SMART CONTRACT AUDIT REPORT

for

PancakeBunny

Prepared By: Yiqun Chen

PeckShield

September 11, 2021

Document Properties

Client	PancakeBunny
Title	Smart Contract Audit Report
Target	PancakeBunny
Version	1.0
Author	Xuxian Jiang
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	September 11, 2021	Xuxian Jiang	Final Release
1.0-rc	September 9, 2021	Xuxian Jiang	Release Candidate

Contact

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

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

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the PancakeBunny 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 PancakeBunny

PancakeBunny is a yield aggregator on Binance Smart Chain (BSC), which allows users to earn higher interest on their underlying crypto assets through its advanced yield optimization strategies. This audit focuses on the new support of the Qubit-based vaults as well as BunnyFeeBox, which enrich the PancakeBunny ecosystem and also presents a unique contribution to current DeFi ecosystem.

The basic information of PancakeBunny is as follows:

Table 1.1: Basic Information of PancakeBunny

Item	Description
Name	PancakeBunny
Website	https://pancakebunny.finance/
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 11, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the audited repository contains a number of sub-directories and this audit focuses on the contracts/vaults/qubit sub-directory, two other vaults (BunnyPoolV2 and VaultBunnyMaximizer), as well as the `BunnyFeeBox` smart contract.

- https://github.com/PancakeBunny-finance/Bunny.git (64eafdc)

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/Bunny.git (0fb99ee)

1.2 About PeckShield

PeckShield Inc. [16] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [15]:

- 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 List of Check Items

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

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

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

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

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

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

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

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

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [14], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

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

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

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

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the `PancakeBunny` implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business 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	1	■
Undetermined	0	
Total	9	

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities, 6 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key PancakeBunny Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Sanity Checks Of System/-Function Parameters	Coding Practices	Fixed
PVE-002	Low	Proper Allowance Reset of Old Minter	Coding Practices	Fixed
PVE-003	Low	Possible Sandwich/MEV Attacks For Reduced Returns	Time and State	Resolved
PVE-004	Informational	Redundant State/Code Removal	Coding Practices	Fixed
PVE-005	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated
PVE-006	Medium	Proper Migration of Both Market Token and vToken	Business Logic	Resolved
PVE-007	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Fixed
PVE-008	Low	Suggested nonReentrant For VaultQubit::withdrawAll()	Time and State	Confirmed
PVE-009	Low	Potential Overflow In VaultCompensa-tion	Numeric Errors	Resolved

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 Improved Sanity Checks For System/Function Parameters

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `PancakeBunny` protocol is no exception. Specifically, if we examine the `BankConfig` contract, they have defined a number of protocol-wide risk parameters, e.g., `getReservePoolBps`, and `interestModel`. In the following, we show an example routine that allows for their changes.

```
    /// @dev Set all the basic parameters. Must only be called by the owner.
    /// @param _reservePoolBps The new interests allocated to the reserve pool value.
    /// @param _interestModel The new interest rate model contract.
    function setParams(uint _reservePoolBps, InterestModel _interestModel) public
        onlyOwner {
        getReservePoolBps = _reservePoolBps;
        interestModel = _interestModel;
    }
```

Listing 3.1: An example`setter` in `BankConfig`

Our result shows the update logic on the above parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of a large `getReservePoolBps` parameter may accrue unreasonable high interest for every borrow position.

In addition, the `BunnyPoolV2` contract has a public function `notifyRewardAmounts()` that is designed to accept new rewards for distribution. This function can be improved by enforcing the given argument has the same length with the `_rewardTokens` array (line 185).

```
    function notifyRewardAmounts(uint[] memory amounts) external override
        onlyRewardsDistribution updateRewards(address(0)) {
        for (uint i = 0; i < _rewardTokens.length; i++) {
            RewardInfo storage rewardInfo = rewards[_rewardTokens[i]];
            if (block.timestamp >= periodFinish) {
                rewardInfo.rewardRate = amounts[i].div(rewardsDuration);
            } else {
                uint remaining = periodFinish.sub(block.timestamp);
                uint leftover = remaining.mul(rewardInfo.rewardRate);
                rewardInfo.rewardRate = amounts[i].add(leftover).div(rewardsDuration);
            }
            rewardInfo.lastUpdateTime = block.timestamp;
            ...
        }

        periodFinish = block.timestamp.add(rewardsDuration);
        emit RewardsAdded(amounts);
    }
```

Listing 3.2: `BunnyPoolV2::notifyRewardAmounts()`

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. Also, consider emitting related events for external monitoring and analytics tools.

Status The issue has been fixed by this commit: `3cecb51`.

3.2 Proper Allowance Reset of Old Minter

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1126 [1]

Description

The `PancakeBunny` protocol has a protocol-wide `QubitPool` contract that can be used to configure various aspects of the `bQBT` pool. And this contract has a permissioned function `setQubitBridge()` that allows for the reconfiguration of a new `VaultQubitBridge` address.

To elaborate, we show below the `setQubitBridge()` routine. This routine not only sets up the new `VaultQubitBridge`, but also permits the new `VaultQubitBridge` contract to transfer the `QBT` funds in the `bQBT` pool. We notice there is also a need to reset the spending allowance of the old `VaultQubitBridge` back to 0.

```
    function setQubitBridge(address newBridge) public onlyOwner {
        require(newBridge != address(0), "QubitPool: bridge must be non-zero address");
        if (IBEP20(QBT).allowance(address(this), newBridge) == 0) {
            QBT.safeApprove(newBridge, uint(-1));
        }
        qubitBridge = IVaultQubitBridge(newBridge);
    }
```

Listing 3.3: QubitPool::setQubitBridge()

Note the `setMinter()` function in other contracts `VaultController` and `VaultBunnyBNB` also shares the same issue.

Recommendation Reset the allowance of the old `VaultQubitBridge` or `Minter`, if any, back to 0

Status The issue has been fixed by this commit: `723f54f`.

3.3 Possible Sandwich/MEV Attacks For Reduced Returns

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BunnyPool`
- Category: Time and State [12]
- CWE subcategory: CWE-682 [6]

Description

As a popular yield aggregator protocol, the `PancakeBunny` protocol has the constant need of swapping one token to another. For example, the computed staking rewards may need to convert into `WBNB` as the final payout. Our analysis shows this mechanism can be improved to avoid unnecessary reward loss.

To elaborate, we show below the related `getReward()` routine. As the name indicates, the function computes and transfers the intended rewards to a pool user. Note that the token conversion essentially performs the swap from `stakingToken` to `WBNB` (line 212).

```
    function getReward() override public nonReentrant updateReward(msg.sender) {
        uint256 reward = rewards[msg.sender];
        if (reward > 0) {
            rewards[msg.sender] = 0;
            reward = _flipToWBNB(reward);
            IBEP20(ROUTER_V1_DEPRECATED.WETH()).safeTransfer(msg.sender, reward);
            emit RewardPaid(msg.sender, reward);
        }
    }

    function _flipToWBNB(uint amount) private returns(uint reward) {
```

```
        address wbnb = ROUTER_V1_DEPRECATED.WETH();
        (uint rewardBunny,) = ROUTER_V1_DEPRECATED.removeLiquidity(
            address(stakingToken), wbnb,
            amount, 0, 0, address(this), block.timestamp);
        address[] memory path = new address[](2);
        path[0] = address(stakingToken);
        path[1] = wbnb;
        ROUTER_V1_DEPRECATED.swapExactTokensForTokens(rewardBunny, 0, path, address(this
            ), block.timestamp);

        reward = IBEP20(wbnb).balanceOf(address(this));
    }
```

Listing 3.4: BunnyPool::getReward()

We notice the conversion is routed to the external `UniswapV2` without any slippage control. With that, it is possible for a malicious actor to launch a flashloan-assisted attack to claim the majority of swaps, resulting in a significantly less amount after the swap. This is possible if the `getReward()` function suffers from a sandwich attack.

Recommendation Develop an effective mitigation to the above sandwich attack to better protect the interests of trading users.

Status The issue has been resolved as the related `BunnyPool` is now deprecated.

3.4 Redundant State/Code Removal

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contract
- Category: Coding Practices [9]
- CWE subcategory: CWE-563 [4]

Description

The `PancakeBunny` protocol makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `SafeMath`, and `ReentrancyGuard`, to facilitate its code implementation and organization. For example, the `VaultQBTBNB` 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 state variables defined in the `VaultQBTBNB` contract, one of them is not used: `rewardTotalSupply`. The unused state can be safely removed.

```
contract VaultQBTBNB is IPresaleLocker, RewardsDistributionRecipientUpgradeable,
    ReentrancyGuardUpgradeable, PausableUpgradeable {
```

```
    ...
    mapping(address => uint256) private _presaleBalances;
    uint256 public presaleEndTime; //1626652800 2021-07-19 00:00:00 UTC
    address public presaleContract;
    uint256 public rewardTotalSupply;
    ...
}
```

Listing 3.5: The `VaultQBTBNB` Contract

Also, the `BunnyPoolV2` contract defines a redundant `Recovered` event, which is duplicated from the inherited `VaultController` contract.

Recommendation Consider the removal of the redundant code with a simplified, consistent implementation.

Status The issue has been fixed by this commit: `5fb1dc2`.

3.5 Trust Issue Of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [8]
- CWE subcategory: CWE-287 [3]

Description

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

```
    function migrate() external onlyOwner {
        require(_bunnyPool != address(0), "VaultBunnyMaximizer: must set BunnyPool");
        uint before = IBEP20(BUNNY).balanceOf(address(this));
        IBunnyPool(BUNNY_POOL_V1).withdrawAll();    // get BUNNY, WBNB

        zap.zapInToken(WBNB, IBEP20(WBNB).balanceOf(address(this)), BUNNY);
        IBunnyPool(_bunnyPool).deposit(IBEP20(BUNNY).balanceOf(address(this)).sub(before
            ));
    }
```

Listing 3.6: `VaultBunnyMaximizer::migrate()`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a `DAO`-like structure.

Note that a compromised `owner` would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the `PancakeBunny` protocol.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed by the team. The team clarifies a plan to move the owner to a multi-sig account during the next week.

3.6 Proper Migration of Both Market Token and vToken

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

Description

The `PancakeBunny` protocol supports a unique `VaultVenus` vault, which is equipped with a helper contract `VaultVenusBridge` to manage the yielding funds in `Venus`. Within the `VaultVenusBridge` helper contract, we notice the supported migration feature needs to be improved.

To elaborate, we show below the `migrateTo()` function, which is designed to migrate the assets to another `VaultVenusBridge`. While it indeed properly migrates the `available` amount of `token`, the current implementation does not migrate the `vTokenAmount` of the corresponding `vToken`.

```
    function migrateTo(address payable target) external override onlyWhitelisted {
        MarketInfo storage market = markets[msg.sender];
        IVaultVenusBridge newBridge = IVaultVenusBridge(target);

        if (market.token == WBNB) {
            newBridge.deposit{value : market.available}(msg.sender, market.available);
        } else {
            IBEP20 token = IBEP20(market.token);
            token.safeApprove(address(newBridge), uint(-1));
            token.safeTransfer(address(newBridge), market.available);
            token.safeApprove(address(newBridge), 0);
            newBridge.deposit(msg.sender, market.available);
        }
        market.available = 0;
        market.vTokenAmount = 0;
```

```
    }
```

Listing 3.7: VaultVenusBridge::migrateTo()

Recommendation Revise the above `migrateTo()` routine to properly migrate both `token` and `vToken`.

Status The issue has been resolved as the related `VaultVenus` will be deprecated after the deployment of `QubitVault`.

3.7 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `VaultVenus`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

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 3.8: 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 `decreaseCollateral()` routine in the `VaultVenus` contract. If the `USDT` token is supported as `_stakingToken`, the unsafe version of `_stakingToken.transferFrom(owner(), address(venusBridge), migrationCost)` (line 309) 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)!

```
288     function decreaseCollateral(uint amountMin, uint supply) external payable onlyKeeper {
289         updateVenusFactors();
290 
291         uint _balanceBefore = balance();
292 
293         supply = msg.value > 0 ? msg.value : supply;
294         if (address(_stakingToken) == WBNB) {
295             venusBridge.deposit{value : supply}(address(this), supply);
296         } else {
297             _stakingToken.safeTransferFrom(msg.sender, address(venusBridge), supply);
298             venusBridge.deposit(address(this), supply);
299         }
300 
301         venusBridge.mint(balanceAvailable());
302         _decreaseCollateral(amountMin);
303         venusBridge.withdraw(msg.sender, supply);
304 
305         updateVenusFactors();
306         uint _balanceAfter = balance();
307         if (_balanceAfter < _balanceBefore && address(_stakingToken) != WBNB) {
308             uint migrationCost = _balanceBefore.sub(_balanceAfter);
309             _stakingToken.transferFrom(owner(), address(venusBridge), migrationCost);
310             venusBridge.deposit(address(this), migrationCost);
311         }
```

```
312     }
```

Listing 3.9: `VaultVenus::decreaseCollateral()`

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

Status The issue has been fixed by this commit: `0fb99ee`.

3.8 Suggested nonReentrant For VaultQubit::withdrawAll()

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `VaultQubit`
- Category: Time and State [11]
- CWE subcategory: CWE-663 [5]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [18] exploit, and the recent `Uniswap/Lendf.Me` hack [17].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `VaultQubit` as an example, the `withdrawAll()` 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`.

Apparently, the interaction with the external contract (line 312) starts before effecting the update on internal states, 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 very same `withdrawAll()` function.

```
305     function withdrawAll() external updateReward(msg.sender) {
306         updateQubitFactors();
307         uint amount = balanceOf(msg.sender);
308         uint depositTimestamp = _depositedAt[msg.sender];
309         uint available = balanceAvailable();
310
311         if (available < amount) {
```

```
            _decreaseCollateral(amount);
            available = balanceAvailable();
        }
        // revert if insufficient liquidity
        require(amount <= available, "VaultQubit: insufficient available");

        totalShares = totalShares.sub(_shares[msg.sender]);
        delete _shares[msg.sender];
        delete _principal[msg.sender];
        delete _depositedAt[msg.sender];

        uint withdrawalFee = canMint() ? _minter.withdrawalFee(amount, depositTimestamp)
             : 0;
        if (withdrawalFee > DUST) {
            qubitBridge.withdraw(withdrawalFee, address(this));
            if (address(_stakingToken) == WBNB) {
                _minter.mintForV2{ value: withdrawalFee }(address(0), withdrawalFee, 0,
                    msg.sender, depositTimestamp);
            } else {
                _minter.mintForV2(address(_stakingToken), withdrawalFee, 0, msg.sender,
                    depositTimestamp);
            }
            amount = amount.sub(withdrawalFee);
        }
        qubitBridge.withdraw(amount, msg.sender);
        getReward();

        if (collateralRatio >= collateralRatioLimit) {
            _decreaseCollateral(0);
        }
        emit Withdrawn(msg.sender, amount, withdrawalFee);
    }
```

Listing 3.10: `VaultQubit::withdrawAll()`

Recommendation Apply necessary `re-entrancy` prevention by adding necessary `nonReentrant` modifier to the above `withdrawAll()` function.

Status This issue has been confirmed. And the team clarifies that all code has followed the recommended checks-effects-interactions pattern.

3.9 Potential Overflow In VaultCompensation

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `VaultCompensation`
- Category: Numeric Errors [13]
- CWE subcategory: CWE-190 [2]

Description

`SafeMath` is a Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, we find that it is not widely used in `VaultCompensation` contract.

In particular, while examining the logic of the `VaultCompensation` contract, we notice that there is a `depositOnBehalfBulk()` function that is designed to deposit a number of requests each with the intended recipient and the amount. However, it comes to our attention that the `sum` calculation `sum += request[i].amount` (line 211) is not overflow-protected, which may introduce unexpected behavior. Fortunately, it is a privileged function which can only be called by a trusted entity. With that, we suggest to use `SafeMath` to avoid unexpected overflows or underflows.

```
208     function depositOnBehalfBulk(DepositRequest[] memory request) external onlyOwner {
209         uint sum;
210         for (uint i = 0; i < request.length; i++) {
211             sum += request[i].amount;
212         }
213 
214         _totalSupply = _totalSupply.add(sum);
215         IBEP20(stakingToken).safeTransferFrom(msg.sender, address(this), sum);
216 
217         for (uint i = 0; i < request.length; i++) {
218             address to = request[i].to;
219             uint amount = request[i].amount;
220             _balances[to] = _balances[to].add(amount);
221             emit Deposited(to, amount);
222         }
223     }
```

Listing 3.11: `VaultCompensation::depositOnBehalfBulk()`

Recommendation Use `SafeMath` in the above `depositOnBehalfBulk()` function to avoid unexpected overflows or underflows.

Status This issue has been confirmed. And the team clarifies that it is a legacy contract which is not longer used.

4 Conclusion

In this audit, we have analyzed the `PancakeBunny` design and implementation. `PancakeBunny` is a yield aggregator on `Binance Smart Chain` (`BSC`), which allows users to earn higher interest on their underlying crypto assets through its advanced yield optimization strategies. This audit focuses on the new support of the `Qubit`-based vaults as well as `BunnyFeeBox`. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

References

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

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.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-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

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

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

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

[12] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[13] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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

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

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