SMART CONTRACT AUDIT REPORT

for

ApeRocket Finance

Prepared By: Yiqun Chen

PeckShield

September 7, 2021

Document Properties

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

Version Info

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

Contact

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

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

Contents

1 Introduction

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

1.1 About ApeRocket Finance

`ApeRocket Finance` is a suite of products in DeFi that provides yield optimization strategies through the `Binance Smart Chain (BSC)`, using `ApeSwap` liquidity. Through automation, `ApeRocket` allows apes of all kinds to reap the benefits of compounding without additional steps. `ApeRocket` calculates the most optimal compound frequency and automatically compounds your tokens to provide you the best yields.

The basic information of ApeRocket Finance is as follows:

Table 1.1: Basic Information of ApeRocket Finance

Item	Description
Name	ApeRocket Finance
Website	https://aperocket.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 7, 2021

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

- https://github.com/warren-0x/platform.git (a8323c0)

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 the 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 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

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

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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

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

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

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

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `ApeRocket` 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	3	■■■
Low	5	■■■■■
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, 3 medium-severity vulnerabilities, 5 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key ApeRocket Finance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Non-initialization of pid in BaseStrategy::constructor()	Business Logic	Fixed
PVE-002	Low	Unintended Reverts in VotingEscrow::mintTo()	Business Logic	Fixed
PVE-003	High	Incorrect Debt Accounting in Vault::_transferUserInfo()	Business Logic	Fixed
PVE-004	Low	Potential Reentrancy Risk in depositTo()	Time and State	Fixed
PVE-005	Low	Less Optimal Swaps For Liquidity Addition	Coding Practice	Fixed
PVE-006	Informational	Simplified Logic of minter()/boostManager() in Vault	Coding Practice	Fixed
PVE-007	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-008	Low	Possible Costly LPs From Improper Vault Initialization	Time and State	Confirmed
PVE-009	Medium	Improved Logic of Vault::_withdraw()	Business Logic	Fixed
PVE-010	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Fixed

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

3 Detailed Results

3.1 Non-initialization of pid in BaseStrategy::constructor()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `BaseStrategy`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [6]

Description

The `ApeRocket` protocol provide users a number of yield optimization strategies. To facilitate the strategy construction and management, the protocol provides a base strategy template, i.e., `BaseStrategy`. This `BaseStrategy` is inherited by all strategy instances.

To elaborate, we show below its `constructor()` function. While it properly configures a number of parameters and states, it fails to properly initialize the `pid` state. Note this `pid` is used in other routines. For example, the `shutdownStrategy()` is used to turn off this strategy by retrieving all funds back to the vault. As a result, an uninitialized `pid` may cause undesirable consequence when the strategy needs to shut down.

```
    constructor(
        address _vault,
        address _feeManager,
        address _rewards_contract,
        uint16 _pid
    ) internal {
        require(_vault != address(0));
        require(_rewards_contract != address(0));

        vault = _vault;
        rewards_contract = _rewards_contract;

        feeManager = _feeManager;
        IERC20(WBNB).safeApprove(_feeManager, uint256(-1));
        keeper = msg.sender;
```

```
    }
```

Listing 3.1: BaseStrategy::constructor()

Recommendation Properly initialize the pool id pid when the strategy is being configured.

Status The issue has been fixed in the following PR: 1.

3.2 Unintended Reverts in VotingEscrow::mintTo()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: VotingEscrow
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [6]

Description

The ApeRocket protocol has a voting escrow contract VotingEscrow that is designed to allow users to gain voting power by staking the required assets. While examining its implementation, we notice one of its public functions, i.e., mintTo(), needs to be improved.

To elaborate, we show below the full implementation of the mintTo() function. It implements a rather straightforward logic in transferring staked assets and granting the users respective voting power. However, in the case when the staked assets are simply appended with the last entry with the same end date, the new voting power is not properly calculated and granted. In other words, the staking user may not receive the due voting power from the staking operation.

```
    function mintTo(address _addr, uint256 _value) external onlyMinter {
        LockedBalance[] storage _vested = vested[_addr];
        uint256 i = _vested.length;
        uint256 _now = block.timestamp;
        uint256 _vp;
        uint256 end = _now.add(VESTING_DAYS * 1 days);

        if (i == 0  _vested[i - 1].end < end) {
            _vp = votingPowerLockedDays(_value, VESTING_DAYS);
            _vested.push(LockedBalance({amount: _value, end: end, vp: _vp}));
        } else {
            _vested[i - 1].amount = _vested[i - 1].amount.add(_value);
        }

        require(_vp > 0, "No benefit to lock");
        if (_value > 0) {
            IERC20(lockedToken).safeTransferFrom(msg.sender, address(this), _value);
        }
```

```
145
146         _mint(_addr, _vp);
147         mintedForVest[_addr] = mintedForVest[_addr].add(_vp);
148         emit Deposit(_addr, _value, end, _now);
149     }
```

Listing 3.2: `VotingEscrow::mintTo()`

Recommendation Revise the above `mintTo()` to properly compute the (new) voting power (lines $137-139$) in all cases.

Status The issue has been fixed in the following PR: `1`.

3.3 Incorrect Debt Accounting in Vault::_transferUserInfo()

- ID: PVE-003
- Severity: High
- Likelihood: High
- Impact: High
- Target: `Vault`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [6]

Description

In the `ApeRocket` protocol, there is an essential `Vault` contract that accepts users funds for investments through the supported strategies. To properly record the contribution from each investing user, the contract computes the share of each user by implementing itself as an `ERC20`-compliant token. While the share tokenization greatly facilitates the reward computation, the fact that it allows the `Vault` share to be transferred requires proper reward distribution.

To elaborate, we show below the related `_transferUserInfo()` helper routine. This helper routine is designed to properly maintain internal accouting to keep track of each user's contribution or debt. However, our analysis shows its logic is currently flawed. In particular, the transferred amount (or share) may not be the full amount (or share) of the sender. In fact, it may only transfer a small portion of the current balance. Because of that, the internal states, i.e., `reward_debt` and `space_debt`, need to be updated accordingly with the portion, not the full amount.

```
    function _transferUserInfo(
        address sender,
        address recipient,
        uint256 shares
    ) internal {
        UserInfo storage old_user = userInfo[sender];
        UserInfo storage new_user = userInfo[recipient];

        new_user.reward_debt = new_user.reward_debt.add(old_user.reward_debt);
```

```
339         new_user.space_debt = new_user.space_debt.add(old_user.space_debt);
340         new_user.last_deposit_time = new_user.space_debt.add(old_user.last_deposit_time)
                ;
341     }
```

Listing 3.3: `Vault::_transferUserInfo()`

Moreover, the update of the `last_deposit_time` (line 340) is also problematic as it directly adds the `space_debt` amount with the sender's `last_deposit_time`!

Recommendation Revise the above `_transferUserInfo()` routine to properly maintain the internal accounting for reward and debt distribution.

Status The issue has been fixed in the following PR: `1`.

3.4 Potential Reentrancy Risk in depositTo()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact:Low
- Target: `Vault`
- Category: Time and State [11]
- CWE subcategory: CWE-682 [5]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [16] exploit, and the recent `Uniswap/Lendf.Me` hack [15].

We notice there are several occasions the `checks-effects-interactions` principle is violated. Using the `Vault` as an example, the `depositTo()` 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 221) starts before effecting the update on internal states (e.g., line 236), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the very same `depositTo()` function.

```
    // Allow to deposit in behalf a user. User is then blocklocked
    function depositTo(uint256 amount, address recipient) external whenNotPaused
        checkBlockLocked isAllowed {
        _deposit(amount, recipient);
    }
```

Listing 3.4: `Vault::_deposit()`

```
    function _deposit(uint256 _amount, address recipient) internal {
        _lock(recipient);
        uint256 _pool = balance();

        uint256 _before = stakingToken.balanceOf(address(this));
        stakingToken.safeTransferFrom(msg.sender, address(this), _amount);
        uint256 _after = stakingToken.balanceOf(address(this));
        _amount = _after.sub(_before); // Additional check for deflationary tokens

        // Save initial deposit from user
        UserInfo storage user = userInfo[recipient];
        user.principal = user.principal.add(_amount);
        user.last_deposit_time = block.timestamp;

        uint256 shares = 0;
        if (totalSupply() == 0) {
            shares = _amount;
        } else {
            shares = (_amount.mul(totalSupply())).div(_pool);
        }
        _mint(recipient, shares);
        _depositIntoStrategy();
    }
```

Listing 3.5: `Vault::_deposit()`

Recommendation Apply necessary reentrancy prevention by making use of the common `nonReentrant` modifier.

Status The issue has been fixed in the following PR: `1`.

3.5 Less Optimal Swaps For Liquidity Addition

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1041 [1]

Description

As mentioned earlier, the `ApeRocket` protocol is designed with a number of yield optimization strategies. Accordingly, there is a constant need of swapping one token to another. In the following, we examine related swap routines that are designed to assist the token swapping.

To elaborate, we show below a helper routine named `notifyRewardAmount()`. The routine is used to convert half assets to `BNB` and then add them as liquidity (`SPACE_WBNB`) before sending them to the fee manager. It comes to our attention that the current approach converts half assets to `BNB` and then sends the another half with the converted `BNB` as liquidity, which may result in a small amount of `BNB` unspent in the current contract. In other words, the current conversion approach is not optimal. Note that the same issue is also present in another routine, i.e., `StrategyOptimizer::_swapToStakingToken()`.

```
    // Strategies will always convert assets to BNB before sending them to the fee
        manager
    function notifyRewardAmount(uint256 amount) external onlyAuthorized {
        IERC20(WBNB).safeTransferFrom(msg.sender, address(this), amount);

        uint256 bnbBalance = IERC20(WBNB).balanceOf(address(this));
        _swapToSpaceBNB(bnbBalance.div(2));
        uint256 balance = IERC20(SPACE_BNB_LP).balanceOf(address(this));

        if(balance > 0) {
            IMultiRewardsDistributionPool(multiRewardsDistributionPool).
                notifyRewardAmount(SPACE_BNB_LP, balance);
        }
    }
```

Listing 3.6: `FeeManager::notifyRewardAmount()`

```
    function _swapToSpaceBNB(uint256 _amount) internal {
        address[] memory path;
        path = new address[](2);
        path[0] = WBNB;
        path[1] = SPACE;

        IPancakeRouter02(ROUTER).swapExactTokensForTokens(
            _amount,
            0,
```

```
            path,
            address(this),
            block.timestamp.add(60)
        );
        _addLiquidity(SPACE, WBNB);
    }

    function _addLiquidity(address token0, address token1) internal {
        uint256 _token0Balance = IERC20(token0).balanceOf(address(this));
        uint256 _token1Balance = IERC20(token1).balanceOf(address(this));

        IPancakeRouter02(ROUTER).addLiquidity(
            token0,
            token1,
            _token0Balance,
            _token1Balance,
            0,
            0,
            address(this),
            block.timestamp
        );
    }
```

Listing 3.7: `FeeManager::_swapToSpaceBNB()/addLiquidity()`

Moreover, the above conversion does not specify any slippage restriction, which may be easily exploited in a possible sandwich or MEV attack for reduced return. Affected routines also include `BaseStrategy::_assessPerformanceFees()` and `MultiRewardsDistributionPool::getReward()`.

Recommendation Perform an optimal allocation of assets between two tokens for matched liquidity addition. Also add necessary slippage control to avoid unnecessary loss of swaps.

Status The issue has been fixed in the following PR: `1`.

3.6 Simplified Logic of minter()/boostManager() in Vault

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [9]
- CWE subcategory: CWE-563 [4]

Description

The `ApeRocket` protocol makes good use of a number of reference contracts, such as `ERC20`, `ReentrancyGuard`, `SafeMath`, and `Address`, to facilitate its code implementation and organization. For example, the

Vault smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the Vault contract, it has defined a getter function minter (), which can be simplified as `return _spaceMinter;` without the need of the `if` condition (line 76). Another function boostManager() can be similarly improved as well.

```
    function minter() public view returns (address) {
        return _spaceMinter != address(0) ? _spaceMinter : address(0);
    }

    function boostManager() public view returns (address) {
        return _boostManager != address(0) ? _boostManager : address(0);
    }
```

Listing 3.8: Vault::minter()/boostManager()

```
    function withdrawAll() external nonReentrant checkBlockLocked isAllowed {
        uint256 shares = balanceOf(msg.sender);
        _getRewards(msg.sender);

        UserInfo storage user = userInfo[msg.sender];
        user.principal = 0;
        user.reward_debt = balanceOf(msg.sender).mul(accRewardPerShare).div(1e12);
        user.space_debt = balanceOf(msg.sender).mul(accSpacePerShare).div(1e12);
        user.last_deposit_time = 0;

        _withdraw(shares);
    }
```

Listing 3.9: Vault::withdrawAll()

In addition, the analysis of another routine withdrawAll() in the same contract shows that the user-related reward_debt and space_debt can be simply reset as 0 since all shares are withdrawn.

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

Status The issue has been fixed in the following PR: 1.

3.7 Trust Issue Of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [7]
- CWE subcategory: CWE-287 [2]

Description

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

```
    function whitelistContract(address _contract) external onlyOwner {
        whitelist[_contract] = true;
    }

    function blacklistContract(address _contract) external onlyOwner {
        whitelist[_contract] = false;
    }
```

Listing 3.10: A number of representative setters in SafeAccessControl

```
    function setAccessToMint(address _contract) external onlyOwner {
        minters[_contract] = true;
    }

    function revokeAccessToMint(address _contract) external onlyOwner {
        minters[_contract] = false;
    }
```

Listing 3.11: A number of representative setters in VotingEscrow

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

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

Status This issue has been confirmed. The team clarifies that a timelock contract will be the actual privileged owner.

3.8 Possible Costly LPs From Improper Vault Initialization

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: Vault
- Category: Time and State [8]
- CWE subcategory: CWE-362 [3]

Description

The ApeRocket protocol allows users to deposit supported assets and get in return av-wrapped tokens to represent the pool share. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the pool token, i.e., avUSDC, extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the deposit() routine. This routine is used for participating users to deposit the supported assets (e.g., USDC) and get respective avUSDC pool tokens in return. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
    function deposit(uint256 amount) external nonReentrant whenNotPaused
        checkBlockLocked isAllowed {
        _deposit(amount, msg.sender);
    }
```

Listing 3.12: Vault::deposit()

```
    function _deposit(uint256 _amount, address recipient) internal {
        _lock(recipient);
        uint256 _pool = balance();

        uint256 _before = stakingToken.balanceOf(address(this));
        stakingToken.safeTransferFrom(msg.sender, address(this), _amount);
        uint256 _after = stakingToken.balanceOf(address(this));
        _amount = _after.sub(_before); // Additional check for deflationary tokens

        // Save initial deposit from user
        UserInfo storage user = userInfo[recipient];
        user.principal = user.principal.add(_amount);
        user.last_deposit_time = block.timestamp;

        uint256 shares = 0;
        if (totalSupply() == 0) {
            shares = _amount;
        } else {
            shares = (_amount.mul(totalSupply())).div(_pool);
        }
        _mint(recipient, shares);
        _depositIntoStrategy();
```

```
238     }
```

Listing 3.13: `Vault::_deposit()`

Specifically, when the pool is being initialized (line 231), the share value directly takes the value of `amount` (line 232), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `shares = _amount = 1 WEI`. With that, the actor can further deposit a huge amount of `USDC` assets with the goal of making the `avUSDC` pool token extremely expensive.

An extremely expensive `avUSDC` pool token can be very inconvenient to use as a small number of $1WEI$ may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool tokens.

Recommendation Revise current execution logic of `deposit()` to defensively calculate the share amount when the pool is being initialized. An alternative solution is to ensure guarded launch that safeguards the first deposit to avoid being manipulated.

Status This issue has been confirmed. The team will exercise extra caution in properly initializing the vault.

3.9 Improved Logic of Vault::_withdraw()

- ID: PVE-009
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Vault`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [6]

Description

As mentioned earlier, the `ApeRocket` protocol allows users to invest their assets for returns. Accordingly, it provides users a number of public functions: `deposit()`, `withdraw()`, and `getRewards()`. The first function invests the user funds, the second function allows the user to withdraw their funds, and the third one allows the user to claim rewards. While examining the related functions, we notice an issue in current implementation.

To elaborate, we show below the related `_claimRewards()` helper that is a part of the `getRewards()` function. This helper implements a rather straightforward logic in retrieving the user rewards. However, it comes to our attention that the logic makes an implicit assumption of the contract balance is sufficient in satisfying the user withdraw request (line 297). Unfortunately, this assumption may not always hold! When violated, it may be of serious detriment to the normal functionality, including the user withdraws and claims of pending rewards.

```
    function _claimRewards(address _user) internal {
        UserInfo storage user = userInfo[_user];
        if (balanceOf(_user) > 0) {
            uint256 reward = earned(_user);
            if (reward > 0) {
                totalPendingRewards = totalPendingRewards.sub(reward);
                uint256 balance = farmedToken.balanceOf(address(this));
                if (balance < reward) {
                    _withdrawRewards(balance, reward);
                }
                farmedToken.safeTransfer(_user, reward);
                user.reward_debt = balanceOf(_user).mul(accRewardPerShare).div(1e12);
            }
        }
    }
```

Listing 3.14: `Vault::_claimRewards()`

Note this issue is applicable to both `_claimRewards()` and `_withdraw()`.

Recommendation Revise the above `_claimRewards()` routine to properly take into account the scenario with an insufficient balance. An example revision is shown as below:

```
    function _claimRewards(address _user) internal {
        UserInfo storage user = userInfo[_user];
        if (balanceOf(_user) > 0) {
            uint256 reward = earned(_user);
            if (reward > 0) {
                totalPendingRewards = totalPendingRewards.sub(reward);
                uint256 balance = farmedToken.balanceOf(address(this));
                if (balance < reward) {
                    reward = _withdrawRewards(balance, reward);
                }
                farmedToken.safeTransfer(_user, reward);
                user.reward_debt = balanceOf(_user).mul(accRewardPerShare).div(1e12);
            }
        }
    }
```

Listing 3.15: Revised `Vault::_claimRewards()`

Status The issue has been fixed in the following PR: `1`.

3.10 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-010
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `BaseStrategy`
- Category: Business Logic [10]
- 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 3.16: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `shutdownStrategy()` routine in the `BaseStrategy` contract. If the `USDT` token is supported as `stakingToken`, the unsafe version of `IERC20(stakingToken).transfer(vault, availableBalance)` (line 150) may revert as there is no return value in the `USDT` token contract's `transfer()/transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
145     /* ========== EMERGENCY ONLY OR STRATEGY UPDATE ========== */
146 
147     function shutdownStrategy() external onlyVault {
148         IRewardsContract(rewards_contract).emergencyWithdraw(pid);
149         uint256 availableBalance = IERC20(stakingToken).balanceOf(address(this));
150         IERC20(stakingToken).transfer(vault, availableBalance);
151     }
```

Listing 3.17: `BaseStrategy::shutdownStrategy()`

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

Status The issue has been fixed in the following PR: `1`.

4 Conclusion

In this audit, we have analyzed the ApeRocket design and implementation. The system presents a unique, robust DeFi offering to provide users with a number of yield optimization strategies. In particular, ApeRocket calculates the most optimal compound frequency and automatically compounds your tokens to provide the best yields. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

[3] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[4] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.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: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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

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

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