SMART CONTRACT AUDIT REPORT

for

KeyOfLife Vault

Prepared By: Xiaomi Huang

PeckShield

March 21, 2023

Document Properties

Client	KeyOfLife Finance
Title	Smart Contract Audit Report
Target	KeyOfLife Vault
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 21, 2023	Xuxian Jiang	Final Release
1.0-rc1	March 20, 2023	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Key of Life Finance (KOL)` 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 KOL Finance

`Key of Life Finance` is a decentralized, multichain automizer vaults that are designed to earn compound interest on their crypto holdings. The purpose is to maximize the highest APYs with safety and efficiency. As a result, the protocol will help to optimize asset and resources allocations for DeFi community overall. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The `Key of Life Finance` Protocol

Item	Description
Client	KeyOfLife Finance
Website	https://keyoflife.fi/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 21, 2023

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

- https://github.com/KeyOfLifeFi/Kol-Vault.git (5dce530)

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

- https://github.com/KeyOfLifeFi/Kol-Vault.git (371a866)

1.2 About PeckShield

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

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on 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.

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

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

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.

1.4 Disclaimer

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

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

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

2 Findings

2.1 Summary

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

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■■■■
Low	2	■■
Informational	1	■
Total	5	

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

2.2 Key Findings

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

Table 2.1: Key KeyOfLife Vault Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated
PVE-002	Medium	Possible Sandwich/MEV Attacks To Collect Most Rewards	Time and State	Mitigated
PVE-003	Informational	Improved Reentrancy Protection in KolAutomizerVault	Time and State	Resolved
PVE-004	Low	Possible Costly LPs From Improper Initialization	Time and State	Resolved
PVE-005	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved

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 Trust Issue of Admin Keys

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

In the `KOL` protocol, the privileged `owner` account plays a critical role in governing and regulating the system-wide operations (e.g., `vault/strategy` integration, reward adjustment, and parameter setting). It also has the privilege to control or govern the flow of assets for investment or full withdrawal among the key components, i.e., `vault` and `strategy`.

With great privilege comes great responsibility. Our analysis shows that the `owner` account is indeed privileged. In the following, we show representative privileged operations in the `KOL` protocol.

```
    function _authorizeUpgrade(address) internal override onlyOwner {}

    function initParams(address _strategy) public onlyOwner {
        require(address(strategy) == address(0),"params initialized");
        strategy = IStrategy(_strategy);
        require(strategy.vault() == address(this),"wrong strategy");
        want = IERC20Upgradeable(strategy.want());
    }

    /**
     * @dev Rescues random funds stuck that the strat can't handle.
     * @param _token address of the token to rescue.
     */
    function inCaseTokensGetStuck(address _token) external onlyOwner {
        require(_token != address(want), "KolVault: STUCK_TOKEN_ONLY");
        uint256 amount = IERC20Upgradeable(_token).balanceOf(address(this));
        IERC20Upgradeable(_token).safeTransfer(msg.sender, amount);
        emit RescuesTokenStuck(_token, amount);
```

```
    }
```

Listing 3.1: Various Setters in KolAutomizerVault

We emphasize that the privilege assignment with various core contracts is necessary and required for proper protocol operations. However, it is worrisome if the `owner` is not governed by a `DAO`-like structure. We point out that a compromised `owner` account would allow the attacker to undermine necessary assumptions behind the protocol and subvert various protocol operations. In addition, the current `owner` account is also able to upgrade the current logic contracts behind the proxy.

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.

3.2 Possible Sandwich/MEV Attacks To Collect Most Rewards

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `KolStrategyThena`
- Category: Time and State [10]
- CWE subcategory: CWE-682 [5]

Description

The `KOL` protocol has designed a new `strategy` contract to invest user deposits (held in `KolStrategyThena`), harvest growing yields, and collect any gains, if any, to the share holders. In the meantime, we notice the protocol takes a different approach by directly rewarding the yields back to investors.

To elaborate, we show below the `_harvest()` function in `KolStrategyThena`. This routine essentially collects any pending rewards via `gauge::getReward()` (line 211) and then distributes the collected rewards evenly to current share holders.

```
210    function _harvest(address callFeeRecipient) internal whenNotPaused {
211        gauge.getReward();
212        uint256 outputBal = IERC20Upgradeable(output).balanceOf(address(this));
213        if (outputBal > 0) {
214            chargeFees(callFeeRecipient);
215            addLiquidity();
216            uint256 wantHarvested = balanceOfWant();
217            uint256 _toProtocol;
218            if (protocolFee>0) {
219                _toProtocol = wantHarvested * protocolFee / PERCENTAGE ;
```

```
                totalProtocolFee += _toProtocol;
                want.transfer(protocolReceiver, _toProtocol);
            }
            totalHarvested += wantHarvested - _toProtocol;

            deposit();

            lastHarvest = block.timestamp;
            emit StratHarvest(msg.sender, wantHarvested, balanceOf());
        }
    }
```

Listing 3.2: `BaseVault::_harvest()`

We notice the collected rewards are evenly distributed to share holders. With that, it is possible for a malicious actor to launch a flashloan-assisted deposit to claim the majority of rewards, resulting in significantly less rewards to legitimate share holders. This is possible as the `harvest()` may be triggered in a permissionless manner, allowing for a crafted contract to directly borrow a flashloan, deposit the borrowed loan into the vault pool, call `harvest()` to claim a majority share in rewards, and finally return the flashloan.

In the meantime, the current protocol supports the conversion of output token to others as liquidity. Because of that, there is a constant need of swapping one asset to another. With that, the protocol has provided a helper routine `addLiquidity()`.

```
    function addLiquidity() internal {
        uint256 outputBal = IERC20Upgradeable(output).balanceOf(address(this));
        uint256 lp0Amt = outputBal / 2;
        uint256 lp1Amt = outputBal - lp0Amt;
        ISolidlyRouter router = ISolidlyRouter(uniRouter);

        if (stable) {
            uint256 out0 = lp0Amt;
            if (lpToken0 != output) {
                out0 =
                (router.getAmountsOut(lp0Amt, outputToLp0Route)[
                outputToLp0Route.length
                ] * 1e18) /
                lp0Decimals;
            }

            uint256 out1 = lp1Amt;
            if (lpToken1 != output) {
                out1 =
                (router.getAmountsOut(lp1Amt, outputToLp1Route)[
                outputToLp1Route.length
                ] * 1e18) /
                lp1Decimals;
            }

            (uint256 amountA, uint256 amountB, ) = router.quoteAddLiquidity(
```

```
                lpToken0,
                lpToken1,
                stable,
                out0,
                out1
            );

            amountA = (amountA * 1e18) / lp0Decimals;
            amountB = (amountB * 1e18) / lp1Decimals;
            uint256 ratio = (((out0 * 1e18) / out1) * amountB) / amountA;
            lp0Amt = (outputBal * 1e18) / (ratio + 1e18);
            lp1Amt = outputBal - lp0Amt;
        }

        if (lpToken0 != output) {
            router.swapExactTokensForTokens(
                lp0Amt,
                0,
                outputToLp0Route,
                address(this),
                block.timestamp
            );
        }

        if (lpToken1 != output) {
            router.swapExactTokensForTokens(
                lp1Amt,
                0,
                outputToLp1Route,
                address(this),
                block.timestamp
            );
        }

        uint256 lp0Bal = IERC20Upgradeable(lpToken0).balanceOf(address(this));
        uint256 lp1Bal = IERC20Upgradeable(lpToken1).balanceOf(address(this));
        router.addLiquidity(
            lpToken0,
            lpToken1,
            stable,
            lp0Bal,
            lp1Bal,
            1,
            1,
            address(this),
            block.timestamp
        );
    }
```

Listing 3.3: `KolStrategyThena::addLiquidity()`

To elaborate, we show above the helper routine. We notice the conversion is routed to `Thena` router in order to swap one asset to another. 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 conversion.

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 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 sandwich attack, including the use of slippage control to the above front-running attack to better protect the interests of farming users.

Status The issue has been resolved as the team confirms the use of their own harvest contracts. In addition, it is planned to run through the `Flashbots` pool.

3.3 Improved Reentrancy Protection in KolAutomizerVault

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `KolAutomizerVault`
- Category: Time and State [9]
- 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` [15] exploit, and the `Uniswap/Lendf.Me` hack [14].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `KolAutomizerVault` as an example, the `deposit()` 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 172) starts before effecting the update on internal states (line 180), 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 `withdraw()` function.

```
165     function withdraw(uint256 _shares) public {
166         uint256 r = (balance()*(_shares))/(totalSupply());
167         _burn(msg.sender, _shares);
168 
169         uint b = want.balanceOf(address(this));
170         if (b < r) {
171             uint _withdraw = r-(b);
172             strategy.withdraw(_withdraw);
173             uint _after = want.balanceOf(address(this));
174             uint _diff = _after-(b);
175             if (_diff < _withdraw) {
176                 r = b+(_diff);
177             }
178         }
179         want.safeTransfer(msg.sender, r);
180         totalWithdrawn[msg.sender] +=r;
181         emit Withdraw(msg.sender, _shares, r);
182     }
```

Listing 3.4: KolAutomizerVault::withdraw()

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`.

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

Status This issue has been fixed in the following commit: `371a866`.

3.4 Possible Costly LPs From Improper Initialization

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `KolAutomizerVault`
- Category: Time and State [7]
- CWE subcategory: CWE-362 [3]

Description

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

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

```
101     function depositFor(address user, uint _amount) public nonReentrant {
102         //if (msg.sender == tx.origin)
103         strategy.beforeDeposit();
104 
105         uint256 _pool = balance();
106         want.safeTransferFrom(msg.sender, address(this), _amount);
107         earn();
108         uint256 _after = balance();
109         _amount = _after-(_pool); // Additional check for deflationary tokens
110         totalDeposit[user] += _amount;
111         uint256 shares = 0;
112         if (totalSupply() == 0) {
113             shares = _amount;
114         } else {
115             shares = (_amount*(totalSupply()))/(_pool);
116         }
117 
118         _mint(user, shares);
119         emit Deposit(msg.sender, shares, _amount);
120     }
```

Listing 3.5: KolAutomizerVault::depositFor()

Specifically, when the pool is being initialized (line 112), the share value directly takes the value of `_amount` (line 113), 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 USDT assets with the goal of making the pool token extremely expensive.

An extremely expensive 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.

This is a known issue that has been mitigated in popular Uniswap. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

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

Status This issue has been fixed in the following commit: `371a866`.

3.5 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `KolStrategyThena`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [1]

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 analyze 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 3.6: USDT Token **Contract**

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()`, 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 `transfer()` as well, i.e., `safeTransfer()`.

```
    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IERC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(
        IERC20 token,
        address spender,
        uint256 value
    ) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        require(
            (value == 0)  (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
            spender, value));
    }
```

Listing 3.7: `SafeERC20::safeApprove()`

In current implementation, if we examine the `BasisStrategy::_giveAllowances()` routine that is designed to approve an authorized spender. To accommodate the specific idiosyncrasy, there is a need to use `safeApprover()`, instead of `approve()` (lines 406-407 and 415-416).

```
    function _giveAllowances() internal {
        address _uniRouter = uniRouter;
        want.approve(address(gauge), type(uint).max);
        IERC20Upgradeable(output).approve(_uniRouter, type(uint).max);
        IERC20Upgradeable(lpToken0).approve(_uniRouter, type(uint).max);
        IERC20Upgradeable(lpToken1).approve(_uniRouter, type(uint).max);
        emit GiveAllowances();
    }

    function _removeAllowances() internal {
        address _uniRouter = uniRouter;
        want.approve(address(gauge), 0);
        IERC20Upgradeable(output).approve(_uniRouter, 0);
        IERC20Upgradeable(lpToken0).approve(_uniRouter, 0);
        IERC20Upgradeable(lpToken1).approve(_uniRouter, 0);
        emit RemoveAllowances();
```

```
    }
```

Listing 3.8: `BasisStrategy::_giveAllowances()/removeAllowances()`

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

Status This issue has been fixed in the following commit: `371a866`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Key of Life Finance` protocol. The audited system is a decentralized, multichain automizer vaults that are designed to earn compound interest on their crypto holdings. The purpose is to maximize the highest APYs with safety and efficiency. As a result, the protocol will help to optimize asset and resources allocations for DeFi community overall. The current code base is clearly 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] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.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-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 CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[7] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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

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

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