



SMART CONTRACT AUDIT REPORT

for

Shoebill



Prepared By: Xiaomi Huang

PeckShield
February 11, 2023

Document Properties

Client	Shoebill
Title	Smart Contract Audit Report
Target	Shoebill
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0-rc	January 28, 2023	Xuxian Jiang	Release Candidate

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	4
1.1	About Shoebill	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	7
2	Findings	9
2.1	Summary	9
2.2	Key Findings	10
3	Detailed Results	11
3.1	Improved Reward-Claiming Logic in StakedTokenIncentivesController	11
3.2	Enhanced Asset Validation in LenderVaults	13
3.3	Incorrect Yield Accounting in KokoaKSDVault And KlayswapUsdtUsdcVault	14
3.4	Trust Issue of Admin Keys	15
3.5	Potential Misuse of Borrow Allowance in LenderVault	17
3.6	Potential Front-Running/MEV With Reduced Return	18
3.7	Unsafe Listing of KlayswapUsdtUsdc LPs	19
4	Conclusion	21
	References	22

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the `Shoebill` 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 Shoebill

`Shoebill` is a decentralized non-custodial liquidity markets protocol that is developed on top of `AAVE`. On existing lending protocols, the interest earned by lenders comes from borrowers. `Shoebill` uses a different model, where yield instead comes from the borrowers' collateral. Specifically, when borrowers provide a token as collateral, `Shoebill` converts it into an interest-bearing token (`ibToken`) using protocols like `Yearn` or `Lido`. Over time, these `ibTokens` accrue yield and the yield from these tokens is then distributed to lenders in the same token they deposited. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Shoebill

Item	Description
Target	Shoebill
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	February 11, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that `Shoebill` assumes a trusted price oracle with timely market price feeds for

supported assets and the oracle itself is not part of this audit.

- <https://github.com/ShoebillFinance/shoebill-contract.git> (f46d48a)

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

- <https://github.com/ShoebillFinance/shoebill-contract.git> (2dbcd9f6)

1.2 About PeckShield

PeckShield Inc. [10] 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	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [9]:

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

Table 1.3: The Full 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

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) [8], 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 `Shoebill` 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	2	
Medium	2	
Low	3	
Informational	0	
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 2 high-severity vulnerabilities, 2 medium-severity vulnerabilities, and 3 low-severity vulnerabilities.

Table 2.1: Key Shoebill Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Reward-Claiming Logic in StakedTokenIncentivesController	Business Logic	Resolved
PVE-002	Low	Enhanced Asset Validation in LenderVaults	Business Logic	Resolved
PVE-003	Low	Incorrect Yield Accounting in KokoaKSD-Vault And KlayswapUsdtUsdcVault	Time and State	Resolved
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-005	High	Potential Misuse of Borrow Allowance in LenderVault	Business Logic	Resolved
PVE-006	Medium	Possible Sandwich/MEV Attacks For Reduced Returns	Time and State	Resolved
PVE-007	High	Unsafe Listing of KlayswapUsdtUsdc LPs	Time and State	Resolved

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 Improved Reward-Claiming Logic in StakedTokenIncentivesController

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: StakedTokenIncentivesController
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

The Shoebill protocol has a built-in incentive contract `StakedTokenIncentivesController`, which manages the distribution of protocol incentives. While reviewing its logic, we notice the current reward-claiming implementation needs to be improved.

To elaborate, we show the related function `_claimRewards()`. As the name indicates, this function is used to claim rewards from the protocol engagement with contribution on liquidity or interest payment. However, it comes to our attention that the remaining rewards recorded in an internal `usersUnclaimedRewards` variable (line 238) will be updated no matter whether the rewards are successfully claimed or not. Also, the `RewardsClaimed` event is always emitted no matter whether the claim is successful or not.

```
208 function _claimRewards(  
209     address[] calldata assets,  
210     uint256 amount,  
211     address claimer,  
212     address user,  
213     address to  
214 ) internal returns (uint256) {  
215     if (amount == 0) {  
216         return 0;  
217     }  
218     uint256 unclaimedRewards = _usersUnclaimedRewards[user];  
219     uint256 length = assets.length;
```

```

220     DistributionTypes.UserStakeInput[] memory userState = new DistributionTypes.
        UserStakeInput[](length);
221     for (uint256 i; i < length; ++i) {
222         userState[i].underlyingAsset = assets[i];
223         (userState[i].stakedByUser, userState[i].totalStaked) = IScaledBalanceToken(assets
            [i])
224             .getScaledUserBalanceAndSupply(user);
225     }
226
227     uint256 accruedRewards = _claimRewards(user, userState);
228     if (accruedRewards > 0) {
229         unclaimedRewards += accruedRewards;
230         emit RewardsAccrued(user, accruedRewards);
231     }
232
233     if (unclaimedRewards == 0) {
234         return 0;
235     }
236
237     uint256 amountToClaim = amount > unclaimedRewards ? unclaimedRewards : amount;
238     _usersUnclaimedRewards[user] = unclaimedRewards - amountToClaim; // Safe due to the
        previous line
239
240     // STAKE_TOKEN.stake(to, amountToClaim);
241     IERC20 stakeToken = IERC20(_addressProvider.getIncentiveToken());
242     if (stakeToken.balanceOf(address(this)) >= amountToClaim) {
243         stakeToken.safeTransfer(to, amountToClaim);
244     }
245
246     emit RewardsClaimed(user, to, claimer, amountToClaim);
247
248     return amountToClaim;
249 }

```

Listing 3.1: StakedTokenIncentivesController::_claimRewards()

Recommendation Revise the above routine in updating the remaining rewards (or emitting RewardsClaimed) only when rewards are successfully claimed.

Status This issue has been addressed in the following commit: 2dbcd9f.

3.2 Enhanced Asset Validation in LenderVaults

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: LenderVaults
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

The Shoebill protocol has a LenderVaults contract that accepts user deposits and interacts directly with the underlying lending protocol. For each supported asset, the LenderVaults contract will instantiate a new internalAssetToken counterpart, which is paired with the supported asset or externalAssetToken. The internalAssetToken will then be used as the asset deposited into the lending protocol. However, the LenderVaults contract has a number of exported functions that can be improved to validate the input asset.

In particular, if we examine the deposit() function, which accepts the asset as the first argument and passes directly into the internal helper _depositToYieldPool(). However, this argument is not validated to ensure it is the same as externalAssetToken. If the external contract ITokenKlayswapSingle may not revert on the depositKct() call, it will create a false deposit issue to the protocol.

```

78  function deposit(address _reserve, uint256 _amount) external virtual {
79      _updateLiquidity();
80      uint256 _internalAssetAmount = _depositToYieldPool(_reserve, _amount);
81      ILendingPool(_addressesProvider.getLendingPool()).deposit(internalAssetToken,
      _internalAssetAmount, msg.sender, 0);
82  }

```

Listing 3.2: LenderVaults::deposit()

```

62  function _depositToYieldPool(address _asset, uint256 _amount) internal override
      returns (uint256) {
63      IERC20(_asset).safeTransferFrom(msg.sender, address(this), _amount);
64      IERC20(_asset).safeApprove(klayswapIToken, 0);
65      IERC20(_asset).safeApprove(klayswapIToken, _amount);
66      IITokenKlayswapSingle(klayswapIToken).depositKct(_amount);
67      ShoebillInternalAsset(internalAssetToken).mint(address(this), _amount);
68      address lendingPool = _addressesProvider.getLendingPool();
69      IERC20(internalAssetToken).safeApprove(lendingPool, 0);
70      IERC20(internalAssetToken).safeApprove(lendingPool, _amount);
71      return _amount;
72  }

```

Listing 3.3: DAILendVault::_depositToYieldPool()

Recommendation Validate the input asset to ensure it is the supported `externalAssetToken`. The same issue is also applicable to the `withdraw()` function.

Status This issue has been addressed in the following commit: `2dbcd9f`.

3.3 Incorrect Yield Accounting in KokoaKSDVault And KlayswapUsdtUsdcVault

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

The Shoebill protocol has a number of built-in vaults. While examining two specific ones, i.e., `KokoaKSDVault` and `KlayswapUsdtUsdcVault`, we notice the current yield attribution logic is flawed and should be corrected.

In the following, we show the code snippet of `_transferYield()` which transfers the yields back to the involved parties, e.g., `yieldDistributor`, `treasury`, and `yieldManager`. Note the attribution to `yieldDistributor` only occurs when the yield asset is equal to the intended incentive token (line 61). However, the `yieldAmount` is always deducted if there is a non-zero `incentiveAmount` (lines 71-73)! To correct, we need to ensure the `incentiveAmount`-related deduction only occurs when `isIncentiveToken` is true as well.

```

53  function _transferYield(address _asset) internal {
54      require(_asset != address(0), Errors.VT_PROCESS_YIELD_INVALID);
55      require(_asset != dKSD, Errors.VT_PROCESS_YIELD_INVALID);
56      uint256 yieldAmount = IERC20(_asset).balanceOf(address(this));
57      if (yieldAmount == 0) return;
58      uint256 incentiveAmount;
59      uint256 fee = _incentiveRatio;
60      bool isIncentiveToken = (getIncentiveToken() == _asset);
61      if (isIncentiveToken && fee != 0) {
62          incentiveAmount = yieldAmount.percentMul(fee);
63          _sendIncentive(incentiveAmount);
64      }
65      fee = _vaultFee;
66      if (fee != 0) {
67          uint256 treasuryAmount = yieldAmount.percentMul(fee);
68          IERC20(_asset).safeTransfer(_treasuryAddress, treasuryAmount);
69          yieldAmount -= treasuryAmount;
70      }

```

```

71     if (incentiveAmount > 0) {
72         yieldAmount -= incentiveAmount;
73     }
74     if (yieldAmount > 0) {
75         address yieldManager = _addressesProvider.getAddress("YIELD_MANAGER");
76         IERC20(_asset).safeTransfer(yieldManager, yieldAmount);
77     }
78     emit ProcessYield(_asset, yieldAmount);
79 }

```

Listing 3.4: KokoakSDVault::_transferYield()

Recommendation Properly transfer yields to involved parties. Note the issue is applicable to both vaults: KokoakSDVault and KlayswapUsdtUsdcVault.

Status This issue has been addressed in the following commit: 2dbcd9f.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [1]

Description

In the Shoebill protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and vault management). Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```

47     function registerVault(address _vaultAddress) external payable override onlyPoolAdmin
48     {
49         pool.registerVault(_vaultAddress);
50     }
51     function batchInitReserve(InitReserveInput[] calldata input) external payable override
52         onlyPoolAdmin {
53         ILendingPool cachedPool = pool;
54         uint256 length = input.length;
55         for (uint256 i; i < length; ++i) {
56             _initReserve(cachedPool, input[i]);
57         }
58     }

```

Listing 3.5: Example Privileged Functions in LendingPoolConfigurator

```

25  function setStrategy(
26      address _iToken,
27      address _rewardToken,
28      address _swapper,
29      address[] calldata _pathForSwap
30  ) external onlyAdmin {
31      klayswapIToken = _iToken;
32      rewardToken = _rewardToken;
33      swapper = _swapper;
34      pathForSwap = _pathForSwap; // []
35  }

```

Listing 3.6: Example Privileged Functions in DAILendVault

```

26  function setConfiguration(address _lpToken) external payable onlyAdmin {
27      require(_lpToken != address(0), Errors.VT_INVALID_CONFIGURATION);
28      require(internalAssetToken == address(0), Errors.VT_INVALID_CONFIGURATION);
29      externalAssetToken = _lpToken;
30      ShoebillInternalAsset _internalToken = new ShoebillInternalAsset(
31          string(abi.encodePacked("shoebill ", IERC20Detailed(_lpToken).symbol())),
32          string(abi.encodePacked("sb", IERC20Detailed(_lpToken).symbol())),
33          IERC20Detailed(_lpToken).decimals(),
34          _lpToken
35      );
36      internalAssetToken = address(_internalToken);
37      emit SetParameters(_lpToken, internalAssetToken);
38  }
39
40  function registerLevSwap(address _levSwap) external onlyAdmin {
41      registeredLevSwap[_levSwap] = true;
42  }

```

Listing 3.7: Example Privileged Functions in KlayswapUsdtUsdcVault

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

3.5 Potential Misuse of Borrow Allowance in LenderVault

- ID: PVE-005
- Severity: High
- Likelihood: High
- Impact: High
- Target: LenderVault
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

As mentioned, earlier, the Shoebill protocol has a LenderVaults contract that accepts user deposits and interacts directly with the underlying lending protocol. While examining the borrow-related operation, we notice the vault supports the `onBehalfOf` feature that requires the borrowing users to pre-approve the borrow allowance. Our analysis shows this feature may be misused to borrow on behalf of other approving innocent users.

In particular, if we show below the borrow-related implementation of `borrow()`, we notice the last argument is `onBehalfOf`, which is directly provided by the user. Since the LenderVaults contract interacts directly with the lending protocol, from the lending protocol perspective, it will consider the user approves the borrow on the LenderVaults contract, not the calling user. As a result, a malicious actor may abuse the trust to directly borrow funds from other approving users, who may not approve on the malicious actor!

```

105     function borrow(
106         address _external,
107         uint256 _amount,
108         uint256 interestRateMode, // 2 = variable
109         uint16 referralCode, // 0 default
110         address onBehalfOf
111     ) external virtual {
112         _updateLiquidity();
113         uint256 _internalAssetAmount = _getInternalAmount(_external, _amount);
114         uint256 beforeBorrow = IERC20(internalAssetToken).balanceOf(address(this));
115         ILendingPool(_addressesProvider.getLendingPool()).borrow(
116             internalAssetToken,
117             _internalAssetAmount,
118             interestRateMode,
119             referralCode,
120             onBehalfOf
121         );
122         uint256 afterBorrow = IERC20(internalAssetToken).balanceOf(address(this));
123         uint256 _amountToWithdraw = afterBorrow - beforeBorrow;
124         uint256 withdrawAmount = _withdrawFromYieldPool(_external, _amountToWithdraw,
125             onBehalfOf);
126         require(withdrawAmount >= _amount, Errors.VT_WITHDRAW_AMOUNT_MISMATCH);

```

126 }

Listing 3.8: LenderVault::borrow()

Recommendation Revisit the above approval issue to avoid being misused.

Status This issue has been addressed in the following commit: 2dbcd9f.

3.6 Potential Front-Running/MEV With Reduced Return

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Time and State [7]
- CWE subcategory: CWE-682 [2]

Description

As mentioned earlier, the Shoebill protocol uses a different yield model, where yield comes from the borrowers' collateral. Specifically, when borrowers provide a token as collateral, Shoebill converts it into an interest-bearing token (ibToken) using protocols like Yearn or Lido. Over time, these ibTokens accrue yield and the yield from these tokens is then distributed to lenders in the same token they deposited. Because of that, there is a constant need of swapping one asset to another. However, our analysis shows the current conversion does not enforce meaningful slippage control.

```

37  function _processYield() internal override returns (uint256) {
38      IITokenKlayswapSingle(klayswapIToken).claimReward();
39      uint256 rewardBalance = IERC20(rewardToken).balanceOf(address(this));
40      IERC20(rewardToken).safeApprove(swapper, 0);
41      IERC20(rewardToken).safeApprove(swapper, rewardBalance);
42      uint256 beforeBal = IERC20(externalAssetToken).balanceOf(address(this));
43      IKSP(swapper).exchangeKctPos(rewardToken, rewardBalance, externalAssetToken, 1,
44          pathForSwap);
45      uint256 afterBal = IERC20(externalAssetToken).balanceOf(address(this));
46      uint256 swapAmount = afterBal - beforeBal;
47      if (_vaultFee != 0) {
48          uint256 treasuryAmount = swapAmount.percentMul(_vaultFee);
49          IERC20(externalAssetToken).safeTransfer(_treasuryAddress, treasuryAmount);
50          swapAmount -= treasuryAmount;
51      }
52      uint256 amountToMint = _calcIncreasedBalance();
53      ShoebillInternalAsset(internalAssetToken).mint(address(this), amountToMint);
54      IERC20(externalAssetToken).safeApprove(klayswapIToken, 0);
55      IERC20(externalAssetToken).safeApprove(klayswapIToken, swapAmount);
56      IITokenKlayswapSingle(klayswapIToken).depositKct(swapAmount);
57      address lendingPool = _addressesProvider.getLendingPool();

```

```

57     IERC20(internalAssetToken).safeApprove(lendingPool, 0);
58     IERC20(internalAssetToken).safeApprove(lendingPool, amountToMint);
59     return amountToMint;
60 }

```

Listing 3.9: USDClendVault::_processYield()

To elaborate, we show above one example routine `_processYield()`. We notice the conversion is routed to an external `swapper` 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 (e.g., slippage control) to the above front-running attack to better protect the interests of farming users.

Status This issue has been addressed in the following commit: [2abcd9f](#).

3.7 Unsafe Listing of KlayswapUsdtUsdc LPs

- ID: PVE-007
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `KlayswapUsdtUsdcVault`
- Category: Time and State [7]
- CWE subcategory: CWE-682 [2]

Description

As mentioned earlier, the `Shoebill` protocol is heavily forked from the popular `AaveV2` protocol. However, the use of `KlayswapUsdtUsdcVault` shows the support of `Klayswap`-based LP tokens as the collateral. Our analysis shows the potential incompatibility of current `AaveV2`-based protocols with `Klayswap`-based LP tokens. In particular, the discussion with the team indicates the `Fair Uniswap's LP Token Pricing` model will be used as the backend oracle.

Unfortunately, the known Fair Uniswap's LP Token Pricing is not compatible with the AaveV2-based lending protocols. The reason is that the fair LP price approach may be manipulated via donation to inflate the LP valuation, which is further combined with leverage to drain pool funds!¹

Recommendation Revisit the support of KlaySwap-based LP tokens as the collateral. Or make use of reliable off-chain price oracles for robust feed of LP token prices.

Status This issue has been fixed by not supporting any UniV2 LPs as collateral until trusted oracle is developed.

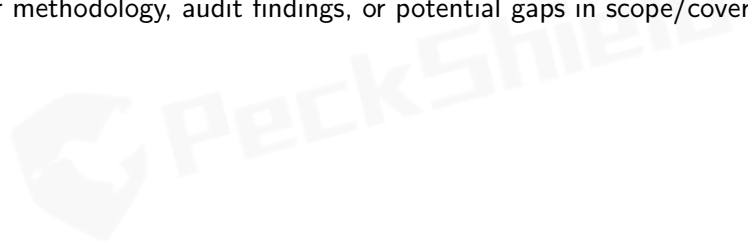


¹This hack is evident in the recent ReoMarket attack as shown in the following transaction <https://etherscan.io/tx/0x927b784148b60d5233e57287671cdf67d38e3e69e5b6d0ecacc7c1aeaa98985b>.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Shoebill` protocol, which is a decentralized non-custodial liquidity markets protocol that is developed on top of one of the largest DeFi protocols, i.e., `AAVE`. On existing lending protocols, the interest earned by lenders comes from borrowers. `Shoebill` uses a different model, where yield instead comes from the borrowers' collateral. Specifically, when borrowers provide a token as collateral, `Shoebill` converts it into an interest-bearing token (`ibToken`) using protocols like `Yearn` or `Lido`. Over time, these `ibTokens` accrue yield and the yield from these tokens is then distributed to lenders in the same token they deposited. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that 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-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [2] MITRE. CWE-682: Incorrect Calculation. <https://cwe.mitre.org/data/definitions/682.html>.
- [3] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. <https://cwe.mitre.org/data/definitions/837.html>.
- [4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.html>.
- [5] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [6] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [7] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. <https://cwe.mitre.org/data/definitions/389.html>.
- [8] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [9] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
- [10] PeckShield. PeckShield Inc. <https://www.peckshield.com>.