SMART CONTRACT AUDIT REPORT

for

KALMAR PROTOCOL

Prepared By: Yiqun Chen

PeckShield
August 20, 2021

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	August 20, 2021	Xuxian Jiang	Final Release
1.0-rc	August 7, 2021	Xuxian Jiang	Release Candidate

Contact

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

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

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the the `Kalmar` 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 Kalmar Protocol

`Kalmar` is a decentralized bank powered by DeFi and NFT. The protocol uses secure financial instruments and advanced gamification models to make banking engaging, transparent and accessible. The audited `leverage-yield-contracts/leverage-yield-contracts-busd` are designed as an evolutional improvement of `Alpha`, which is a leveraged yield farming and leveraged liquidity providing protocol launched on the `Ethereum`. The audited implementation makes improvements, including the direct integration of mining support at the protocol level as well as the customizability of base tokens (instead of native tokens).

The basic information of the `Kalmar` protocol is as follows:

Table 1.1: Basic Information of Kalmar Protocol

Item	Description
Issuer	Kalmar Protocol
Website	https://kalmar.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 20, 2021

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

- https://github.com/kalmar-io/leverage-yield-contracts.git (ad08aef)
- https://github.com/kalmar-io/leverage-yield-contracts-busd.git (1fd562e)

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

- https://github.com/kalmar-io/leverage-yield-contracts.git (a0f5299)
- https://github.com/kalmar-io/leverage-yield-contracts-busd.git (5436dda)

1.2 About PeckShield

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

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

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

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;

- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

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

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

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

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

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

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered

Table 1.3: The Full List of Check Items

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

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

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

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.

2 Findings

2.1 Summary

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

Severity	# of Findings	
Critical	0	
High	0	
Medium	3	■■■
Low	3	■■■
Informational	1	■
Total	7	

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 3 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings of Kalmar Protocol Protocol

ID	Severity	Title	Category	Status
PVE-001	Medium	Possible Costly LPs From Improper Vault Initialization	Time and State	Resolved
PVE-002	Medium	Trading Fee Discrepancy Between Kalmar And PancakeSwap	Business Logic	Resolved
PVE-003	Low	No payable in All Four Strategies	Coding Practices	Resolved
PVE-004	Informational	Inconsistency Between Document and Implementation	Coding Practices	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Business Logic	Confirmed
PVE-006	Low	Potential Sandwich Attacks For Reduced Returns	Time and State	Resolved
PVE-007	Low	Improved Precision By Multiplication And Division Reordering	Numeric Errors	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 Possible Costly LPs From Improper Bank Initialization

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Bank
- Category: Time and State [9]
- CWE subcategory: CWE-362 [5]

Description

In the Kalmar protocol, the Bank contract is an essential one that manages current debt positions and mediates the access to various workers (or Goblins). Meanwhile, the Bank contract allows liquidity providers to provide liquidity so that lenders can earn high interest and the lending interest rate comes from leveraged yield farmers. While examining the share calculation when lenders provide liquidity (via deposit()), we notice an issue that may unnecessarily make the Bank-related pool token extremely expensive and bring hurdles (or even causes loss) for later liquidity providers.

To elaborate, we show below the deposit() routine. This routine is used for liquidity providers to deposit desired liquidity 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.

```
    /// @dev Add more ETH to the bank. Hope to get some good returns.
    function deposit() external payable accrue(msg.value) nonReentrant {
        uint256 total = totalETH().sub(msg.value);
        uint256 share = total == 0 ? msg.value : msg.value.mul(totalSupply()).div(total)
            ;
        _mint(msg.sender, share);
    }

    /// @dev Withdraw ETH from the bank by burning the share tokens.
    function withdraw(uint256 share) external accrue(0) nonReentrant {
        uint256 amount = share.mul(totalETH()).div(totalSupply());
        _burn(msg.sender, share);
        SafeToken.safeTransferETH(msg.sender, amount);
```

```
115     }
```

Listing 3.1: `Bank::deposit()/withdraw()`

Specifically, when the pool is being initialized, the share value directly takes the given value of `msg.value` (line 106), which is under control by the malicious actor. As this is the first deposit, the current total supply equals the calculated `share = total == 0 ? msg.value : msg.value.mul(totalSupply()).div(total)= 1WEI`. After that, the actor can further transfer a huge amount of tokens 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 `UniswapV2`. 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 stake provider, but this cost is expected to be low and acceptable. Another alternative requires a guarded launch to ensure the pool is always initialized properly.

Recommendation Revise current execution logic of `deposit()` to defensively calculate the share amount when the pool is being initialized.

Status This issue has been fixed by requiring a minimal share in the `Bank` by the following commit: `a0f5299`.

3.2 Trading Fee Discrepancy Between Kalmar And PancakeSwap

- ID: PVE-002
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Business Logic [11]
- CWE subcategory: CWE-841 [7]

Description

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

To elaborate, we show below the getAmountOut() routine inside the the UniswapV2Library. For comparison, we also show the getMktSellAmount() routine in MasterChefGoblin. It is interesting to note that MasterChefGoblin has implicitly assumed the trading fee is 0.3%, instead of 0.25%. The difference in the built-in trading fee may skew the optimal allocation of assets in the developed strategies (e.g., StrategyAddETHOnly and StrategyAddTwoSidesOptimal) and other contracts (e.g., MasterChefPoolRewardPairGoblin and MasterChefPoolRewardPairGoblin).

```
// given an input amount of an asset and pair reserves, returns the maximum output amount of the other asset
function getAmountOut(
  uint256 amountIn,
  uint256 reserveIn,
  uint256 reserveOut
) internal pure returns (uint256 amountOut) {
  require(amountIn > 0, 'UniswapV2Library: INSUFFICIENT_INPUT_AMOUNT');
  require(reserveIn > 0 && reserveOut > 0, 'UniswapV2Library: INSUFFICIENT_LIQUIDITY');
  uint256 amountInWithFee = amountIn.mul(997);
  uint256 numerator = amountInWithFee.mul(reserveOut);
  uint256 denominator = reserveIn.mul(1000).add(amountInWithFee);
  amountOut = numerator / denominator;
}
```

Listing 3.2: UniswapV2Library::getAmountOut()

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

Listing 3.3: MasterChefGoblin::getMktSellAmount()

Recommendation Make the built-in trading fee in Kalmar consistent with the actual trading fee in PancakeSwap.

Status This issue has been fixed by the following commit: `a0f5299`.

3.3 No payable in All Four Strategies

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [10]
- CWE subcategory: CWE-1126 [2]

Description

The `Kalmar` protocol has four built-in strategies `StrategyAllBaseTokenOnly`, `StrategyAddTwoSidesOptimal`, `StrategyLiquidate`, and `StrategyWithdrawMinimizeTrading`. The first strategy allows for adding native tokens for farming, the second strategy supports the addition of both base tokens and farming tokens, the third strategy is designed to liquidate the farming tokens back to base tokens, and the four strategy enables the withdrawal with minimized trading. These four strategies define their own `execute()` routines to perform respective tasks.

To elaborate, we show below the `execute()` function from the first strategy. It comes to our attention that this function comes with the `payable` keyword. However, the implementation does not handle any native tokens. Note this is the same for all the above four strategies in the `leverage-yield-contracts-busd` repository, but not in the `leverage-yield-contracts` repository.

```
    function execute(address /* user */, uint256 /* debt */, bytes calldata data)
        external
        payable
        nonReentrant
    {
        // 1. Find out what farming token we are dealing with and min additional LP tokens.
        (address baseToken, address fToken, uint256 minLPAmount) = abi.decode(data, (address, address, uint256));
        IUniswapV2Pair lpToken = IUniswapV2Pair(factory.getPair(fToken, baseToken));
        // 2. Compute the optimal amount of ETH to be converted to farming tokens.
        uint256 balance = baseToken.myBalance();
        (uint256 r0, uint256 r1, ) = lpToken.getReserves();
        uint256 rIn = lpToken.token0() == baseToken ? r0 : r1;
        uint256 aIn = Math.sqrt(rIn.mul(balance.mul(3988000).add(rIn.mul(3988009)))).sub(rIn.mul(1997)) / 1994;
        // 3. Convert that portion of ETH to farming tokens.
        address[] memory path = new address[](2);
        path[0] = baseToken;
        path[1] = fToken;
        baseToken.safeApprove(address(router), 0);
        baseToken.safeApprove(address(router), uint(-1));
```

```
48          router.swapExactTokensForTokens(aIn, 0, path, address(this), now);
49          // 4. Mint more LP tokens and return all LP tokens to the sender.
50          fToken.safeApprove(address(router), 0);
51          fToken.safeApprove(address(router), uint(-1));
52          (,, uint256 moreLPAmount) = router.addLiquidity(
53              baseToken, fToken, baseToken.myBalance(), fToken.myBalance(), 0, 0, address(
                    this), now
54          );
55          require(moreLPAmount >= minLPAmount, "insufficient LP tokens received");
56          lpToken.transfer(msg.sender, lpToken.balanceOf(address(this)));
57      }
```

Listing 3.4: `StrategyAllBaseTokenOnly::execute()`

Recommendation Remove the unnecessary `payable` in the four strategies from the `leverage-yield-contracts-busd` repository.

Status This issue has been fixed by the following commit: `a0f5299`.

3.4 Inconsistency Between Document and Implementation

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [10]
- CWE subcategory: CWE-1041 [1]

Description

There are a few misleading comments embedded among lines of solidity code, which bring unnecessary hurdles to understand and/or maintain the software.

A few example comments can be found in various `execute()` routines scattered in different contacts, e.g., line 27 of `StrategyAllBaseTokenOnly`, line 86 of `StrategyAddTwoSidesOptimal`, and line 27 of `StrategyWithdrawMinimizeTrading`. Using the `StrategyAllBaseTokenOnly::execute()` routine as an example, the preceding function summary indicates that this routine expects to "*Take LP tokens + ETH. Return LP tokens + ETH.*." However, our analysis shows that it only takes base tokens and returns `LP tokens` back to the sender.

```
29      function execute(address /* user */, uint256 /* debt */, bytes calldata data)
30          external
31          payable
32          nonReentrant
33      {
34          // 1. Find out what farming token we are dealing with and min additional LP
                tokens.
```

```
        (address baseToken, address fToken, uint256 minLPAmount) = abi.decode(data, (address, address, uint256));
        IUniswapV2Pair lpToken = IUniswapV2Pair(factory.getPair(fToken, baseToken));
        // 2. Compute the optimal amount of ETH to be converted to farming tokens.
        uint256 balance = baseToken.myBalance();
        (uint256 r0, uint256 r1, ) = lpToken.getReserves();
        uint256 rIn = lpToken.token0() == baseToken ? r0 : r1;
        uint256 aIn = Math.sqrt(rIn.mul(balance.mul(3988000).add(rIn.mul(3988009)))).sub(rIn.mul(1997)) / 1994;
        // 3. Convert that portion of ETH to farming tokens.
        address[] memory path = new address[](2);
        path[0] = baseToken;
        path[1] = fToken;
        baseToken.safeApprove(address(router), 0);
        baseToken.safeApprove(address(router), uint(-1));
        router.swapExactTokensForTokens(aIn, 0, path, address(this), now);
        // 4. Mint more LP tokens and return all LP tokens to the sender.
        fToken.safeApprove(address(router), 0);
        fToken.safeApprove(address(router), uint(-1));
        (,, uint256 moreLPAmount) = router.addLiquidity(
            baseToken, fToken, baseToken.myBalance(), fToken.myBalance(), 0, 0, address(this), now
        );
        require(moreLPAmount >= minLPAmount, "insufficient LP tokens received");
        lpToken.transfer(msg.sender, lpToken.balanceOf(address(this)));
    }
```

Listing 3.5: `StrategyAllBaseTokenOnly::execute()`

Note that the `StrategyLiquidate::execute()` routine takes `LP tokens` and returns base tokens; the `StrategyAddTwoSidesOptimal::execute()` routine takes base and `fToken` tokens and returns `LP tokens`; while the `StrategyWithdrawMinimizeTrading::execute()` routine takes `LP tokens` and returns base and `fToken` tokens.

Recommendation Ensure the consistency between documents (including embedded comments) and implementation.

Status This issue has been fixed by the following commit: `a0f5299`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [8]
- CWE subcategory: CWE-287 [4]

Description

In the `Kalmar` protocol, all debt positions are managed by the `Bank` contract. And there is a privileged account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and strategy adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

To elaborate, we show below the `kill()` routine in the `Bank` contract. This routine allows anyone to liquidate the given position assuming it is underwater and available for liquidation. There is a key factor, i.e., `killFactor`, that greatly affects the decision on whether the position can be liquidated (line 193). Note that `killFactor` is a risk parameter that can be dynamically configured by the privileged owner.

```
186     function kill(uint256 id) external onlyEOA accrue(0) nonReentrant {
187         // 1. Verify that the position is eligible for liquidation.
188         Position storage pos = positions[id];
189         require(pos.debtShare > 0, "no debt");
190         uint256 debt = _removeDebt(id);
191         uint256 health = Goblin(pos.goblin).health(id);
192         uint256 killFactor = config.killFactor(pos.goblin, debt);
193         require(health.mul(killFactor) < debt.mul(10000), "can't liquidate");
194         // 2. Perform liquidation and compute the amount of ETH received.
195         uint256 beforeETH = token.myBalance();
196         Goblin(pos.goblin).liquidate(id);
197         uint256 back = token.myBalance().sub(beforeETH);
198         uint256 prize = back.mul(config.getKillBps()).div(10000);
199         uint256 rest = back.sub(prize);
200         // 3. Clear position debt and return funds to liquidator and position owner.
201         if (prize > 0) {
202             address rewardTo = killBpsToTreasury == true ? treasuryAddr : msg.sender;
203             SafeToken.safeTransfer(token, rewardTo, prize);
204         }
205         uint256 left = rest > debt ? rest - debt : 0;
206         if (left > 0) SafeToken.safeTransfer(token, pos.owner, left);
207         emit Kill(id, msg.sender, prize, left);
```

```
    }
```

Listing 3.6: `Vault::kill()`

Also, if we examine the privileged function on available `MasterChefGoblin`, i.e., `setCriticalStrategies()`, this routine allows the update of new strategies to work on a user's position. It has been highlighted that bad strategies can steal user funds. Note that this privileged function is guarded with `onlyOwner`.

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

Listing 3.7: `MasterChefGoblin::setCriticalStrategies()`

It is worrisome if the privileged `owner` account is a plain EOA account. The discussion with the team confirms that the `owner` account is currently managed by a timelock. A plan needs to be in place to migrate it under community governance. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

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

Status This issue has been confirmed with the team.

3.6 Potential Sandwich Attacks For Reduced Returns

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Time and State [12]
- CWE subcategory: CWE-682 [6]

Description

As a yield farming and leveraged liquidity providing protocol, `Kalmar` has a constant need of performing token swaps between base and farming tokens. In the following, we examine the re-investment logic from the new `MasterChefGoblin` contract.

To elaborate, we show below the `reinvest()` implementation. As the name indicates, it is designed to re-invest whatever this worker has earned to the staking pool. In the meantime, the caller will be incentivized with reward bounty based on the risk parameter `reinvestBountyBps` and a portion of the reward bounty will be sent to `reinvestToTreasury` to increase the size.

```
    /// @dev Re-invest whatever this worker has earned back to staked LP tokens.
    function reinvest() public onlyEOA nonReentrant {
        // 1. Withdraw all the rewards.
        masterChef.withdraw(pid, 0);
        uint256 reward = rewardToken.balanceOf(address(this));
        if (reward == 0) return;
        // 2. Send the reward bounty to the caller or Owner.
        uint256 bounty = reward.mul(reinvestBountyBps) / 10000;

        address rewardTo = reinvestToTreasury == true ? treasuryAddr : msg.sender;

        rewardToken.safeTransfer(rewardTo, bounty);
        // 3. Convert all the remaining rewards to ETH.
        address[] memory path = new address[](3);
        path[0] = address(rewardToken);
        path[1] = address(wbnb);
        path[2] = address(baseToken);
        router.swapExactTokensForTokens(reward.sub(bounty), 0, path, address(this), now)
            ;
        // 4. Use add ETH strategy to convert all ETH to LP tokens.
        baseToken.safeTransfer(address(addStrat), baseToken.myBalance());
        addStrat.execute(address(0), 0, abi.encode(baseToken, fToken, 0));
        // 5. Mint more LP tokens and stake them for more rewards.
        masterChef.deposit(pid, lpToken.balanceOf(address(this)));
        emit Reinvest(msg.sender, reward, bounty);
    }
```

Listing 3.8: `MasterChefGoblin::reinvest()`

We notice the remaining rewards are routed to `pancakeSwap` and the actual swap operation `swapExactTokensForTokens()` does not specify any restriction (with `amountOutMin=0`) on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of yielding.

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 front-running attack to better protect the interests of farming users.

Status The issue has been confirmed. Moreover, according to the discussion with the development team, the funds to be used to swap when `reinvest()` is called is not large. Hence, the risk is in the acceptable range.

3.7 Improved Precision By Multiplication And Division Reordering

- ID: PVE-007
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `CakeMaxiWorkerConfig`
- Category: Numeric Errors [13]
- CWE subcategory: CWE-190 [3]

Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, the lack of `float` support in `Solidity` may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (`mul`) and division (`div`) are involved.

In particular, we use the `MasterChefGoblinConfig::isStable()` as an example. This routine is used to measure the stability of the given worker and prevent it from being manipulated.

```
51      /// @dev Return whether the given goblin is stable, presumably not under
            manipulation.
52      function isStable(address goblin) public view returns (bool) {
53          IUniswapV2Pair lp = IMasterChefGoblin(goblin).lpToken();
54          address token0 = lp.token0();
55          address token1 = lp.token1();
56          // 1. Check that reserves and balances are consistent (within 1%)
57          (uint256 r0, uint256 r1,) = lp.getReserves();
58          uint256 t0bal = token0.balanceOf(address(lp));
59          uint256 t1bal = token1.balanceOf(address(lp));
60          require(t0bal.mul(100) <= r0.mul(101), "bad t0 balance");
61          require(t1bal.mul(100) <= r1.mul(101), "bad t1 balance");
62          // 2. Check that price is in the acceptable range
63          (uint256 price, uint256 lastUpdate) = oracle.getPrice(token0, token1);
64          require(lastUpdate >= now - 7 days, "price too stale");
65          uint256 lpPrice = r1.mul(1e18).div(r0);
66          uint256 maxPriceDiff = goblins[goblin].maxPriceDiff;
67          require(lpPrice <= price.mul(maxPriceDiff).div(10000), "price too high");
68          require(lpPrice >= price.mul(10000).div(maxPriceDiff), "price too low");
69          // 3. Done
70          return true;
71      }
```

Listing 3.9: `MasterChefGoblinConfig::isStable()`

We notice the comparison between the `lpPrice` and the external oracle price (lines 67−68) involves mixed multiplication and devision. For improved precision, it is better to calculate the multiplication before the division, i.e., `require(lpPrice.mul(10000)<= price.mul(maxPriceDiff))`, instead of current `require(lpPrice <= price.mul(maxPriceDiff).div(10000))` (line 67). Note that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible.

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status This issue has been fixed by the following commit: `a0f5299`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Kalmar` protocol, which is a decentralized bank powered by DeFi and NFT. The audited implementation is a leveraged-yield farming protocol built on the `BSC` with an initial fork from `Alpha`. The system continues the innovative design and makes it distinctive and valuable when compared with current yield farming offerings. The current code base is well organized and those identified issues are promptly confirmed and fixed.

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

References

[1] 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-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

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

[6] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

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

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

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

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

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

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

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

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