOR_OWNER_CHECK);
        }

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

        market.isBuyAvailable = true;

        emit MarketBuyListed(cToken);

        return uint(Error.NO_ERROR);
    }


    function _setBuyFactor(CToken cToken, uint newBuyFactorMantissa) external returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_COLLATERAL_FACTOR_OWNER_CHECK);
        }

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

        if (newBuyFactorMantissa != 0 && oracle.getUnderlyingPrice(cToken) == 0) {
            return fail(Error.PRICE_ERROR, FailureInfo.SET_COLLATERAL_FACTOR_WITHOUT_PRICE);
        }

        // Set market's buy factor to new buy factor, remember old value
        uint oldBuyFactorMantissa = market.buyFactorMantissa;
        market.buyFactorMantissa = newBuyFactorMantissa;

        // Emit event with asset, old buy factor, and new buy factor
        emit NewBuyFactor(cToken, oldBuyFactorMantissa, newBuyFactorMantissa);

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

Listing 3.9: Example `Setters` in the `Comptroller` Contract

Apparently, if the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that current contracts have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

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 with the team. For the time being, the team has confirmed that these privileged functions should be called by a trusted multi-sig account, not a plain EOA account.

4 Conclusion

In this audit, we have analyzed the `Paxo` design and implementation. `Paxo` is a decentralized money market protocol that opens up investment loan options in `DeFi` for global users to invest in digital assets via multi-pool borrowing. Its investment loans are used to invest in other digital assets from a connected `DEX` via the `Paxo` protocol. Afterward, the acquired asset (investment) is securely locked in `Paxo` until the loan is repaid with interest. If not, the asset is liquidated in an auction when it reaches the liquidation point. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

References

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

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

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

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

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

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

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

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

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