SMART CONTRACT AUDIT REPORT

for

ROYALE FINANCE

Prepared By: Shuxiao Wang

PeckShield
April 1, 2021

Document Properties

Client	Royale Finance
Title	Smart Contract Audit Report
Target	Royale Finance
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Huaguo Shi, Jeff Liu
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 1, 2021	Xuxian Jiang	Final Release
1.0-rc2	March 30, 2021	Xuxian Jiang	Release Candidate #2
1.0-rc1	March 29, 2021	Xuxian Jiang	Release Candidate #1
0.3	March 26, 2021	Xuxian Jiang	Add More Findings #2
0.2	March 19, 2021	Xuxian Jiang	Add More Findings #1
0.1	March 15, 2021	Xuxian Jiang	Initial Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the **Royale 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 Royale Finance

`Royale Finance` is a decentralized protocol for directing optimized stablecoin liquidity pools toward an on-chain funding solution for `iGaming` products. By providing a decentralized liquidity network, `Royale` aims to lower the barriers of entry for the `iGaming` industry and foster a robust crypto-economic system that incentivizes both `iGaming` entrepreneurs and liquidity providers to become long-term ecosystem participants. It is argued that the countercyclical nature of the `Royale Finance` ecosystem should make it an attractive venue for the contribution of decentralized stablecoin liquidity.

The basic information of `Royale Finance` is as follows:

Table 1.1: Basic Information of Royale Finance

Item	Description
Issuer	Royale Finance
Website	https://royale-finance-docs.netlify.app/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 1, 2021

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

- https://github.com/RoyaleFinanceV1/royale-contracts.git (f357b1f)

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

- https://github.com/RoyaleFinanceV1/royale-contracts.git (4c648f6)

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

Likelihood

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.

PeckShield

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Royale 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	1	
Medium	4	
Low	4	
Informational	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 1 high-severity vulnerability, 4 medium-severity vulnerabilities, 4 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Royale Finance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Safe-Version Replacement With safeApprove(), safeTransfer() And safeTransferFrom()	Coding Practices	Fixed
PVE-002	Low	Potential OOG Execution of RoyaleLP::updateWithdrawQueue()	Coding Practices	Fixed
PVE-003	Informational	Two-Step Transfer Of Privileged Account Ownership	Security Features	Fixed
PVE-004	Medium	Potential Underflow For Unlimited Discounted Amounts	Numeric Errors	Fixed
PVE-005	Low	Simplification of RoyaleLP::requestWithdrawWithRPT()	Coding Practices	Fixed
PVE-006	Medium	Non-Permissioned ERC1155Receiver::setShouldReject()	Security Features	Fixed
PVE-007	Low	Possible Friction That May Block User Withdrawals	Business Logic	Fixed
PVE-008	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-009	Medium	Possible CooldownTimestamp Manipulation	Business Logic	Fixed
PVE-010	High	Possible Sandwich/MEV For Reduced Return And Skewed Withdrawals	Time And State	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 Safe-Version Replacement With safeApprove(), safeTransfer() And safeTransferFrom()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `rStrategy`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [2]

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 this section, we examine the `approve()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
/**
* @dev Approve the passed address to spend the specified amount of tokens on behalf
    of msg.sender.
* @param _spender The address which will spend the funds.
* @param _value The amount of tokens to be spent.
*/
function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

    // To change the approve amount you first have to reduce the addresses`
    //  allowance to zero by calling `approve(_spender, 0)` if it is not
    //  already 0 to mitigate the race condition described here:
```

```
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.1: USDT Token Contract

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. In the following, we use the `rStrategy::deposit()` routine as an example. This routine is designed to trigger default handling. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice (line 93): the first one reduces the allowance to 0; and the second one sets the new allowance.

```
    // deposits stable tokens into the 3pool and stake received LPtoken(3CRV) in the
        curve 3pool gauge
    function deposit(uint[3] memory amounts) external onlyRoyaleLP(){
        uint currentTotal;
        for(uint8 i=0; i<3; i++) {
            if(amounts[i] > 0) {
               uint decimal;
               decimal=tokens[i].decimals();
               tokens[i].approve(address(pool), amounts[i]);
               currentTotal =currentTotal.add(amounts[i].mul(1e18).div(10**decimal));
            }
        }
       /* uint256 returnedAmount;
        bool status;
        (returnedAmount,status)=calculateProfit();
        if(status){
            totalProfit =totalProfit.add(returnedAmount);
        }
        else{
            totalProfit =totalProfit.sub(returnedAmount);
        } */
        uint256 mintAmount = currentTotal.mul(1e18).div(pool.get_virtual_price());
        pool.add_liquidity(amounts, mintAmount.mul(DENOMINATOR.sub(DepositSlip)).div(
            DENOMINATOR));
        //virtualPrice=pool.get_virtual_price();
        stakeLP();
    }
```

Listing 3.2: rStrategy :: deposit()

Moreover, it is important to note that for certain non-compliant ERC20 tokens (e.g., USDT), the `transfer()` function does not have a return value. However, the `IERC20` interface has defined the `transfer()` interface with a `bool` return value. As a result, the call to `transfer()` may expect a return value. With the lack of return value of `USDT`'s `transfer()`, the call will be unfortunately reverted.

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. To use this library you can add a using SafeERC20 for IERC20. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

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

Status The issue has been fixed by this commit: `4c648f6`.

3.2 Potential OOG Execution of RoyaleLP::updateWithdrawQueue()

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

Description

The `Royale Finance` protocol has a core `RoyaleLP` contract that provides the main entry for borrowing/supplying users. In addition to normal lending functionalities, e.g., `deposit()/withdraw()` and `borrow()/repay()`, the protocol further implements a number of enhancements, including the lock period (before allowing deposited assets to be withdrawn) and passive yield generation. These enhancements are useful to improve the utilization or efficiency of assets in the lending pools.

To elaborate, we show below the `updateWithdrawQueue()` routine that is an essential helper to move withdrawing users to the `reserveRecipients` so that they can claim back the assets. While it properly implements the intended functionality, it comes to our attention this routine may process a long list of withdrawing user and runs into the risk of `out-of-gas` (`OOG`) execution. If the undesirable `OOG` occurs, the normal withdraw operation will be unnecessarily affected and blocked.

```
    //this function is called when Royale Govenance withdarawl from yield generation
        pool.It add all the withdarawl amount in the reserve amount.
    //All the users who requested for the withdarawl are added to the reserveRecipients.
    function updateWithdrawQueue() internal{
        for(uint8 i=0;i<3;i++){
            reserveAmount[i]=reserveAmount[i].add(totalWithdraw[i]);
            totalWithdraw[i]=0;
        }
        for(uint i=0; i<withdrawRecipients.length; i++) {
            reserveRecipients[withdrawRecipients[i]]=true;
```

```
            isInQ[withdrawRecipients[i]]=false;
        }
        uint count=withdrawRecipients.length;
        for(uint i=0;i<count;i++){
            withdrawRecipients.pop();
        }
    }
```

Listing 3.3: RoyaleLP::updateWithdrawQueue()

Recommendation Timely monitor the length of `withdrawRecipients` and proactively call `withdraw()` to clean up the list.

Status The issue has been fixed by this commit: `4c648f6`.

3.3 Two-Step Transfer Of Privileged Account Ownership

- ID: PVE-003
- Severity: Informational
- Likelihood: Low
- Impact: N/A
- Targets: `Multiple Contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-282 [3]

Description

The `Royale Finance` protocol implements a rather basic access control mechanism that allows a privileged account, i.e., `wallet`, to be granted exclusive access to typically sensitive functions (e.g., withdraw all liquidity and rebalance the investment). Because of the privileged access and the implications of these sensitive functions, the `wallet` account is essential for the protocol-level safety and operation. In the following, we elaborate with the `wallet` account.

Within the governing contract `RoyaleLP`, a specific function, i.e., `transferOwnership()`, is provided to allow for possible `wallet`/`owner` updates. However, current implementation achieves its goal within a single transaction. This is reasonable under the assumption that the new `_wallet` parameter is always correctly provided. However, in the unlikely situation, when an incorrect `_wallet` is provided, the contract owner may be forever lost, which might be devastating for `Royale Finance` operation and maintenance.

As a common best practice, instead of achieving the owner update within a single transaction, it is suggested to split the operation into two steps. The first step initiates the owner update intent and the second step accepts and materializes the update. Both steps should be executed in two separate transactions. By doing so, it can greatly alleviate the concern of accidentally transferring the contract owner to an uncontrolled address. In other words, this two-step procedure ensures that a owner public key cannot be nominated unless there is an entity that has the corresponding private key. This is explicitly designed to prevent unintentional errors in the owner transfer process.

```
    function transferOwnership(address _wallet) external onlyWallet(){
        wallet =_wallet;
    }
```

Listing 3.4: RoyaleLP::transferOwnership()

Recommendation Implement a two-step approach for owner update (or transfer): `transferOwnership()` and `acceptOwnership()`.

Status The issue has been fixed by this commit: `4c648f6`.

3.4 Potential Underflow For Unlimited Discounted Amounts

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-282 [3]

Description

The `Royale Finance` protocol has developed three different staking lots, i.e., `StakingLotKing`, `StakingLotQueen`, and `StakingLotRoyaleFlush`. These staking lots mainly differ in their lot prices. In the following, we notice the staking lot implementation lacks proper `SafeMath` protection that may result in unintended underflow.

To illustrate, we show below the `_beforeTokenTransfer()` helper routine. This helper routine is internally used to facilitate the staking lot pool token transfer from one account to another. Specifically, it is called in every single `transfer()` operation.

```
    function _beforeTokenTransfer(address from, address to, uint256 amount) internal
        override {
        if (from != address(0)) saveProfit(from);
        if (to != address(0)) saveProfit(to);
        if (
            lastBoughtTimestamp[from].add(lockupPeriod) > block.timestamp &&
            lastBoughtTimestamp[from] > lastBoughtTimestamp[to]
        ) {
            require(
                !_revertTransfersInLockUpPeriod[to],
                "the recipient does not accept blocked funds"
            );
            lastBoughtTimestamp[to] = lastBoughtTimestamp[from];
```

```
1134         }
1135
1136         if(isDiscounted[from]>0){
1137             isDiscounted[from] -= amount;
1138             isDiscounted[to] += amount;
1139         }
1140     }
```

Listing 3.5: StakingLotKing::_beforeTokenTransfer()

The analysis with the above helper routine indicates the `isDiscounted` updates are incorrect (lines $1137-1138$). In fact, a malicious actor can take advantage of it to create a crafted situation so that an underflowed computation of `isDiscounted[from]` can be caused (line 1137). Once it is underflowed, the lack of `SafeMath` in `buyWithNFT()` can be further exploited to allow the malicious actor to buy the lots at the discounted price, and later sell them as the normal price for profit.

Recommendation Apply the `SafeMath` to block unintended overflows and/or underflows.

Status The issue has been fixed by this commit: `4c648f6`.

3.5 Simplification of RoyaleLP::requestWithdrawWithRPT()

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

Description

As discussed in Section 3.2, the `Royale Finance` protocol has a core `RoyaleLP` contract that provides the main entry for borrowing/supplying users. In addition to normal lending functionalities, e.g., `deposit()/withdraw()` and `borrow()/repay()`, the protocol further implements a number of enhancements. In the following, we analyze the lock up period logic implemented in `requestWithdrawWithRPT()`.

To elaborate, we show below the full `requestWithdrawWithRPT()` implementation. Our analysis shows an opportunity to simplify current logic. In particular, the current statements at lines $240-244$ can be revised to be a much simplifier one, i.e., `require(availableRPT>=amount, "NA")`.

```
236     function requestWithdrawWithRPT(uint256 amount,uint256 _index) external nonReentrant
            validAmount(amount){
237         require(!reserveRecipients[msg.sender],"Claim first");
238         require(rpToken.balanceOf(msg.sender) >= amount, "low RPT");
239         (,uint availableRPT)=availableLiquidity(msg.sender,_index,true );
240         bool instant=true;
241          if(availableRPT < amount) {
```

```
                instant=false;
            }
            require(instant,"NA");
            uint256 total = calculateTotalToken(true);
            uint256 tokenAmount;
            tokenAmount=amount.mul(total).div(rpToken.totalSupply());
            require(tokenAmount <= calculateTotalToken(false),"Not Enough Pool balance");
            uint decimal;
            decimal=tokens[_index].decimals();
            checkWithdraw(tokenAmount.mul(10**decimal).div(10**18),amount,_index);
        }
```

Listing 3.6: RoyaleLP::requestWithdrawWithRPT()

Recommendation Simplify the current `requestWithdrawWithRPT()` logic.

Status The issue has been fixed by this commit: `4c648f6`.

3.6 Non-Permissioned ERC1155Receiver::setShouldReject()

- ID: PVE-006
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `ERC1155Receiver`
- Category: Security Features [7]
- CWE subcategory: CWE-282 [3]

Description

PeckShield

As discussed in Section 3.4, the `Royale Finance` protocol has developed three different staking lots, i.e., `StakingLotKing`, `StakingLotQueen`, and `StakingLotRoyaleFlush`. Each staking lot is implemented by inheriting from the `ERC1155Receiver` contract. Note that `ERC1155` defines a standard interface for contracts that manage multiple token types. As a result, a single deployed contract may include any combination of fungible tokens, non-fungible tokens or other configurations (e.g. semi-fungible tokens).

```
contract ERC1155Receiver is CommonConstants {

    // Keep values from last received contract.
    bool public shouldReject;

    bytes public lastData;
    address public lastOperator;
    address public lastFrom;
    uint256 public lastId;
    uint256 public lastValue;
```

```
    function setShouldReject(bool _value) public {
        shouldReject = _value;
    }

    function onERC1155Received(address _operator, address _from, uint256 _id, uint256
        _value, bytes calldata _data) external returns(bytes4) {
        lastOperator = _operator;
        lastFrom = _from;
        lastId = _id;
        lastValue = _value;
        lastData = _data;
        if (shouldReject == true) {
            revert("onERC1155Received: transfer not accepted");
        } else {
            return ERC1155_ACCEPTED;
        }
    }

    function onERC1155BatchReceived(address _operator, address _from, uint256[] calldata
         _ids, uint256[] calldata _values, bytes calldata _data) external returns(bytes4
        ) {
        lastOperator = _operator;
        lastFrom = _from;
        lastId = _ids[0];
        lastValue = _values[0];
        lastData = _data;
        if (shouldReject == true) {
            revert("onERC1155BatchReceived: transfer not accepted");
        } else {
            return ERC1155_BATCH_ACCEPTED;
        }
    }

    // ERC165 interface support
    function supportsInterface(bytes4 interfaceID) external view returns (bool) {
        return  interfaceID == 0x01ffc9a7    // ERC165
                interfaceID == 0x4e2312e0;    // ERC1155_ACCEPTED ^
                    ERC1155_BATCH_ACCEPTED;
    }
}
```

Listing 3.7: The ERC1155Receiver Contract

However, it comes to our attention that this `ERC1155Receiver` contract has a public function, i.e., `setShouldReject()`, which allows any one to set the `shouldReject` flag. In other words, a denial-of-service situation may occur with the inheriting token contracts in their `transfer()` functionality.

Recommendation Authenticate the caller to `setShouldReject()` to be a trusted entity.

Status The issue has been fixed by this commit: `4c648f6`.

3.7 Possible Friction That May Block User Withdrawals

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: High
- Targets: `RoyaleLP`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

As discussed earlier, the `RoyaleLP` contract provides the main entry for borrowing/supplying users. In the section, we further examine the withdraw logic implemented in `requestWithdrawWithRPT()`. Our analysis leads to the finding of a potential denial-of-service situation that may block legitimate user withdrawal.

To elaborate, we show below the full `requestWithdrawWithRPT()` implementation. After performing necessary validations on the withdrawing user, it further delegates the call to an internal handler, i.e., `checkWithdraw()`.

```
    function requestWithdrawWithRPT(uint256 amount,uint256 _index) external nonReentrant
         validAmount(amount){
        require(!reserveRecipients[msg.sender],"Claim first");
        require(rpToken.balanceOf(msg.sender) >= amount, "low RPT");
        (,uint availableRPT)=availableLiquidity(msg.sender,_index,true );
        bool instant=true;
         if(availableRPT < amount) {
             instant=false;
        }
        require(instant,"NA");
        uint256 total = calculateTotalToken(true);
        uint256 tokenAmount;
        tokenAmount=amount.mul(total).div(rpToken.totalSupply());
        require(tokenAmount <= calculateTotalToken(false),"Not Enough Pool balance");
        uint decimal;
        decimal=tokens[_index].decimals();
        checkWithdraw(tokenAmount.mul(10**decimal).div(10**18),amount,_index);
    }
```

Listing 3.8: RoyaleLP::requestWithdrawWithRPT()

The internal handler evaluates whether current pool balance is able to satisfy the withdraw request. If yes, it transfers the requested amount to user (line 99). Otherwise, it puts the request into the withdrawal queue (line 104).

```
88      function checkWithdraw(uint256 amount,uint256 burnAmt,uint _index) internal{
89          uint256 poolBalance;
90          poolBalance = getBalances(_index);
91          rpToken.burn(msg.sender, burnAmt);
```

```
        if(amount < poolBalance) {
            uint decimal;
            bool result;
            decimal=tokens[_index].decimals();
            uint temp = amount.sub(amount.mul(fees).div(DENOMINATOR));
            selfBalance=selfBalance.sub(temp.mul(10**18).div(10**decimal));
            updateLockedRPT(msg.sender,burnAmt);
            result = tokens[_index].transfer(msg.sender, temp);
            require(result,"Transfer not Succesful");
            emit userRecieved(msg.sender, temp);
         }
         else {
            _takeBackQ(amount,burnAmt,_index);
            emit userAddedToQ(msg.sender, amount);
        }
    }
```

Listing 3.9: RoyaleLP::checkWithdraw()

Interestingly, there is a requirement inside `_takeBackQ()`: `require((totalWithdraw[0]+total)<=YieldPoolBalance,"Not enough balance")`. In essence, it requires the `YieldPoolBalance` should be able to accommodate the withdraw request. Note this is not necessarily the case! The assets may be invested in the strategy, i.e., the yielding pool. There are still some portion of assets held in current contract.

```
    function _takeBackQ(uint256 amount,uint256 _burnAmount,uint256 _index) internal {
        amountWithdraw[msg.sender][_index] =amountWithdraw[msg.sender][_index].add(
            amount);
        amountBurnt[msg.sender][_index]=amountBurnt[msg.sender][_index].add(_burnAmount)
            ;
        totalWithdraw[_index] =totalWithdraw[_index].add(amount);
        uint total;
        total=(totalWithdraw[1].add(totalWithdraw[2])).mul(1e18).div(10**6);
        require((totalWithdraw[0]+total)<=YieldPoolBalance,"Not enough balance");
        if(!isInQ[msg.sender]) {
            isInQ[msg.sender] = true;
            withdrawRecipients.push(msg.sender);

        }

    }
```

Listing 3.10: RoyaleLP::_takeBackQ()

For simplicity, consider the following scenario with only one supplying user with $10,000$ DAIs. Right after the very first deposit, the pool owner calls `rebalance()` to invest half of them into the yield-generating strategy. And the user wants to withdraw back, the exact requirement inside `_takeBackQ()` effectively blocks the legitimate request.

Recommendation Revisit the requirement logic that may need to take into consideration of the rebalance logic.

Status The issue has been fixed by this commit: `72d6f30`.

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 [4]

Description

In the `Royale Finance` protocol, the privileged owner account plays a critical role in governing and regulating the system-wide operations (e.g., settings of risk parameters). It also has the privilege to regulate or govern the flow of assets for borrowing and lending among the involved components.

In the following, we show a representative privileged operation in the `RoyaleLP` protocol. This routine essentially allows the pool owner to collect all funds in current pool.

```
    function transferAllFunds(address _address)external onlyWallet(){
        selfBalance=0;
        for(uint8 i=0;i<3;i++){
            tokens[i].transfer(_address,tokens[i].balanceOf(address(this)));
        }
    }
```

Listing 3.11: RoyaleLP:: transferAllFunds ()

Also, if we examine the `ERC20Recovery::recoverERC20()`, which also allows the privileged owner to transfer all funds held in the contract. Currently, the `tokenSwap` contract inherits from this `ERC20Recovery` contract.

```
abstract contract ERC20Recovery is Ownable{
    using SafeERC20 for IERC20;
    function recoverERC20(IERC20 token) external onlyOwner {
        token.safeTransfer(owner(), token.balanceOf(address(this)));
    }
}
```

Listing 3.12: ERC20Recovery::recoverERC20()

We emphasize that the privilege assignment with certain accounts is necessary and required for proper protocol operations. However, it is worrisome if the above two routines are managed by an EOA account. In fact, there is a clear need for a proper governance to regulate and restrict such operations. The discussion with the team has confirmed that this owner account will be managed by a multi-sig account.

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 and partially mitigated with a multi-sig account to regulate the governance privileges.

3.9 Possible CooldownTimestamp Manipulation

- ID: PVE-0009
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `StakedRoya`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

To engage protocol users, the `Royale Finance` protocol has a `StakedRoya` contract that has been designed to reward participating users if they stake their tokens to receive pro-rata staking rewards. In order to prevent possible flashloan-assisted front-running attacks that may claim the majority of rewards, the staking logic is designed to have a cooldown period for staked assets. For each account, the associated cooldown period is recorded internally as `stakersCooldowns()`.

When there is a stake operation, the staking user's cooldown timestamp is properly updated. When the pool token is transferred, the receiver's cooldown timestamp will also be updated. The new cooldown timestamp is calculated in the following `getNextCooldownTimestamp()` routine.

```
function getNextCooldownTimestamp(
  uint256 fromCooldownTimestamp,
  uint256 amountToReceive,
  address toAddress,
  uint256 toBalance
) public returns (uint256) {
  uint256 toCooldownTimestamp = stakersCooldowns[toAddress];
  if (toCooldownTimestamp == 0) {
    return 0;
  }

  uint256 minimalValidCooldownTimestamp = block.timestamp.sub(COOLDOWN_SECONDS).sub(
    UNSTAKE_WINDOW
  );
```

```

    if (minimalValidCooldownTimestamp > toCooldownTimestamp) {
      toCooldownTimestamp = 0;
    } else {
      uint256 fromCooldownTimestamp = (minimalValidCooldownTimestamp >
          fromCooldownTimestamp)
        ? block.timestamp
        : fromCooldownTimestamp;

      if (fromCooldownTimestamp < toCooldownTimestamp) {
        return toCooldownTimestamp;
      } else {
        toCooldownTimestamp = (
          amountToReceive.mul(fromCooldownTimestamp).add(toBalance.mul(
              toCooldownTimestamp))
        )
          .div(amountToReceive.add(toBalance));
      }
    }
    stakersCooldowns[toAddress] = toCooldownTimestamp;

    return toCooldownTimestamp;
  }
```

Listing 3.13: StakedToken:getNextCooldownTimestamp()

If a staking user has not passed the cooldown timestamp, the staked funds will be locked inside the staking contract. It comes to out attention that this above `getNextCooldownTimestamp()` routine is public, which means any one is able to call it. Also, it surprisingly updates the given `toAddress`'s cooldown timestamp directly. In other words, a malicious actor may simply lock another victim's staking funds inside the contract.

Recommendation Restrict the `getNextCooldownTimestamp()` call or make the function view-only.

Status The issue has been fixed by this commit: `4c648f6`.

3.10 Possible Sandwich/MEV For Reduced Return And Skewed Withdrawals

- ID: PVE-010
- Severity: High
- Likelihood: High
- Impact: High
- Target: `CurveStrategy`
- Category: Time and State [10]
- CWE subcategory: CWE-682 [5]

Description

As mentioned in Section 3.2, the `Royale Finance` protocol has a yield-generating pool that is managed by the `rStrategy`. This strategy essentially invest the liquidity into the popular `Curve` pool, harvest growing yields, and sell any gains, if any, to the original assets.

Specifically, if we examine the `rStrategy` implementation, there is a `sellCRV()` routine that can be by the owner to basically convert the collected `CRV` rewards to the designated stable token (lines 245 – 251) for the next round of investment.

```
226     // Function to sell CRV using uniswap to any stable token and send that token to an address
227     function sellCRV(uint8 _index) public onlyWallet() returns(uint256) {  //here index=0 means convert crv into DAI , index=1 means crv into USDC , index=2 means crv into USDT
228         uint256 crvAmt = IERC20(crvAddr).balanceOf(address(this));
229         uint256 prevCoin = tokens[_index].balanceOf(address(this));
230         require(crvAmt > 0, "insufficient CRV");
231         crvAmt=crvAmt.mul(crvBreak).div(DENOMINATOR);
232         crvAddr.approve(address(uniAddr), crvAmt);
233         address[] memory path;
234         if(TEST) {
235             path = new address[](2);
236             path[0] = address(crvAddr);
237             path[1] = address(tokens[_index]);

239         } else {
240             path = new address[](3);
241             path[0] = address(crvAddr);
242             path[1] = wethAddr;
243             path[2] = address(tokens[_index]);
244         }
245         UniswapI(uniAddr).swapExactTokensForTokens(
246             crvAmt,
247             uint256(0),
248             path,
249             address(this),
250             now + 1800
251         );
```

```
        uint256 postCoin=tokens[_index].balanceOf(address(this));
        tokens[_index].transfer(yieldDistributor,postCoin.sub(prevCoin));
        emit yieldTransfered(_index,postCoin.sub(prevCoin));
    }
```

Listing 3.14: rStrategy :: sellCRV()

We notice the collected yields are routed to `UniswapV2` in order to swap them to the intended stable coins as rewards. And the swap operation does not specify any restriction on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of yielding.

```
    modifier onlyRoyaleLP() {
        require(msg.sender == royaleAddress || true, "Not authorized");
        _;
    }

    function withdraw(uint[3] memory amounts) external onlyRoyaleLP() {
        uint256 max_burn = pool.calc_token_amount(amounts,false);
        max_burn=max_burn.mul(DENOMINATOR.add(withdrawSlip)).div(DENOMINATOR);
        unstakeLP(max_burn);
        pool.remove_liquidity_imbalance(amounts, max_burn);
        for(uint8 i=0;i<3;i++){
            if(amounts[i]!=0){
                tokens[i].transfer(royaleAddress, tokens[i].balanceOf(address(this)));
            }
        }
        stakeLP();
    }
```

Listing 3.15: rStrategy :: sellCRV()

In the same vein, we notice the presence of the `withdraw()` routine that is guarded with a modifier `onlyRoyaleLP`. However, it turns out that this modifier is essentially a `no-op`, which could be exploited to withdraw current funds from a potentially manipulated or highly skewed pool. As a result, the `withdraw()` may be exploited to burn a huge amount of `3CRV` pool tokens for a small withdrawn amount.

Note 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 or the `rStrategy` contract in our case 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 fixed by this commit: `4c648f6`.

PeckShield

4 | Conclusion

In this audit, we have analyzed the `Royale Finance` design and implementation. The system presents a unique, robust offering as a decentralized protocol for directing optimized stablecoin liquidity pools toward an on-chain funding solution for `iGaming` products. The current code base is well structured and neatly organized. 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.

PeckShield

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-282: Improper Ownership Management. https://cwe.mitre.org/data/definitions/282.html.

[4] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.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: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.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.

PeckShield