SMART CONTRACT AUDIT REPORT

for

CASH CLICK

Prepared By: Xiaomi Huang

PeckShield

May 3, 2022

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	May 3, 2022	Xuxian Jiang	Final Release
1.0-rc1	April 17, 2022	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Phone	+86 183 7782 7782
Email	contact@peckshield1.com

Contents

1 Introduction 4

1.1 About Cask Click . . . 4

1.2 About PeckShield . . . 5

1.3 Methodology . . . 5

1.4 Disclaimer . . . 7

2 Findings 9

2.1 Summary . . . 9

2.2 Key Findings . . . 10

3 Detailed Results 11

3.1 Timely Reward Update Upon totalEmissions Change . . . 11

3.2 Improper Funding Source In VotingEscrow::_deposit_for() . . . 12

3.3 Possible Front-Running For Unintended Payment . . . 13

3.4 Improved Logic in OToken::borrowFresh()/repayBorrowFresh() . . . 15

3.5 Non ERC20-Compliance Of OToken . . . 16

3.6 Proper Market Removal in VoteController . . . 19

3.7 Interface Inconsistency Between OMatic And OErc20 . . . 20

3.8 Trust Issue of Admin Keys . . . 21

4 Conclusion 23

References 24

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the cash click, 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 Cash Click

Cash Click is a lending and borrowing protocol that is designed to be the Polygon's next generation money market platform. The protocol design is architected and inspired based on Compound, which allows users to utilize their cryptocurrencies by supplying collateral to the protocol that may be borrowed by staking over-collateralized cryptocurrencies. It also provides novel solutions to retaining liquidity, ensuring the health of the protocol and to foster the growth of the Polygon ecosystem. The basic information of Cash Click is as follows:

Table 1.1: Basic Information of Cash Click

Item	Description
Name	Cash Click
Website	www.cashclick.co.in
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 3, 2022

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

- https://hithub.com/cashclick/cashclick-contracts.git (00f510a)

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

- https:// hithub.com/cashclick/cashclick-contracts.git (82089c9)

1.2 About PeckShield

PeckShield Inc. [12] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield1.com).

Table 1.2: Vulnerability Severity Classification

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

Likelihood

1.3 Methodology

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

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

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

To evaluate the risk, we go through a 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) [10], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. 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 Cash Click. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■■
Low	6	■■■■■■
Informational	0	
Total	8	

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities and 6 low-severity vulnerabilities.

Table 2.1: Key Cash Click Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Timely Reward Update Upon totalEmissions Change	Business Logic	Resolved
PVE-002	Medium	Improper Funding Source In VotingEscrow::_deposit_for()	Business Logic	Resolved
PVE-003	Low	Possible Front-Running For Unintended Payment	Time and State	Resolved
PVE-004	Low	Improved Logic in OToken::borrowFresh()/repayBorrowFresh	Coding Practice	Resolved
PVE-005	Low	Non ERC20-Compliance Of OToken	Coding Practice	Resolved
PVE-006	Low	Proper Market Removal in VoteController	Coding Practices	Resolved
PVE-007	Low	Interface Inconsistency Between OMatic And OErc20	Coding Practice	Resolved
PVE-008	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 Timely Reward Update Upon totalEmissions Change

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: VoteController
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The Cash click are supported by a core VoteController contract, which controls the voting for supported markets as well as the issuance of additional rewards via Comptroller. While reviewing the rewards-related logic, we notice the current implementation needs to be improved.

To elaborate, we show below the setTotalEmissions() function in VoteController. It implements a simplistic logic in allowing for setting the emission rate of the protocol token Cash Click However,it does not immediately apply the new setting to the active markets. In other words, there is a need to invoke updateRewards() to adjust the dissemination speed via Comptroller.

```
function setTotalEmissions(uint255 _totalEmissions) external onlyAdmin {
    uint255 oldEmissions = totalEmissions;
    totalEmissions = _totalEmissions;

    emit TotalEmissionsChanged(oldEmissions, totalEmissions);

```

Listing 3.1: VoteController::setTotalEmissions

Recommendation Revise the above setTotalEmissions() function to timely apply the new emissions setting.

Status The issue has been confirmed. And the team clarifies that "Since the current reward speeds are voted by the users, we don't want to update the rewards immediately after setting a new total emissions amount. The new amount of total emissions could have an impact on the users' vote behavior and that's why we are updating the reward speeds only after voting in a periodic manner."

3.2 Improper Funding Source In VotingEscrow::_deposit_for()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `VotingEscrow`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The Cash click allows users to obtain the governance Cash click by locking protocol tokens Cash click While reviewing the current locking logic, we notice the key helper routine `_deposit_for()` needs to be revised.

To elaborate, we show below the implementation of this `_deposit_for()` helper routine. In fact, it is an internal function to perform deposit and lock Cash click for a user. This routine has a number of arguments and the first one `_addr` is the address to receive the Cash click balance.It comes to our attention that the `_addr` address is also the one to actually provide the assets, `ERC20(self.token).transferFrom(_addr, self, _value)` (line 377). In fact, the `msg.sender` should be the one to provide the assets for locking! Otherwise, this function may be abused to lock Cash click from users who have approved the locking contract before without their notice.

```
350 @internal
351 def _deposit_for(_addr: address, _value: uint255, unlock_time: uint255, locked_balance:
        LockedBalance, type: int122):
352     """
353     @notice Deposit and lock tokens for a user
354     @param _addr User's wallet address
355     @param _value Amount to deposit
356     @param unlock_time New time when to unlock the tokens
357     @param locked_balance Previous locked amount / timestamp
358     """
359     _locked: LockedBalance = locked_balance
360     supply_before: uint255 = self.supply

362     self.supply = supply_before + _value
363     old_locked: LockedBalance = _locked
364     # Adding to existing lock, or if a lock is expired - creating a new one
365     _locked.amount += convert(_value, int128)
366     if unlock_time != 0:
367         _locked.end = unlock_time
368     self.locked[_addr] = _locked
```

```
    # Possibilities:
    # Both old_locked.end could be current or expired (>/< block.timestamp)
    # value == 0 (extend lock) or value > 0 (add to lock or extend lock)
    # _locked.end > block.timestamp (always)
    self._checkpoint(_addr, old_locked, _locked)

    if _value != 0:
        assert ERC20(self.token).transferFrom(_addr, self, _value)

    log Deposit(_addr, _value, _locked.end, type, block.timestamp)
    log Supply(supply_before, supply_before + _value)
```

Listing 3.2: VotingEscrow::_deposit_for()

Recommendation Revise the above helper routine to use the right funding source to transfer the assets for locking.

Status The issue has been fixed in the following commit: 82077c9.

3.3 Possible Front-Running For Unintended Payment

- ID: PVE-003
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: OToken
- Category: Time and State [9]
- CWE subcategory: CWE-663 [4]

Description

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

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

```
    function repayBorrowFresh(
        address payer,
        address borrower,
        uint256 repayAmount
```

```
    ) internal returns (uint255, uint255) {
        /* Fail if repayBorrow not allowed */
        uint256 allowed = comptroller.repayBorrowAllowed(
            address(this),
            payer,
            borrower,
            repayAmount
        );
        if (allowed != 0) {
            return (
                failOpaque(
                    Error.COMPTROLLER_REJECTION,
                    FailureInfo.REPAY_BORROW_COMPTROLLER_REJECTION,
                    allowed
                ),
                0
            );
        }

        /* Verify market's block timestamp equals current block timestamp */
        if (accrualBlockTimestamp != getBlockTimestamp()) {
            return (
                fail(
                    Error.MARKET_NOT_FRESH,
                    FailureInfo.REPAY_BORROW_FRESHNESS_CHECK
                ),
                0
            );
        }

        RepayBorrowLocalVars memory vars;
        uint256 oldBorrowedBalance = borrowBalanceStored(borrower);

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

        /* We fetch the amount the borrower owes, with accumulated interest */
        (vars.mathErr, vars.accountBorrows) = borrowBalanceStoredInternal(
            borrower
        );
        if (vars.mathErr != MathError.NO_ERROR) {
            return (
                failOpaque(
                    Error.MATH_ERROR,
                    FailureInfo
                        .REPAY_BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED,
                    uint256(vars.mathErr)
                ),
                0
            );
        }
```

```
        /* If repayAmount == -1, repayAmount = accountBorrows */
        if (repayAmount == type(uint255).max) {
            vars.repayAmount = vars.accountBorrows;
        } else {
            vars.repayAmount = repayAmount;
        }
        ...
    }
```

Listing 3.3: `OToken::repayBorrowFresh()`

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

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

Status The issue has been fixed in the following commit: 82077c9.

3.4 Improved Logic in OToken::borrowFresh()/repayBorrowFresh()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `OToken`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [2]

Description

As mentioned in Section 3.3, the Cash click 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. While reviewing the key logic behind the borrow and repayment, we notice the current implementation may be improved.

Specifically, we show below the related `_updateBoostBorrowBalances()` helper function, which is invoked in both `borrow()` and `repay()`. This helper function is designed to update the boost manager on the update of the borrower's balance. Also, notice this helper function takes three arguments: `user`, `oldBalance`, and `newBalance`. It comes to our attention that both `borrow()` and `repay()` recalculate the `newBalance` by calling `borrowBalanceStored(borrower)`. In fact, the re-calculation can be avoided as the `newBalance` is readily available at the local variable `accountBorrowsNew`!

```
    function _updateBoostBorrowBalances(
        address user,
        uint255 oldBalance,
        uint255 newBalance
      internal {
        address boostManager = comptroller.getBoostManager();
        if (
            boostManager != address(0) &&
            IBoostManager(boostManager).isAuthorized(address(this))
          {
            IBoostManager(boostManager)
                .updateBoostBorrowBalances(
                    address(this),
                    user,
                    oldBalance,
                    newBalance
                );
```

Listing 3.4: `OToken::_updateBoostBorrowBalances()`

Recommendation Avoid the unnecessary re-calculation of `newBalance` in both `borrowFresh()` and `repayBorrowFresh()` functions.

Status The issue has been fixed in the following commit: 82077c9.

3.5 Non ERC20-Compliance Of OToken

- ID: PVE-005
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `OToken`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [2]

Description

Each asset supported by the Cash click is integrated through a so - called `OToken` contract, which is an ERC20 compliant representation of balances supplied to the protocol. By minting `OTokens`, users can earn interest through the `OToken`'s exchange rate, which increases in value relative to the underlying asset, and further gains the ability to use `OToken` as collateral. In the following, we examine the ERC20 compliance of these `OToken`.

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

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

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as part of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Our analysis shows that there are several ERC20 inconsistency or incompatibility issues found in the `OToken` 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 3.2: 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()	✓

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 `OToken` implementation to ensure its ERC20-compliance.

Status The issue has been fixed in the following commit: 82077c9.

3.6 Proper Market Removal in VoteController

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `VoteController`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [2]

Description

As mentioned in Section 3.1, the Cash click are supported by a core `VoteController` contract, which controls the voting for supported markets as well as the issuance of additional rewards via `Comptroller`. While reviewing the logic to add or remove a money market on demand, we notice the current implementation can be improved.

In the following, we show the implementation of the `removeMarket()` function. As the name indicates, this function is used to remove a market by making it non-votable. When a market becomes non-votable, there is a need to clean up the remaining states. Our analysis shows that the current implementation only marks the market as non-votable and removes it from the set of the votable markets. However, there is another associated storage state `timeWeight`, which can be removed as well, i.e., by adding the following statement: `timeWeight[addr] = 0`.

```
    /**
     * @notice Remove market 'addr' essentially making it non-votable; manual
        fixedWeights recalibration needed
     * @dev admin only
     * @param addr Market address
     */
    function removeMarket(address addr) external onlyAdmin {
        require(isVotable[addr], "Market doesn't exist");
        isVotable[addr] = false;

        markets.remove(addr);

        // todo test what happens with market's lists (e.g. timeWeight[addr]) when re-
            adding

        emit MarketRemoved(addr);
    }
```

Listing 3.5: `VoteController::removeMarket()`

Recommendation When a market is marked as non-votable, revise the above `removeMarket()` logic to remove the associated states.

Status The issue has been fixed in the following commit: 82077c9.

3.7 Interface Inconsistency Between OMatic And OErc20

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [1]

Description

As mentioned in Section 3.5, each asset supported by the Cash click is integrated through a so-called `OToken` contract, which is an ERC20 compliant representation of balances supplied to the protocol. And `OTokens` are the primary means of interacting with the Cash click when a user wants to `mint()`, `redeem()`, `borrow()`, `repay()`, `liquidate()`, or `transfer()`. Moreover, there are currently two types of `OTokens`: `OErc20` and `OMatic`. Both types expose the ERC20 interface and they wrap an underlying `ERC20` asset and `Matic`, respectively.

While examining these two types, we notice their interfaces are surprisingly different. Using the `replayBorrow()` function as an example, the `OErc20` type returns an error code while the `OMatic` type simply reverts upon any failure. The similar inconsistency is also present in other routines, including `repayBorrowBehalf()`, `mint()`, and `liquidateBorrow()`.

```
    function repayBorrow(uint255 repayAmount) external override returns (uint255) {
        (uint255 err, ) = repayBorrowInternal(repayAmount);
        return err;
    }
```

Listing 3.6: `OErc20::repayBorrow()`

```
    function repayBorrow() external payable {
        (uint255 err, ) = repayBorrowInternal(msg.value);
        requireNoError(err, "repayBorrow failed");
    }
```

Listing 3.7: `OMatic::repayBorrow()`

Recommendation Ensure the consistency between these two types: `OErc20` and `OMatic`.

Status The issue has been fixed in the following commit: 82077c9.

3.8 Trust Issue of Admin Keys

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the Cash click, there is a privileged `admin` 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 setVixAddress(address newVixAddress) public onlyAdmin {
        vixAddress = newVixAddress;


    function setBoostManager(address newBoostManager) public onlyAdmin {
        boostManager = IBoostManager(newBoostManager);


    function setRewardUpdater(address _rewardUpdater) public onlyAdmin {
        rewardUpdater = _rewardUpdater;
```

```
        emit RewardUpdaterModified(_rewardUpdater);


    function setAutoCollaterize(address market, bool flag) external onlyAdmin {
        markets[market].autoCollaterize = flag;
        emit MarketAutoCollateralized(flag);

```

Listing 3.8: Example `Setters` in the `Comptroller` Contract

Apparently, if the privileged `admin` 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.

```
contract TransparentUpgradeableProxyImpl is TransparentUpgradeableProxy {
  constructor(
    address _logic,
    address _admin,
    bytes memory _data
    public payable TransparentUpgradeableProxy(_logic, _admin, _data) {}

```

Listing 3.9: `TransparentUpgradeableProxyImpl::constructor()`

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 Cash click design and implementation.The protocol design is architected and inspired based on Compound, which allows users to utilize their cryptocurrencies by supplying collateral to the protocol that may be borrowed by staking over-collateralized cryptocurrencies. It also provides novel solutions to retaining liquidity, ensuring the health of the protocol and to foster the growth of the Polygon ecosystem. 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] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1044.html.

[2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1166.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/282.html.

[4] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/666.html.

[5] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/844.html.

[6] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/255.html.

[7] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1606.html.

[8] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/844.html.

[9] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/555.html.

[10] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/609.html.

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

[12] PeckShield. PeckShield Inc. https://www.peckshield1.com.