SMART CONTRACT AUDIT REPORT

for

RABBIT FINANCE

Prepared By: Yiqun Chen

PeckShield

July 28, 2021

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	July 28, 2021	Xuxian Jiang	Final Release
1.0-rc	July 28, 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 source code of the the `Rabbit Finance 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 Rabbit Finance

`Rabbit Finance` is a leveraged trading protocol based on deposit and borrowing functions. The key features of `Rabbit Finance` include deposit and borrowing, leveraged yield farming and leveraged trading, and will support options, synthetic assets, and NFT trading functions in the future. It is designed as an evolutional improvement of earlier offerings (e.g., `Alpaca` and `Alpha Homora`) with the goal of continuously improving the utilization of deposit users' funds. New application scenarios of borrowing and leverage farming are continuously discovered and explored. The audited implementation makes improvements, including the support of additional workers (e.g., `MdexWorkerGobin`) and strategies (e.g., `espAddStrategy`).

The basic information of Rabbit Finance is as follows:

Table 1.1: Basic Information of Rabbit Finance

Item	Description
Issuer	Rabbit Finance
Website	https://rabbitfinance.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 28, 2021

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

- https://github.com/RabbitFinanceProtocol/rabbit_finance_bsc.git (5f711ce)

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

- https://github.com/RabbitFinanceProtocol/rabbit_finance_bsc.git (ef036cf)

1.2 About PeckShield

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

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

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

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

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

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

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

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

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

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

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

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

1.4 Disclaimer

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

Table 1.3: 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

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 implementation of the `Rabbit` 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 logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	4	■■■■
Low	5	■■■■■
Informational	0	
Total	10	

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 4 medium-severity vulnerabilities, and 5 low-severity vulnerabilities.

Table 2.1: Key Audit Findings of Rabbit Finance Protocol

ID	Severity	Title	Category	Status
PVE-001	High	Force Investment Risk in Bank	Business Logic	Fixed
PVE-002	Low	Use Of Proper Prize Recipient in kill()	Business Logic	Confirmed
PVE-003	Medium	Proper totalReserve Accounting in calInterest()	Business Logic	Fixed
PVE-004	Medium	Timely massUpdatePools During Pool Updates	Business Logic	Confirmed
PVE-005	Medium	Trust Issue of Admin Keys	Business Logic	Mitigated
PVE-006	Medium	Trading Fee Discrepancy In PancakeGoblin And EspAddStrategy	Business Logic	Fixed
PVE-007	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Coding Practices	Fixed
PVE-008	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Fixed
PVE-009	Low	Improved Sanity Checks For System Parameters	Coding Practices	Confirmed
PVE-010	Low	Possible Sandwich/MEV Attacks For Reduced Returns	Time and State	Confirmed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Force Investment Risk in Bank

- ID: PVE-001
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: Bank
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The Rabbit protocol is a leveraged trading protocol based on deposit and borrowing functions. It is inspired from the Alpha/Alpaca framework and thus shares similar architecture with vault, worker, and strategies. While examining the vault implementation (inside the Bank contract), we notice a potential force investment risk that has been exploited in earlier hacks, e.g., yDAI [15] and BT.Finance [1]. To elaborate, we show below the related Bank::work() routine.

Specifically, the Bank contract is designed and implemented to invest borrowed funds (held in Bank), harvest growing yields, and return any gains, if any, to the users. For safety, the protocol requires each position is subject to the health check (lines 1001-1002) for each borrow-related operation.

```
945     function work(uint256 posId, uint256 pid, uint256 borrow, bytes calldata data)
946     external payable onlyEOA nonReentrant {
947 
948         if (posId == 0) {
949             posId = currentPos;
950             currentPos ++;
951             positions[posId].owner = msg.sender;
952             positions[posId].productionId = pid;
953             positions[posId].debtShare = 0;
954 
955             userPosition[msg.sender].push(posId);
956         } else {
957             require(posId < currentPos, "bad position id");
958             require(positions[posId].owner == msg.sender, "not position owner");
959             pid = positions[posId].productionId;
```

```
        }

        Production storage production = productions[pid];
        require(production.isOpen, 'Production not exists');
        require(borrow == 0  production.canBorrow, "Production can not borrow");

        calInterest(production.borrowToken);

        uint256 debt = _removeDebt(posId, production).add(borrow);
        bool isBorrowBNB = production.borrowToken == address(0);

        uint256 sendBNB = msg.value;
        uint256 beforeToken = 0;
        if (isBorrowBNB) {
            sendBNB = sendBNB.add(borrow);
            require(sendBNB <= address(this).balance && debt <= banks[production.
                borrowToken].totalVal, "insufficient BNB in the bank");
            beforeToken = address(this).balance.sub(sendBNB);
        } else {
            beforeToken = SafeToken.myBalance(production.borrowToken);
            require(borrow <= beforeToken && debt <= banks[production.borrowToken].
                totalVal, "insufficient borrowToken in the bank");
            beforeToken = beforeToken.sub(borrow);
            SafeToken.safeApprove(production.borrowToken, production.goblin, borrow);
        }

        Goblin(production.goblin).work{value:sendBNB}(posId, msg.sender, production.
            borrowToken, borrow, debt, data);

        uint256 backToken = isBorrowBNB? (address(this).balance.sub(beforeToken)) :
            SafeToken.myBalance(production.borrowToken).sub(beforeToken);

        if(backToken > debt) { //
            backToken = backToken.sub(debt);
            debt = 0;

            isBorrowBNB? SafeToken.safeTransferETH(msg.sender, backToken):
                SafeToken.safeTransfer(production.borrowToken, msg.sender, backToken);

        }else if (debt > backToken) { //
            debt = debt.sub(backToken);
            backToken = 0;

            require(debt >= production.minDebt && debt <= production.maxDebt, "Debt
                scale is out of scope");
            uint256 health = Goblin(production.goblin).health(posId, production.
                borrowToken);
            require(health.mul(production.openFactor) >= debt.mul(GLO_VAL), "bad work
                factor");

            _addDebt(posId, production, debt);
        }
```

```
        emit Work(posId, debt, backToken);
    }
```

Listing 3.1: `Bank::work()`

It comes to our attention that the health check does not have the stability check on the liquidity pool into which the borrowed funds will be added. In other words, if the configured strategy blindly invests the deposited funds into an imbalanced `Ellipse/PancakeSwap` pool, the strategy will not result in a profitable investment. In fact, earlier incidents (`yDAI` and `BT.Finance` hacks [1, 15]) have prompted the need of a guarded call before kicking off the actual investment. For the very same reason, we argue for the guarded stability check associated with every single `health()` call.

Recommendation Ensure the target liquidity pool is stable before the borrowed funds can be added into as liquidity.

Status This issue has been fixed by adding the suggested stability check as reflected in the following commit: `94d87ec`.

3.2 Use Of Proper Prize Recipient in kill()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Bank`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

At the core of `Rabbit` is the the `Bank` contract that manages current debt positions and mediates the access to various `workers`. A position may be liquidated if the supplying asset is below the underwater threshold (specified by the parameter `liquidateFactor`). Moreover, the protocol provides incentive in assigning a portion of liquidated funds or `prize` to the liquidator.

To elaborate, we show below the `kill()` function. As the name indicates, it attempts to "kill" the given position by liquidating it immediately if the `liquidateFactor` condition is met. However, our analysis shows that the collected `prize` is returned back to the `devAddr`, instead of the liquidator. Note that the liquidation is not open to public. Instead, the governance will select those accounts that are authorized to perform liquidation operations.

```
    function kill(uint256 posId) external payable onlyEOA nonReentrant {
        require(killWhitelist[msg.sender],"Not Whitelist");

        Position storage pos = positions[posId];
```

```
        require(pos.debtShare > 0, "no debt");
        Production storage production = productions[pos.productionId];

        uint256 debt = _removeDebt(posId, production);

        uint256 health = Goblin(production.goblin).health(posId, production.borrowToken)
            ;
        require(health.mul(production.liquidateFactor) < debt.mul(GLO_VAL), "can't
            liquidate");
        bool isBNB = production.borrowToken == address(0);
        uint256 before = isBNB? address(this).balance: SafeToken.myBalance(production.
            borrowToken);

        Goblin(production.goblin).liquidate(posId, pos.owner, production.borrowToken);

        uint256 back = isBNB? address(this).balance: SafeToken.myBalance(production.
            borrowToken);
        back = back.sub(before);

        uint256 prize = back.mul(config.getLiquidateBps()).div(GLO_VAL);
        uint256 rest = back.sub(prize);
        uint256 left = 0;

        if (prize > 0) {
            isBNB? SafeToken.safeTransferETH(devAddr, prize): SafeToken.safeTransfer(
                production.borrowToken, devAddr, prize);
        }
        if (rest > debt) {
            left = rest.sub(debt);
            isBNB? SafeToken.safeTransferETH(pos.owner, left): SafeToken.safeTransfer(
                production.borrowToken, pos.owner, left);
        } else {
            banks[production.borrowToken].totalVal = banks[production.borrowToken].
                totalVal.sub(debt).add(rest);
        }
        emit Kill(posId, msg.sender, prize, left);
    }
```

Listing 3.2: `Bank::kill()`

Note that the same issue is also present in a number of other `reinvest()` routines in various `Goblin`-based workers.

Recommendation Properly return the liquidation prize to the liquidator, instead of the `devAddr`.

Status This issue has been confirmed.

3.3 Proper totalReserve Accounting in callInterest() And withdrawReserve()

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Bank
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The Rabbit protocol has been designed to provide efficient position management with timely accounting of current debt as well as the actual value. By design, there is a need to timely collect interest accrued from the debt. While examining the interest collection, we notice the state totalReserve is not properly accounted for.

To elaborate, we show below the related functions calInterest() and withdrawReserve(). The first function is used to compute accrued interest from the current debt and the second function allows the authorized entity to withdraw the reserve funds represented as totalReserve. Note that the internal accounting does not consider totalReserve as part of totalVal (reflected in calInterest()), which indicates the withdrawn reserve in the second function should not be used to decrease totalVal. The current implementation does decrease totalVal with the withdrawn reserve affects adverse influence to the share calculation for supplying users.

```
function calInterest(address token) public {
    TokenBank storage bank = banks[token];
    require(bank.isOpen, 'token not exists');
    if (now > bank.lastInterestTime) {
        uint256 timePast = now.sub(bank.lastInterestTime);

        uint256 totalDebt = bank.totalDebt;
        uint256 totalBalance = totalToken(token);

        uint256 ratePerSec = config.getInterestRate(totalDebt, totalBalance);
        uint256 interest = ratePerSec.mul(timePast).mul(totalDebt).div(1e18);

        uint256 toReserve = interest.mul(config.getReserveBps()).div(GLO_VAL);

        bank.totalReserve = bank.totalReserve.add(toReserve);
        bank.totalDebt = bank.totalDebt.add(interest);
        bank.lastInterestTime = now;
    }
}

function withdrawReserve(address token, address to, uint256 value) external onlyGov
    nonReentrant {
    TokenBank storage bank = banks[token];
    require(bank.isOpen, 'token not exists');

```

```
        uint balance = token == address(0)? address(this).balance: SafeToken.myBalance(
            token);
        if(balance >= bank.totalVal.add(value)) {

        } else {
            bank.totalReserve = bank.totalReserve.sub(value);
            bank.totalVal = bank.totalVal.sub(value);
        }

        if (token == address(0)) {
            SafeToken.safeTransferETH(to, value);
        } else {
            SafeToken.safeTransfer(token, to, value);
        }
    }
```

Listing 3.3: Bank:: callInterest ()/withdrawReserve()

Recommendation Revise the aforementioned routines to better maintain the accurate `totalReserve` state

Status This issue has been fixed in this commit: `94d87ec`.

3.4 Timely massUpdatePools During Pool Updates

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `FairLaunch`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The `Rabbit` protocol provides incentive mechanisms that reward the staking of supported assets. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

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

```
  // Update the given pool's rabbit allocation point. Can only be called by the owner.
function setPool(
  uint256 _pid,
```

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

Listing 3.4: `FairLaunch::setPool()`

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

Recommendation Timely invoke `massUpdatePools()` when any pool's weight has been updated. In fact, the third parameter (`_withUpdate`) to the `set()` routine can be simply ignored or removed. Also, keep in mind that the current `FairLaunch` contract does not support deflationary tokens! A vetting process needs to be in place to ensure incompatible deflationary tokens will not be supported as the pool token for farming.

Status The issue has been confirmed.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [3]

Description

In the `Rabbit` protocol, all debt positions are managed by the `Bank` contract. And there is a privileged `governor` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and strategy 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.

To elaborate, we show below the `kill()` routine in the `Bank` contract. This routine allows anyone to liquidate the given position assuming it is underwater and available for liquidation. There is a key factor, i.e., `liquidateFactor`, that greatly affects the decision on whether the position can be liquidated (line 1019). Note that `liquidateFactor` is a risk parameter that can be dynamically configured by the privileged governor.

```
1009     function kill(uint256 posId) external payable onlyEOA nonReentrant {
1010         require(killWhitelist[msg.sender],"Not Whitelist");
1011 
1012         Position storage pos = positions[posId];
1013         require(pos.debtShare > 0, "no debt");
1014         Production storage production = productions[pos.productionId];
1015 
1016         uint256 debt = _removeDebt(posId, production);
1017 
1018         uint256 health = Goblin(production.goblin).health(posId, production.borrowToken);
1019         require(health.mul(production.liquidateFactor) < debt.mul(GLO_VAL), "can't liquidate");
1020         bool isBNB = production.borrowToken == address(0);
1021         uint256 before = isBNB? address(this).balance: SafeToken.myBalance(production.borrowToken);
1022 
1023         Goblin(production.goblin).liquidate(posId, pos.owner, production.borrowToken);
1024 
1025         uint256 back = isBNB? address(this).balance: SafeToken.myBalance(production.borrowToken);
1026         back = back.sub(before);
1027 
1028         uint256 prize = back.mul(config.getLiquidateBps()).div(GLO_VAL);
1029         uint256 rest = back.sub(prize);
1030         uint256 left = 0;
1031 
1032         if (prize > 0) {
1033             isBNB? SafeToken.safeTransferETH(devAddr, prize): SafeToken.safeTransfer(production.borrowToken, devAddr, prize);
1034         }
1035         if (rest > debt) {
1036             left = rest.sub(debt);
1037             isBNB? SafeToken.safeTransferETH(pos.owner, left): SafeToken.safeTransfer(production.borrowToken, pos.owner, left);
1038         } else {
1039             banks[production.borrowToken].totalVal = banks[production.borrowToken].totalVal.sub(debt).add(rest);
1040         }
1041         emit Kill(posId, msg.sender, prize, left);
1042     }
```

Listing 3.5: `Bank::kill()`

Also, if we examine the privileged function of `PancakeswapWorkerGoblin`, i.e., `setCriticalStrategies`

(), this routine allows the update of new strategies to work on a user's position. It has been highlighted that bad strategies can steal user funds. Note that this privileged function is guarded with onlyGov.

```
    /// @dev Update critical strategy smart contracts. EMERGENCY ONLY. Bad strategies
        can steal funds.
    /// @param _addStrat The new add strategy contract.
    /// @param _liqStrat The new liquidate strategy contract.
    function setCriticalStrategies(Strategy _addStrat, Strategy _liqStrat) external
        onlyGov {
        addStrat = _addStrat;
        liqStrat = _liqStrat;
    }
```

Listing 3.6: PancakeswapWorkerGoblin::setCriticalStrategies()

It is worrisome if the privileged governor account is a plain EOA account. The discussion with the team confirms that the governor account is currently managed by a timelock. A plan needs to be in place to migrate it under community governance. 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.

In the following, we make efforts to keep track of the current deployment of various contracts in Rabbit and the results are shown in Table 3.1. Note a number of contracts are deployed by taking a proxy-based approach where the proxy contract is deployed at the front-end while the logic contract contains the actual business logic implementation. Specifically, it takes a delegatecall-based proxy pattern so that each component is split into two contracts: a back-end logic contract (that holds the implementation) and a front-end proxy (that contains the data and uses delegatecall to interact with the logic contract). From the user's perspective, they interact with the proxy while the code is executed on the logic contract. Accordingly, the privileged admin account of these front-end proxies also needs to be trusted. Fortunately, as shown in the Table 3.1, the current deployment is eventually managed by the Timelock contract (deployed at 6d37951c6e711c220637107b511a58c64ddc3625).

A further examination of the Timelock parameters shows the pre-configured $86,400s$ delay, which is 24 hours. In other words, all privileged operations will go through 24-hour timelock, which greatly alleviates the centralized admin key concerns.

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, it will be mitigated by a 24-hour timelock to balance efficiency and timely adjustment. After the protocol becomes stable,

Table 3.1: Current Contract Deployment of `Rabbit` (as of 2021/07/18)

Contract	Address	Note	Owner/Admin/Governor
Deployer	0x200263e7c46c0866f34ef3eab31a796040a6c902	EOA	
TimelockAdmin	0x7cb5307485d93480db6daf1440c5d752ece3f4cd	EOA	
OracleOwner	0x7dfc3c12ddcef329675d05b29f7e63e63a863b22	EOA	
Timelock	6d37951c6e711c220637107b511a58c64ddc3625	Contract	TimelockAdmin
ProxyAdmin	0x62c6a4396e306a5628bfde974f33905954d48068	Contract	Timelock
Bank	0xc18907269640d11e2a91d7204f33c5115ce3419e	Proxy	Timelock/ProxyAdmin
Bank Impl	0xbeeb9d4ca070d34c014230bafdfb2ad44a110142	Contract	
BankConfig	0x931c031f86d6fea071dec760395fd2c28dc88e3d	Contract	Timelock/ProxyAdmin
FairLaunch	0x81c1e8a6f8eb226aa7458744c5e12fc338746571	Contract	Timelock
RABBIT	0x95a1199eba84ac5f19546519e287d43d2f0e1b41	ERC20 Tokens	
WBNB	0xbb4cdb9cbd36b01bd1cbaebf2de08d9173bc095c	ERC20 Tokens	
ibBNB	0x45b887d3569caca67e10662075241f972d337850	ERC20 Tokens	
MDEX LP Rabbit-BUSD	0x0025d20d85788c2cae2feb9c298bdafc93bf08ce	ERC20 Tokens	
GoblinPriceOracle	0xde81c3db43c6c462b691ff427781bbef5bd191c9	Contract	OracleOwner
StrategyAllTokenOnly	0xd39dbf13c0bb678da2f3e803b6f1bd8b99187cb5	Contract	Deployer
StrategyLiquidate	0x5085c49828b0b8e69bae99d96a8e0fcf0a033369	Contract	Deployer
StrategyAddTwoSidesOptimal	0x5085c49828b0b8e69bae99d96a8e0fcf0a033369	Contract	Deployer

it is expected to migrate to a multi-sig account, and eventually be managed by community proposals for decentralized governance.

3.6 Trading Fee Discrepancy In PancakeGoblin And EspAddStrategy

- ID: PVE-006
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

In the `Rabbit` protocol, a number of situations require the real-time swap of one token to another. For example, the `StrategyAddTwoSidesOptimal` strategy takes one underlying token and converts some portion of it to another underlying token so that their ratio matches the current swap price in the `PancakeSwap` pool. Note that in `PancakeSwap`, if you make a token swap or trade on the exchange, you will need to pay a 0.25% trading fee, which is broken down into two parts. The first part of 0.17% is returned to liquidity pools in the form of a fee reward for liquidity providers, the 0.03% is sent to the `PancakeSwap Treasury`, and the remaining 0.05% is used towards `CAKE` buyback and burn.

To elaborate, we show below the the `getMktSellAmount()` routine in `PancakeGoblin`. It is interesting to note that `PancakeGoblin` has implicitly assumed the trading fee is 0.3%, instead of 0.25%. The difference in the built-in trading fee with the actual `PancakeSwap` may skew the optimal allocation of assets in the developed strategies, including `StrategyAddTwoSidesOptimal`.

```
    /// @dev Return maximum output given the input amount and the status of Uniswap
        reserves.
    /// @param aIn The amount of asset to market sell.
    /// @param rIn the amount of asset in reserve for input.
    /// @param rOut The amount of asset in reserve for output.
    function getMktSellAmount(uint256 aIn, uint256 rIn, uint256 rOut) public pure
        returns (uint256) {
        if (aIn == 0) return 0;
        require(rIn > 0 && rOut > 0, "bad reserve values");
        uint256 aInWithFee = aIn.mul(997);
        uint256 numerator = aInWithFee.mul(rOut);
        uint256 denominator = rIn.mul(1000).add(aInWithFee);
        return numerator / denominator;
    }
```

Listing 3.7: PancakeGoblin::getMktSellAmount()

The same issue is also applicable to the `EspAddStrategy` on the use of the `Ellipsis` trading fee.

Recommendation Make the built-in trading fee consistent with the actual trading fee in `PancakeSwap` and `Ellipsis`.

Status This issue has been fixed in the following commit: `7dd3fcb`.

3.7 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Time and State [10]
- CWE subcategory: CWE-663 [4]

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

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `Boardroom` as an example, the `claimReward()` 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 906) starts before effecting the update on the internal state (line 909), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
903     function claimReward() public {
904         uint256 reward = earned(msg.sender);
905         if (reward > 0) {
906             cash.safeTransfer(msg.sender, reward);
907             emit RewardPaid(msg.sender, reward);
908         }
909         directors[msg.sender].lastSnapshotIndex = latestSnapshotIndex();
910     }
```

Listing 3.8: Boardroom::claimReward()

In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`. However, it is important to take precautions in making use of `nonReentrant` to block possible `re-entrancy`.

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible `re-entrancy`.

Status This issue has been fixed in the following commit: `ba7e693`.

3.8 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

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 3.9: 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 `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `migrate()` routine in the `Treasury` contract. If the `USDT` token is supported as `cash`, the unsafe version of `IERC20(cash).transfer(target, IERC20(cash).balanceOf(address(this)))` (line 1369) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)!

```
1363    function migrate(address target) public onlyOperator checkOperator {
1364        require(!migrated, 'Treasury: migrated');
1365
1366        // cash
1367        Operator(cash).transferOperator(target);
1368        Operator(cash).transferOwnership(target);
1369        IERC20(cash).transfer(target, IERC20(cash).balanceOf(address(this)));
1370
1371        migrated = true;
1372        emit Migration(target);
1373    }
```

Listing 3.10: `Treasury::migrate()`

Note this issue is also applicable to other routines, including `burnReward()` from the `Boardroom` contract.

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transfer()`, `transferFrom()`, and `approve()`.

Status This issue has been fixed in the following commit: `ba7e693`.

3.9 Improved Sanity Checks For System Parameters

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [2]

Description

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

```
    function setParams(uint256 _getReserveBps, uint256 _getLiquidateBps, InterestModel
        _interestModel) public onlyOwner {
        getReserveBps = _getReserveBps;
        getLiquidateBps = _getLiquidateBps;
        interestModel = _interestModel;
    }
```

Listing 3.11: `BankConfig::setParams()`

```
    function createProduction(
        uint256 pid,
        bool isOpen,
        bool canBorrow,
        address coinToken,
        address currencyToken,
        address borrowToken,
        address goblin,
        uint256 minDebt,
        uint256 maxDebt,
        uint256 openFactor,
        uint256 liquidateFactor
        ) external onlyGov {

        if(pid == 0){
            pid = currentPid;
            currentPid ++;
        } else {
            require(pid < currentPid, "bad production id");
        }

        Production storage production = productions[pid];
        production.isOpen = isOpen;
        production.canBorrow = canBorrow;
        production.coinToken = coinToken;
        production.currencyToken = currencyToken;
        production.borrowToken = borrowToken;
        production.goblin = goblin;

        production.minDebt = minDebt;
        production.maxDebt = maxDebt;
        production.openFactor = openFactor;
        production.liquidateFactor = liquidateFactor;
    }
```

Listing 3.12: `Bank::createProduction()`

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 `liquidateFactor` parameter will make every position liquidatable.

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 confirmed.

3.10 Possible Sandwich/MEV Attacks For Reduced Returns

- ID: PVE-010
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Time and State [11]
- CWE subcategory: CWE-682 [5]

Description

The `PancakeswapGoblin` contract has a public routine, i.e., `reinvest()`, that is designed to reinvest whatever this worker has earned back to staked LP tokens. It has a rather straightforward logic in computing the intended amount for conversion, next performing the actual swap via the `UniswapV2` router, and finally calling the `addStrat` to convert into LP tokens for `MasterChef` deposit.

```
    /// @dev Re-invest whatever this worker has earned back to staked LP tokens.
    function reinvest() override public onlyEOA nonReentrant {
        require(killWhitelist[msg.sender],"Not Whitelist");

        // 1. Withdraw all the rewards.
        masterChef.withdraw(pid, 0);
        uint reward = cake.balanceOf(address(this));
        if (reward == 0) return;
        // 2. Send the reward bounty to the caller.
        uint fee = reward.mul(feeBps) / 10000;
        cake.safeTransfer(devAddr,fee);

        // 3. Convert all the remaining rewards to BNB.
        if (baseToken != cake) {
            address[] memory path;
            if (baseToken == wbnb) {
              path = new address[](2);
              path[0] = address(cake);
              path[1] = address(wbnb);
            } else {
              path = new address[](3);
              path[0] = address(cake);
```

```
                path[1] = address(wbnb);
                path[2] = address(baseToken); // cake
            }
            router.swapExactTokensForTokens(reward.sub(fee), 0, path, address(this), now);
        }

        // 4. Use add BNB strategy to convert all BNB to LP tokens.
        baseToken.safeTransfer(address(addStrat),baseToken.myBalance());
        addStrat.execute(address(0),address(0), 0,0, abi.encode(baseToken, farmingToken, 0));
        // 5. Mint more LP tokens and stake them for more rewards.
        masterChef.deposit(pid, lpToken.balanceOf(address(this)));
        emit Reinvest(msg.sender, reward, 0);
    }
```

Listing 3.13: `PancakeswapGoblin::reinvest()`

To elaborate, we show above the `reinvest()` routine. We notice the token swap is routed to `router` and the actual swap operation `swapExactTokensForTokens()` essentially does not specify any effective restriction on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of yielding. Note that a number of other related `reinvest()` routines in different `Goblin`-based workers share the same issue.

We acknowledge that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above front-running attack to better protect the interests of farming users.

Status The issue has been confirmed by the teams.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Rabbit` protocol, which is a leveraged trading protocol based on deposit and borrowing functions. Current key features include deposit and borrowing, leveraged yield farming, as well as leveraged trading. The system continues the innovative design and makes it distinctive and valuable when compared with current lending/yield farming offerings. The current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, 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] BT Finance. BT.Finance Exploit Analysis Report. https://btfinance.medium.com/bt-finance-exploit-analysis-report-a0843cb03b28.

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

[5] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.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 CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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

[15] PeckShield. The yDAI Incident Analysis: Forced Investment. https://peckshield.medium.com/the-ydai-incident-analysis-forced-investment-2b8ac6058eb5.

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

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