SMART CONTRACT AUDIT REPORT

for

Qubit Finance

Prepared By: Yiqun Chen

PeckShield

August 13, 2021

Document Properties

Client	Qubit Finance
Title	Smart Contract Audit Report
Target	Qubit Finance
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 13, 2021	Xuxian Jiang	Final Release
1.0-rc1	August 7, 2021	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

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

1.1 About Qubit Finance

`Qubit Finance` is a DeFi lending protocol developed by `MOUND`, the team behind `Pancake Bunny`, a leading yield aggregator on the BSC. It allows users to freely deposit virtual assets as collateral in order to borrow virtual assets and, the interest is automatically calculated by the smart contract protocol. `Qubit` is designed as a commodity for the benefit of the DeFi ecosystem. In keeping with this ethos, `Qubit` does not charge withdrawal fees so that services utilizing `Qubit` as a building block can maximize their returns.

The basic information of Qubit Finance is as follows:

Table 1.1: Basic Information of Qubit Finance

Item	Description
Name	Qubit Finance
Website	https://qbt.fi/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 13, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that `Qubit` assumes a trusted price oracle with timely market price feeds for supported assets and the oracle itself is not part of this audit.

- https://github.com/PancakeBunny-finance/qubit-finance.git (8a4c7de)

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

- https://github.com/PancakeBunny-finance/qubit-finance.git (f37f63c)

1.2 About PeckShield

PeckShield Inc. [13] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [12]:

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

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

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

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

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

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

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

1.4 Disclaimer

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

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

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

comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

2 Findings

2.1 Summary

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

Severity	# of Findings	
Critical	0	
High	0	
Medium	3	■ ■ ■
Low	5	■ ■ ■ ■ ■
Informational	1	■
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 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 3 medium-severity vulnerabilities, 5 low-severity vulnerabilities, 1 informational recommendation, and 1 undetermined issue.

Table 2.1: Key Qubit Finance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Undetermined	QToken-Wide vs. Protocol-Wide Reentrancy Protection	Business Logic	Fixed
PVE-002	Low	Non ERC20-Compliance Of QToken	Coding Practices	Fixed
PVE-003	Low	Possible Front-running For Unintended Payment In repayBorrowBehalf()	Time And State	Fixed
PVE-004	Low	Strengthened Self-Transfer Handling in notifyTransferred()	Coding Practice	Fixed
PVE-005	Low	Simplified Logic in removeMarket() for Gas Optimization	Coding Practice	Fixed
PVE-006	Informational	Redundant State/Code Removal	Coding Practice	Fixed
PVE-007	Low	Suggested Emit of Related Events Upon Settings Changes	Coding Practices	Fixed
PVE-008	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-009	Medium	Unsound PriceCalculator Assumption On Asset Decimals	Business Logic	Fixed
PVE-010	Medium	Proper accruedQubit() Calculation	Business Logic	Fixed

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

3 | Detailed Results

3.1 QToken-Wide vs. Protocol-Wide Reentrancy Protection

- ID: PVE-001
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The `Qubit` protocol is a DeFi lending protocol that is heavily inspired from `Compound` with a variety of improvements on the code quality and supported features. During our analysis, we notice the current implementation places necessary reentrancy protection at the granularity of each individual `QToken`, which is a back-end unit of account for the `Qubit` protocol.

To elaborate, we show below the `borrow()` function. As the name indicates, it support the basic lending functionality in borrowing certain assets from the protocol. It comes to our attention that this function has the associated `nonReentrant` modifier, which indeed blocks possible reentrancy at each market level. However, considering the protocol-wide cross-market assets for collateralization and validation on lending operations, we suggest to enforce the reentrancy prevention at the protocol-level, i.e., at the corresponding `borrow()` function at the `Qore` contract.

```
    function borrow(address account, uint amount) external override accrue onlyQore
        nonReentrant returns (uint) {
        require(getCash() >= amount, "QToken: borrow amount exceeds cash");
        updateBorrowInfo(account, amount, 0);
        _doTransferOut(account, amount);

        emit Borrow(account, amount, borrowBalanceOf(account));
        return amount;
    }
```

Listing 3.1: `QToken::borrow()`

```
    function borrow(address qToken, uint amount) external override onlyListedMarket(qToken) {
        _enterMarket(qToken, msg.sender);
        require(IQValidator(qValidator).borrowAllowed(qToken, msg.sender, amount), "Qore: cannot borrow");

        IQToken(payable(qToken)).borrow(msg.sender, amount);
        qDistributor.notifyBorrowUpdated(qToken, msg.sender);
    }
```

Listing 3.2: `Qore::borrow()`

Recommendation Enforce the reentrancy prevention at the protocol-wide level, instead of each individual market.

Status The issue has been fixed by this commit: `30aef6f`.

3.2 Non ERC20-Compliance Of QToken

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `QToken`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [2]

Description

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

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 exists certain ERC20 inconsistency or incompatibility issues found in the `QToken` contract. Specifically, when `QTokens` are minted or burned, current `Transfer` event needs to be emitted with properly-encoded information. However, current event uses the contract itself (not `address(0)`) as the address to reflect the mint/burn operations.

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	✓

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.

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

Status The issue has been fixed by this commit: `0df221b1`.

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	✓

3.3 Possible Front-running For Unintended Payment In repayBorrowBehalf()

- ID: PVE-003
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `QToken`
- Category: Time and State [10]
- CWE subcategory: CWE-663 [5]

Description

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

To elaborate, we show below the routine `repayBorrowBehalf()` that allows a non-borrower to repay partial or full current borrowing balance. It is interesting to note that the `Qubit` protocol supports the payment on behalf of another borrowing user. And the logic 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 repayBorrowBehalf(
        address payer,
        address borrower,
        uint amount
```

```
    ) external payable override accrue onlyQore nonReentrant returns (uint) {
        if (amount == uint(-1)) {
            amount = borrowBalanceOf(borrower);
        }
        return _repay(payer, borrower, underlying == address(WBNB) ? msg.value : amount)
            ;
    }
```

Listing 3.3: `QToken::repayBorrowBehalf()`

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 by this commit: `af440fb`.

3.4 Strengthened Self-Transfer Handling in notifyTransferred()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `QDistributor`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [2]

Description

The `Qubit` protocol has a built-in incentive mechanism to reward protocol users upon a variety of protocol operations, such as `mint()`, `redeem()`, `borrow()`, and `repay()`. In the following, we show the internal handler `notifyTransferred()` in the `QDistributor` contract. This handler is used to properly keep track of the reward amount for related accounts when their balances are updated.

To elaborate, we show below the `notifyTransferred()` routine. It firstly computes the rewards up to the current moment and then updates the balances for both `sender` and `receiver`. It comes to our attention this current routine works fine on the condition that `sender` is not equal to `receiver`. Fortunately, the current implementation has placed necessary validations so that this routine will not be invoked when they are the same. However, as a security pre-caution, we recommend the explicit disallowance within the `notifyTransferred()` routine to filter out the unsupported case, i.e., `require(sender!=receiver)`.

```
    function notifyTransferred(
        address qToken,
        address sender,
        address receiver
    ) external override nonReentrant onlyMarket updateDistributionOf(qToken) {
        DistributionInfo storage dist = distributions[qToken];
        UserInfo storage senderInfo = marketUsers[qToken][sender];
        UserInfo storage receiverInfo = marketUsers[qToken][receiver];

        if (senderInfo.boostedSupply > 0) {
            uint accQubitPerShare = dist.accPerShareSupply.sub(senderInfo.
                accPerShareSupply);
            senderInfo.accruedQubit = senderInfo.accruedQubit.add(
                accQubitPerShare.mul(senderInfo.boostedSupply).div(1e18)
            );
        }
        senderInfo.accPerShareSupply = dist.accPerShareSupply;

        if (receiverInfo.boostedSupply > 0) {
            uint accQubitPerShare = dist.accPerShareSupply.sub(receiverInfo.
                accPerShareSupply);
            receiverInfo.accruedQubit = receiverInfo.accruedQubit.add(
                accQubitPerShare.mul(receiverInfo.boostedSupply).div(1e18)
            );
        }
        receiverInfo.accPerShareSupply = dist.accPerShareSupply;

        uint boostedSenderSupply = _calculateBoostedSupply(qToken, sender);
        uint boostedReceiverSupply = _calculateBoostedSupply(qToken, receiver);
        dist.totalBoostedSupply = dist
            .totalBoostedSupply
            .add(boostedSenderSupply)
            .add(boostedReceiverSupply)
            .sub(senderInfo.boostedSupply)
            .sub(receiverInfo.boostedSupply);
        senderInfo.boostedSupply = boostedSenderSupply;
        receiverInfo.boostedSupply = boostedReceiverSupply;
    }
```

Listing 3.4: `QDistributor::notifyTransferred()`

Recommendation Validate the given input to the `notifyTransferred()` routine by adding the following requirement, i.e., `require(sender!=receiver)`.

Status The issue has been fixed by this commit: `4f74ff7`.

3.5 Simplified Logic in removeMarket() for Gas Optimization

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: QoreAdmin
- Category: Coding Practices [8]
- CWE subcategory: CWE-1041 [1]

Description

As mentioned earlier, Qubit is inspired from Compound with shared components. In particular, the QoreAdmin contract plays the role of Comptroller in administrating the overall protocol operations, including the additions or removals of supported markets. While examining the related administration operations, we notice the current logic on market removal can be improved for gas efficiency.

To elaborate, we show below the related removeMarket(). The current implementation iterates each current market and copies all markets except the removed one to a new array, which can be optimized. In particular, we can simply locate the index of the given market for removal, next swap it with the last element in the supported market list, and then drop the element placed in the last position of the list. By doing so, we can reduce the gas cost for iterating all market elements in the list.

```
    function removeMarket(address payable qToken) external onlyKeeper {
        require(marketInfos[qToken].isListed, "Qore: unlisted market");
        require(IQToken(qToken).totalSupply() == 0 && IQToken(qToken).totalBorrow() ==
            0, "Qore: cannot remove market");

        address[] memory updatedMarkets = new address[](markets.length - 1);
        uint counter = 0;
        for (uint i = 0; i < markets.length; i++) {
            if (markets[i] != qToken) {
                updatedMarkets[counter++] = markets[i];
            }
        }
        markets = updatedMarkets;
        delete marketInfos[qToken];
    }
}
```

Listing 3.5: QoreAdmin::removeMarket()

Note that other routines share the same issue, including removeUserFromList() and _removeUserMarket ().

Recommendation Reduce unnecessary gas cost in the above routines.

Status The issue has been fixed by this commit: 7985015.

3.6 Redundant State/Code Removal

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [4]

Description

The `Qubit` protocol makes good use of a number of reference contracts, such as `ERC20`, `SafeBEP20`, `SafeMath`, and `Address`, to facilitate its code implementation and organization. For example, the `QoreAdmin` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `QubitToken` contract, it has defined a modifier `onlyMinter()`, which essentially enforces the following requirement: `require(msg.sender != address(0)&& isMinter(msg.sender))`. Note that the first part on `msg.sender != address(0)` is not necessary as it always evaluates to be true.

```
contract QubitToken is BEP20Upgradeable {
    /* ========== STATE VARIABLES ========== */

    mapping(address => bool) private _minters;

    /* ========== MODIFIERS ========== */

    modifier onlyMinter() {
        require(msg.sender != address(0) && isMinter(msg.sender), "QBT: caller is not
            the minter");
        _;
    }
    ...
    }
```

Listing 3.6: The `QubitToken` Contract

Recommendation Consider the removal of the redundant state (or code) with a simplified, consistent implementation.

Status The issue has been fixed by this commit: `330876c`.

3.7 Suggested Emit of Related Events Upon Settings Changes

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: QoreAdmin
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [4]

Description

In Ethereum, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the QoreAdmin contract as an example. While examining the events that reflect the protocol dynamics, we notice the related events are not emitted when various important parameters are updated. To elaborate, we show below related three routines, i.e., setKeeper(), setQValidator(), and setQDistributor(). The changes to respective states keeper, qValidator, and qDistributor are sensitive, which brings the need to emit meaningful events to reflect their changes.

```
    function setKeeper(address _keeper) external onlyKeeper {
        require(_keeper != address(0), "Qore: invalid keeper address");
        keeper = _keeper;
    }

    function setQValidator(address _qValidator) external onlyKeeper {
        require(_qValidator != address(0), "Qore: invalid qValidator address");
        qValidator = _qValidator;
    }

    function setQDistributor(address _qDistributor) external onlyKeeper {
        require(_qDistributor != address(0), "Qore: invalid qDistributor address");
        qDistributor = IQDistributor(_qDistributor);
    }
```

Listing 3.7: QoreAdmin::setKeeper()/setQValidator()/setQDistributor()

Recommendation Properly emit the related events in all updates on sensitive states or configurations. They are very helpful for external analytics and reporting tools.

Status The issue has been fixed by this commit: 501b2a4.

3.8 Trust Issue Of Admin Keys

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [3]

Description

In the `Qubit` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., performing sensitive operations and configuring system parameters). In the following, we show the representative functions potentially affected by the privilege of the `owner` account.

```
    function setQore(address _qore) public onlyOwner {
        require(_qore != address(0), "QMarket: invalid qore address");
        qore = IQore(_qore);
    }

    function setUnderlying(address _underlying) public onlyOwner {
        require(_underlying != address(0), "QMarket: invalid underlying address");
        require(underlying == address(0), "QMarket: set underlying already");
        underlying = _underlying;
    }

    function setRateModel(address _rateModel) public accrue onlyOwner {
        require(_rateModel != address(0), "QMarket: invalid rate model address");
        rateModel = IRateModel(_rateModel);
    }

    function setReserveFactor(uint _reserveFactor) public accrue onlyOwner {
        require(_reserveFactor <= RESERVE_FACTOR_MAX, "QMarket: invalid reserve factor");
        reserveFactor = _reserveFactor;
    }
```

Listing 3.8: A number of representative `setters` in `QMarket`

Using the `QubitPresale` contract as an example, the privileged function `sweep()` may be exercised by `owner` to collect available funds in the `QubitPresale` contract.

```
    function sweep(uint _lpAmount, uint _offerAmount) public override onlyOwner {
        require(_lpAmount <= IBEP20(BUNNY_WBNB_LP).balanceOf(address(this)), "QubitPresale: not enough token 0");
        require(_offerAmount <= IBEP20(QBT).balanceOf(address(this)), "QubitPresale: not enough token 1");
        BUNNY_WBNB_LP.safeTransfer(msg.sender, _lpAmount);
        QBT.safeTransfer(msg.sender, _offerAmount);
```

```
    }
```

Listing 3.9: `QubitPresale::sweep()`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the `owner` is not governed by a `DAO`-like structure. Note that a compromised `owner` account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the `MOAR` design.

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 The team confirms that a multisig account will be introduced before mainnet launch.

3.9 Unsound PriceCalculator Assumption On Asset Decimals

- ID: PVE-009
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `PriceCalculatorBSC`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The `Qubit` protocol has an important oracle-related contract named `PriceCalculatorBSC`. This contract is essential to query for the price of the underlying assets behind supported `QTokens`. In particular, it has a main function `getUnderlyingPrice()` that is used to obtain the price of the given asset and has been used in various scenarios, including the calculation of collateralized assets in USD for health check etc.

To elaborate, we show below the related `getUnderlyingPrice()` function. It implements a rather straightforward logic in firstly querying the effective price fee, then checking the valid hardcoded price, and finally converting the resulting amount with the `BNB` denomination. However, it comes to our attention that the computation makes an implicit assumption of the asset decimal, i.e., `18`. Unfortunately, this assumption may not always hold! When violated, it may be of serious detriment to the overall protocol operations, including the collateralization evaluation and allowed borrow amount calculation.

```
    function priceOf(address asset) public view override returns (uint priceInUSD) {
        (, priceInUSD) = _oracleValueOf(asset, 1e18);
        return priceInUSD;
```

```
    }
    ...
    function getUnderlyingPrice(address qToken) public view override returns (uint) {
        return priceOf(IQToken(qToken).underlying());
    }
    ...
    function _oracleValueOf(address asset, uint amount) private view returns (uint
        valueInBNB, uint valueInUSD) {
        valueInUSD = 0;
        if (tokenFeeds[asset] != address(0)) {
            (, int price, , , ) = AggregatorV3Interface(tokenFeeds[asset]).
                latestRoundData();
            valueInUSD = uint(price).mul(1e10).mul(amount).div(1e18);
        } else if (references[asset].lastUpdated > block.timestamp.sub(1 days)) {
            valueInUSD = references[asset].lastData.mul(amount).div(1e18);
        }
        valueInBNB = valueInUSD.mul(1e18).div(priceOfBNB());
    }
```

Listing 3.10: `PriceCalculatorBSC::getUnderlyingPrice()`

Recommendation Revise the above `getUnderlyingPrice()` routine to properly take into account the asset decimal.

Status The issue has been fixed by this commit: `f37f63c`.

3.10 Proper accruedQubit() Calculation

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

Description

As mentioned in Section 3.4, `Qubit` has a built-in incentive mechanism to reward protocol users upon a variety of protocol operations. Each user can also use the public function `accruedQubit()` to query for current accrued rewards.

To elaborate, we show below the `accruedQubit()` function. It computes the available rewards ready for claim. However, our analysis shows that this routine does not necessarily return the full rewards credited to the user.

```
    function accruedQubit(address market, address user) external view override returns (
        uint) {
        DistributionInfo storage dist = distributions[market];
```

```
134         UserInfo storage userInfo = marketUsers[market][user];
135 
136         uint _accruedQubit = userInfo.accruedQubit;
137         uint accPerShareSupply = dist.accPerShareSupply;
138         uint accPerShareBorrow = dist.accPerShareBorrow;
139 
140         uint timeElapsed = block.timestamp > dist.accruedAt ? block.timestamp.sub(dist.
            accruedAt) : 0;
141         if (timeElapsed > 0) {
142             if (dist.totalBoostedSupply > 0) {
143                 accPerShareSupply = accPerShareSupply.add(
144                     dist.supplyRate.mul(timeElapsed).mul(1e18).div(dist.
                        totalBoostedSupply)
145                 );
146 
147                 uint pendingQubit = userInfo.boostedSupply.mul(accPerShareSupply.sub(
                    userInfo.accPerShareSupply)).div(
148                     1e18
149                 );
150                 _accruedQubit = _accruedQubit.add(pendingQubit);
151             }
152 
153             if (dist.totalBoostedBorrow > 0) {
154                 accPerShareBorrow = accPerShareBorrow.add(
155                     dist.borrowRate.mul(timeElapsed).mul(1e18).div(dist.
                        totalBoostedBorrow)
156                 );
157 
158                 uint pendingQubit = userInfo.boostedBorrow.mul(accPerShareBorrow.sub(
                    userInfo.accPerShareBorrow)).div(
159                     1e18
160                 );
161                 _accruedQubit = _accruedQubit.add(pendingQubit);
162             }
163         }
164         return _accruedQubit;
165     }
```

Listing 3.11: `QDistributor::accruedQubit()`

Specifically, the internal `if`-loop (line 141) is conditioned on `timeElapsed > 0)`, which does not take into account two other scenarios: `accPerShareSupply != userInfo.accPerShareSupply` and `accPerShareBorrow != userInfo.accPerShareBorrow`. As a result, the current computed amount is likely less than the due reward amount.

Recommendation Revise the above function to accommodate the missed scenarios by revising the internal `if`-loop to be the following: `if (timeElapsed > 0|| (accPerShareSupply != userInfo.accPerShareSupply)|| (accPerShareBorrow != userInfo.accPerShareBorrow))`.

Status The issue has been fixed by this commit: `4f74ff7`.

4 | Conclusion

In this audit, we have analyzed the `Qubit` design and implementation. The system presents a unique, robust DeFi lending protocol that allows users to freely deposit virtual assets as collateral in order to borrow virtual assets. 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/1041.html.

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

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

[4] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

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

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

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

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

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

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

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

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