SMART CONTRACT AUDIT REPORT

for

TenFinance Yieldex

Prepared By: Yiqun Chen

PeckShield

September 28, 2021

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	September 28, 2021	Jing Wang	Final Release
1.0-rc	September 20, 2021	Jing Wang	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 smart contract source code of the `Yieldex` 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 Yieldex

The `TenFinance Yieldex` is a decentralized liquidity mining platform which provides an incentive mechanism to reward the staking of supported assets with certain reward tokens. Comparing to the traditional framing protocol, `TenFinance Yieldex` adds the new support of `Hold one token, earn another`. The `LP` provider could deposit a specific kind of tokens into the platform and the platform will do necessary swap and farm with the converted `LP` tokens for rewards.

The basic information of `TenFinance Yieldex` is as follows:

Table 1.1: Basic Information of Yieldex

Item	Description
Target	Yieldex
Website	https://ten.finance/
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 28, 2021

In the following, we list the reviewed file and the commit hash values used in this audit. We need to point out that this audit is focus on the `yieldex.sol` contract, the `TenFarm` and the related strategy is out of this audit's scope.

- https://github.com/tenfinance/yieldex/blob/main/yieldex.sol (a68e4e7)

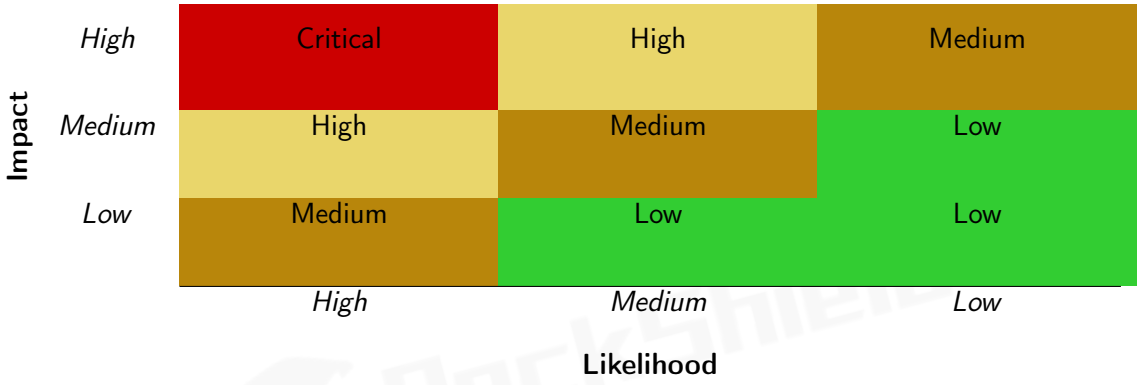
And here are the commit IDs after all fixes for the issues found in the audit have been checked in:

- https://github.com/tenfinance/yieldex/blob/main/yieldex.sol (8fb2a0f)

1.2 About PeckShield

PeckShield Inc. [12] 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



1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [11]:

- 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.

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

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) [10], 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 `Yieldex` 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	2	■■
Informational	0	
Total	6	

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 2 low-severity vulnerabilities.

Table 2.1: Key Yieldex Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Validation Of Function Arguments	Coding Practices	Fixed
PVE-002	Medium	Inconsistent Multiplier of rewardShare	Business Logics	Fixed
PVE-003	High	Improved Precision By Multiplication-Before-Division	Numeric Errors	Fixed
PVE-004	High	Improved Share Deduction For withdraw()	Business Logics	Fixed
PVE-005	Medium	Possible Sandwich/MEV Attacks For Reduced Return	Time and State	Confirmed
PVE-006	Low	Improved Logic of deposit() When addLiqudity()	Business Logics	Fixed

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

3 Detailed Results

3.1 Improved Validation Of Function Arguments

- ID: PVE-001
- Severity: low
- Likelihood: Low
- Impact: Medium
- Target: `Yieldex`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

Description

In the `Yieldex` contract, several routines have the inherent assumptions on the correctness of the input parameters, which are not enforced in their implementations. In the following, we examine these routines whose validations of the input parameters need to be improved.

The first routine is `createPool()`. This routine is provided to create a new pool with the configurations from the `owner`. To elaborate, we show below the related code snippet.

```
1124    function createPool(
1125        IERC20 [] memory _Lpaddress,
1126        uint256[] memory _weights,
1127        uint256[] memory _pids
1128        )
1129    external
1130    onlyOwner
1131    returns(bool)
1132    {
1133        require(_Lpaddress.length > 2 && _Lpaddress.length < 7,"Error: LPaddres array
                should be >2 & < 7");
1134        uint256 sumWeight = 0;
1135        for(uint8 i = 0; i < _Lpaddress.length ; i++){
1136            require(_weights[i] > 0 , "Error : Incorrect Pool Weight");
1137            sumWeight = sumWeight.add(_weights[i]);
1138        }
1139        ...
1140    }
```

Listing 3.1: `Yieldex::createPool()`

It comes to our attention that the `createPool()` has the inherent assumption on the same length of the given arrays, i.e., `_Lpaddress`, `_weights` and `_pids`. However, this is not enforced in the `createPool()` routine. If the `owner` adds a pool with different array lengths of `_Lpaddress`, `_weights` and `_pids`, the protocol may encounter unexpected errors.

The second routine is `withdrawBalance()`. This routine is provided to calculate the user balance based on the user shares. To elaborate, we show below the related code snippet.

```
1395    function withdrawBalance(address userAddress, uint256 yieldPoolId, IERC20 lpAddress,
            uint256 balance, uint256 percent )internal view returns(uint256){
1396        uint256 share = userIndexInfo[yieldPoolId][lpAddress][userAddress].shares;
1397        uint256 _withdrawBalance = share.div(indexLpTotalShares[yieldPoolId][lpAddress].
                totalShare).mul(balance).mul(percent).div(1000);
1398        return _withdrawBalance;
1399    }
```

Listing 3.2: `Yieldex::withdrawBalance()`

It comes to our attention that `withdrawBalance()` has the inherent assumption on `percent` will be smaller than 1000. However, this is not enforced in the `withdrawBalance()` routine. If this routine is called with a `percent` larger than 1000, the routine will give a `_withdrawBalance` which exceeds the user balance.

The third routine is `deposit()`. This routine is provided to take the user's assets and swap them into the demanding tokens to add liquidity. To elaborate, we show below the related code snippet.

```
1168    function deposit(uint256 poolId, address depositTokenAddress, uint256 amount,uint256
            _slippageFactor) nonReentrant public payable  {
1169            ...
1170            if( swapPaths[iAddress].token0path[swapPaths[iAddress].token0path.length.sub
                    (1)]!= depositTokenAddress) {
1171            // Swap token0 ;
1172            token0Amt = _safeSwap(
1173                routerAddress,
1174                amountin.div(2),
1175                _slippageFactor,
1176                swapPaths[iAddress].token0path,
1177                address(this),
1178                block.timestamp.add(600)
1179                );
1180            }
1181            if(swapPaths[iAddress].token1path[swapPaths[iAddress].token1path.length.sub
                    (1)] != depositTokenAddress) {
1182            // swap token1
1183            token1Amt = _safeSwap(
1184                routerAddress,
1185                amountin.div(2),
1186            ...
1187    }
```

Listing 3.3: `Yieldex::deposit()`

It comes to our attention that the `deposit()` routine has the inherent assumption on the same value between `depositTokenAddress` and `swapPath[iAddress].token0path[0]`. However, this is not enforced in the `deposit()` routine. If the this routine is called with a different `depositTokenAddress` from `swapPath[iAddress].token0path[0]`, the user will encounter unexpected errors.

The fourth routine is `lpToToken()`. This routine is provided to take the `LP` tokens and convert them into the demanding tokens. To elaborate, we show below the related code snippet.

```
1460    function lpToToken(IERC20 iAddress, address tokenAddress, uint256 slippage )
            internal  {
1461            ...
1462            uint256 _returned1 = amountA; uint256 _returned2 = amountB;
1463            if(reverseSwapPaths[iAddress].token0path[0] != tokenAddress) {
1464                _returned1 = _safeSwap(
1465                    routerAddress,
1466                    amountA,
1467                    slippage,
1468                    reverseSwapPaths[iAddress].token0path,
1469                    address(this),
1470                    block.timestamp.add(600)
1471                );
1472            }
1473            ...
1474    }
```

Listing 3.4: `Yieldex::lpToToken()`

It comes to our attention that the `lpToToken()` has the inherent assumption on the same value between `tokenAddress` and `reverseSwapPaths[iAddress].token0path[reverseSwapPaths[iAddress].token0path.length.sub(1)]`. However, this is not enforced in the `lpToToken()` routine. If this routine is called with a different `tokenAddress` from `reverseSwapPaths[iAddress].token0path[reverseSwapPaths[iAddress].token0path.length.sub(1)]`, the user will encounter unexpected errors.

Recommendation Make the validation of function parameters explicit in above functions.

Status This issue has been fixed in this commit: `fd5ffc9`.

3.2 Inconsistent Multiplier of rewardShare

- ID: PVE-002
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `Yieldex`
- Category: Business Logics [7]
- CWE subcategory: CWE-841 [5]

Description

In the `Yieldex` protocol, the `supplyFarm()` routine is used to deposit `LP` tokens into the `TenFarm` and deliver the earned rewarding to share holders. To elaborate, we show below the related code snippet of the `supplyFarm()` routine.

```
1315    function supplyFarm(uint256 yieldPoolId) internal {
1316        for(uint i = 0; i < poolInfo[yieldPoolId].length; ++i) {
1317
1318            // Fetch the LP Address of the pool
1319            IERC20 iAddress = poolInfo[yieldPoolId].Lpaddress[i];
1320
1321            // Fetch the LP balance the pool
1322            uint256 balance =  IERC20(iAddress).balanceOf(address(this));
1323
1324            // Decrease the allowance
1325            IERC20(iAddress).safeApprove(farmAddress,0);
1326
1327            // Increase the allowance
1328            IERC20(iAddress).safeIncreaseAllowance(farmAddress, balance);
1329
1330            // Check the balance of the ten Token
1331            uint256 tenbalance = IERC20(tenToken).balanceOf(address(this));
1332
1333            // deposit the balance in the farm
1334            TenFarm(farmAddress).deposit(poolInfo[yieldPoolId].pids[i],balance);
1335
1336            // Ten Earned
1337            uint256 tenEarned = IERC20(tenToken).balanceOf(address(this)).sub(tenbalance
                    );
1338            rewardShare = 0 ;
1339            // accTennPerShare is calculated here
1340            if(indexLpTotalShares[yieldPoolId][iAddress].totalShare > 0){
1341            indexLpTotalShares[yieldPoolId][iAddress].accTenPerShare =
                        indexLpTotalShares[yieldPoolId][iAddress].accTenPerShare.add(tenEarned.
                        mul(1e12).div(indexLpTotalShares[yieldPoolId][iAddress].totalShare));
1342            rewardShare = indexLpTotalShares[yieldPoolId][iAddress].accTenPerShare.mul(
                        userIndexInfo[yieldPoolId][iAddress][msg.sender].shares);
1343        }
1344
```

```
1345        indexLpTotalShares[yieldPoolId][iAddress].totalReward = indexLpTotalShares[
                yieldPoolId][iAddress].totalReward.add(tenEarned);
1346
1347        if(rewardShare > userIndexInfo[yieldPoolId][iAddress][msg.sender].rewardDebt){
1348        IERC20(tenToken).safeTransfer(msg.sender,(rewardShare.sub(userIndexInfo[
                yieldPoolId][iAddress][msg.sender].rewardDebt)).div(1e13));
1349        userIndexInfo[yieldPoolId][iAddress][msg.sender].rewardDebt =
1350        userIndexInfo[yieldPoolId][iAddress][msg.sender].rewardDebt.add(rewardShare.sub(
                userIndexInfo[yieldPoolId][iAddress][msg.sender].rewardDebt));
1351        }
1352
1353        // Increase the total share mapping(uint256 => mapping(address => uint256))
                indexLpTotalShares;
1354        indexLpTotalShares[yieldPoolId][iAddress].totalShare=indexLpTotalShares[
                yieldPoolId][iAddress].totalShare.add(balance);
1355
1356        userIndexInfo[yieldPoolId][iAddress][msg.sender].shares = userIndexInfo[
                yieldPoolId][iAddress][msg.sender].shares.add(balance);
1357
1358        }
1359    }
```

Listing 3.5: `Yieldex::supplyFarm()`

While examining the logic of above function, we notice the calculation of `rewardShare` is using `1e12` as the multiplier to derive `tenEarned` (line 1341), but the same `rewardShare` is divided by `1e13` to get the `tenEarned` (line 1348). This is inconsistent and would cause expected error for the calculation of `accTenPerShare`.

Recommendation Be consistent when applying the multiplier to properly compute the `rewardShare` .

Status This issue has been fixed in this commit: `fd5ffc9`.

3.3 Improved Precision By Multiplication-Before-Division

- ID: PVE-003
- Severity: High
- Likelihood: High
- Impact: High
- Target: `Yieldex`
- Category: Numeric Errors [9]
- CWE subcategory: CWE-190 [2]

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 `withdrawBalance()` routine as an example. This routine is provided to calculate the user balance based on the user shares. To elaborate, we show below the related code snippet.

```
1359    function withdrawBalance(address userAddress, uint256 yieldPoolId, IERC20 lpAddress,
            uint256 balance, uint256 percent )internal view returns(uint256){
1360        uint256 share = userIndexInfo[yieldPoolId][lpAddress][userAddress].shares;
1361        uint256 _withdrawBalance = share.div(indexLpTotalShares[yieldPoolId][lpAddress].
                totalShare).mul(balance).mul(percent).div(1000);
1362        return _withdrawBalance;
1363    }
```

Listing 3.6: `Yieldex::withdrawBalance()`

We notice the calculation of the `_withdrawBalance` involves both multiplication and division: `share.div(indexLpTotalShares[yieldPoolId][lpAddress].totalShare).mul(balance).mul(percent).div(1000)` (line 1397). For improved precision, it is better to calculate the multiplication before the division, i.e., `share.mul(balance).mul(percent).div(1000).div(indexLpTotalShares[yieldPoolId][lpAddress].totalShare)`. Note that the resulting precision loss will cause `_withdrawBalance` to be 0 as long as the user shares are smaller than total shares, which eventually result in 0 withdraw from the `TenFarm`.

```
1403    function withdraw(uint256 yieldPoolId, uint percent, address tokenAddress, uint256
            slippage)public nonReentrant returns(uint){
1404        for(uint i = 0; i < poolInfo[yieldPoolId].length; i++) {
1405            rewardShare = 0;
1406            subValue = 0;
1407            IERC20 iAddress = poolInfo[yieldPoolId].Lpaddress[i];
1408            uint256 balance = TenFarm(farmAddress).stakedWantTokens(poolInfo[yieldPoolId
                    ].pids[i], address(this));
1409            uint256 _withdrawBalance = withdrawBalance(msg.sender, yieldPoolId, iAddress
                    , balance, percent);
1410            uint256 tenbalance = IERC20(tenToken).balanceOf(address(this));
1411            TenFarm(farmAddress).withdraw(poolInfo[yieldPoolId].pids[i],_withdrawBalance
                    );
1412            ...
1413        }
```

Listing 3.7: `Yieldex::withdraw()`

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

Status This issue has been fixed in this commit: `fd5ffc9`.

3.4 Improved Share Calculation For withdraw()

- ID: PVE-004
- Severity: High
- Likelihood: High
- Impact: High
- Target: Yieldex
- Category: Business Logics [7]
- CWE subcategory: CWE-841 [5]

Description

In the Yieldex protocol, the withdraw() routine is used to withdraw LP tokens from the TenFarm and deliver the earned rewards to the share holders. To elaborate, we show below the related code snippet of the withdraw() routine.

```
1403    function withdraw(uint256 yieldPoolId, uint percent, address tokenAddress, uint256
            slippage)public nonReentrant returns(uint){
1404        for(uint i = 0; i < poolInfo[yieldPoolId].length; i++) {
1405            rewardShare = 0;
1406            subValue = 0;
1407            IERC20 iAddress = poolInfo[yieldPoolId].Lpaddress[i];
1408            uint256 balance = TenFarm(farmAddress).stakedWantTokens(poolInfo[yieldPoolId
                    ].pids[i], address(this));
1409            uint256 _withdrawBalance = withdrawBalance(msg.sender, yieldPoolId, iAddress
                    , balance, percent);
1410            uint256 tenbalance = IERC20(tenToken).balanceOf(address(this));
1411            TenFarm(farmAddress).withdraw(poolInfo[yieldPoolId].pids[i],_withdrawBalance
                    );
1412            // accTennPerShare is calculated here
1413            uint256 tenEarned = (IERC20(tenToken).balanceOf(address(this))).sub(
                    tenbalance);
1414
1415
1416            // accTennPerShare is calculated here
1417            if(indexLpTotalShares[yieldPoolId][iAddress].totalShare > 0) {
1418                indexLpTotalShares[yieldPoolId][iAddress].accTenPerShare =
                        indexLpTotalShares[yieldPoolId][iAddress].accTenPerShare.add(
                        tenEarned.mul(1e12).div(indexLpTotalShares[yieldPoolId][iAddress].
                        totalShare));
1419                rewardShare = indexLpTotalShares[yieldPoolId][iAddress].accTenPerShare.
                        mul(userIndexInfo[yieldPoolId][iAddress][msg.sender].shares);
1420            }
1421
1422            indexLpTotalShares[yieldPoolId][iAddress].totalReward = indexLpTotalShares[
                    yieldPoolId][iAddress].totalReward.add(tenEarned);
1423            // uint256 rewardShare = indexLpTotalShares[yieldPoolId][iAddress].
                    accTenPerShare.mul(userIndexInfo[yieldPoolId][iAddress][msg.sender].
                    shares.mul(1e12).div(indexLpTotalShares[yieldPoolId][iAddress].
                    totalShare));
1424            {
```

```
1425                subValue =userIndexInfo[yieldPoolId][iAddress][msg.sender].rewardDebt;
1426            }
1427
1428            if(rewardShare > userIndexInfo[yieldPoolId][iAddress][msg.sender].rewardDebt
                    ) {
1429                IERC20(tenToken).safeTransfer(msg.sender,(rewardShare.sub(subValue).div
                        (1e13)));
1430                if (percent != 0) {
1431                    if(tokenAddress == address(0)) {
1432                        IERC20(iAddress).safeTransfer(msg.sender,IERC20(iAddress).
                            balanceOf(address(this)));
1433                    }
1434                    else {
1435                        lpToToken(iAddress, tokenAddress, slippage);
1436                    }
1437                }
1438            }
1439
1440            if(percent == 1000){
1441                userIndexInfo[yieldPoolId][iAddress][msg.sender].rewardDebt = 0;
1442            }
1443
1444            else{
1445                userIndexInfo[yieldPoolId][iAddress][msg.sender].rewardDebt =
1446                userIndexInfo[yieldPoolId][iAddress][msg.sender].rewardDebt.add(
                    rewardShare.sub(userIndexInfo[yieldPoolId][iAddress][msg.sender].
                    rewardDebt));
1447                indexLpTotalShares[yieldPoolId][iAddress].totalShare =
                    indexLpTotalShares[yieldPoolId][iAddress].totalShare.sub(
                    _withdrawBalance);
1448                userIndexInfo[yieldPoolId][iAddress][msg.sender].shares = userIndexInfo[
                    yieldPoolId][iAddress][msg.sender].shares.sub(_withdrawBalance);
1449            }
1450        }
1451    }
```

Listing 3.8: `Yieldex::withdraw()`

While examining the logic of above function, we notice several logic errors in it.The first one is the requirement of `rewardShare > userIndexInfo[yieldPoolId][iAddress][msg.sender].rewardDebt` (line 1428), which is used to calculate if there are pending rewards for the specific user, should not been used to constrain the `LP` tokens' withdraw. The second one is `userShare` deduction should be properly handled in the case of `percent == 1000` (line 1440). If the deduction is not handled when `percent == 1000`, a bad actor could do the 1000 percent withdraw multiple times until all funds of the protocol are drained. The last one is when computing the user shares deduction, `_withdrawBalance` should be converted to share, i.e., `_withdrawBalance/balance*totalShares`.

Recommendation Fix the logic errors in the above `withdraw()` routine by following the specific suggestions.

Status This issue has been fixed in this commit: 53d2f85. Due to the TenFarm contract is out of this audit's scope, we consult the team about the funds deposited into the TenFarm contract. The team clarifies there would not be any loss for the LPs investment, the TenFarm will return the original amount of LPs tokens and the rewards.

3.5 Possible Sandwich/MEV Attacks For Reduced Conversion

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Yieldex
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

Description

The Yieldex contract has a helper routine, i.e., _safeSwap(), that is designed to convert user assets into the demanding tokens. To elaborate, we show below the related code snippet.

```
478     function _safeSwap(
479         address _uniRouterAddress,
480         uint256 _amountIn,
481         uint256 _slippageFactor,
482         address[] memory _path,
483         address _to,
484         uint256 _deadline
485     ) internal virtual returns(uint256) {
486         _approveTokenIfNeeded(_path[0],_uniRouterAddress);
487         uint256[] memory amounts =
488             IPancakeRouter02(_uniRouterAddress).getAmountsOut(_amountIn, _path);
489         uint256 amountOut = amounts[amounts.length.sub(1)].mul(_slippageFactor).div
                (1000);
490         uint256 _returned = IPancakeRouter02(_uniRouterAddress)
491             .swapExactTokensForTokens(
492             _amountIn,
493             amountOut,
494             _path,
495             _to,
496             _deadline
497         )[1];
498         return _returned;
499     }
```

Listing 3.9: Yieldex::_safeSwap()

We notice the token swap is routed to a router IPancakeRouter02. And the actual swap operation swapExactTokensForTokens() essentially does not specify a valid restriction on possible slippage and

is therefore vulnerable to possible front-running attacks, resulting in a smaller converted amount. Other routines `deposit()` and `lpToToken()` share the same issue.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user or the virtual account in our case because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above sandwich attack to better protect the interests of protocol users.

Status The issue has been confirmed by the team.

3.6 Improved Logic of deposit() When addLiqudity()

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Yieldex`
- Category: Business Logics [7]
- CWE subcategory: CWE-754 [4]

Description

According to the `Yieldex` design, the `deposit()` routine will convert user assets into the demanding tokens and add liquidity for the target `LP` tokens (via `addLiqudity()`). While examining the process of adding liquidity, we notice the leftover assets may be locked in the `Yieldex` contract. What's more, the leftover `WBNB` could be exploited to increase the deposited amount by the next user.

To elaborate, we show below the related code snippet of the `Yieldex()` contract.

```
1168    function deposit(uint256 poolId, address depositTokenAddress, uint256 amount,uint256
            _slippageFactor) nonReentrant public payable  {
1169        if(depositTokenAddress == address(0)) {
1170            _wrapBNB();
1171            depositTokenAddress = wbnbAddress;
1172            ...
1173            uint256 token0Amt = amountin.div(2); uint256 token1Amt = amountin.div(2);
1174
1175            if( swapPaths[iAddress].token0path[swapPaths[iAddress].token0path.length.sub
                    (1)]!= depositTokenAddress) {
1176            // Swap token0 ;
```

```
1177            token0Amt = _safeSwap(
1178                routerAddress,
1179                amountin.div(2),
1180                _slippageFactor,
1181                swapPaths[iAddress].token0path,
1182                address(this),
1183                block.timestamp.add(600)
1184                );
1185            }
1186            if(swapPaths[iAddress].token1path[swapPaths[iAddress].token1path.length.sub
                    (1)] != depositTokenAddress) {
1187            // swap token1
1188            token1Amt = _safeSwap(
1189                routerAddress,
1190                amountin.div(2),
1191                _slippageFactor,
1192                swapPaths[iAddress].token1path,
1193                address(this),
1194                block.timestamp.add(600)
1195                );
1196            }
1197            ...
1198            IPancakeRouter02(routerAddress).addLiquidity(
1199                    swapPaths[iAddress].token0path[swapPaths[iAddress].token0path.length
                            .sub(1)],
1200                    swapPaths[iAddress].token1path[swapPaths[iAddress].token1path.length
                            .sub(1)],
1201                    token0Amt,
1202                    token1Amt,
1203                    0,
1204                    0,
1205                    address(this),
1206                    block.timestamp.add(600)
1207                );
1208            }
1209
1210        }
```

Listing 3.10: `Yieldex::deposit()`

This routine will swap half of the deposited tokens into `token0`/`token1` and add the output to provide liquidity. However, the logic ignores the fact that the swap of the deposited token to the `token0`/`token1` will drive up the price of the deposited token in the pair, which will lead to the result where certain deposited tokens are left after the calling of `addLiquidity()`. What's more, if the leftover tokens are `WBNB`, the exploiter could increase the amount of deposited token on the next report by using the leftover `WBNB`.

Recommendation Refund the extra `tokenAmt - amountA` tokens to the `msg.sender` or do an optimization for the calculation of an optimal swap token amount.

Status This issue has been fixed in this commit: `53d2f85`. The team adds a function to withdraw the leftover tokens.

4 Conclusion

In this audit, we have analyzed the `Yieldex` design and implementation. The `Yieldex` provides a incentive mechanism to reward the staking of supported assets with certain reward tokens. 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

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

[4] MITRE. CWE-754: Improper Check for Unusual or Exceptional Conditions. https://cwe.mitre.org/data/definitions/754.html.

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

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

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

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

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

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

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

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