![PeckShield]

SMART CONTRACT AUDIT REPORT

for

Phoenix LevergedPool

Prepared By: Yiqun Chen

Hangzhou, China
August 5, 2020

Document Properties

Client	Phoenix Protocol
Title	Smart Contract Audit Report
Target	Phoenix LevergedPool
Version	1.0
Author	Xuxian Jiang
Auditors	Xiaotao Wu, Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 5, 2020	Xuxian Jiang	Final Release
1.0-rc1	July 5, 2020	Xuxian Jiang	Release Candidate #1
0.3	July 2, 2020	Xuxian Jiang	Additional Findings #2
0.2	June 28, 2020	Xuxian Jiang	Additional Findings #1
0.1	June 19, 2020	Xuxian Jiang	Initial Draft

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 `Phoenix LevergedPool` 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 Phoenix LevergedPool

The `Phoenix LevergedPool` protocol is a permissionless protocol to leverage or hedge their positions in a variety of crypto-assets. It is innovative in having leveraged or hedged exposure for multiple assets from staking pools. Specifically, the protocol enables anyone anywhere to leverage or hedge their positions in a variety of crypto-assets. With the possibility of running on multiple chains including `Ethereum` and `Wanchain`, the `Phoenix LevergedPool` protocol provides a valuable instrument to hedge risks and control excessive exposure from market fluctuation and dynamics, therefore presenting a unique contribution to current DeFi ecosystem.

The basic information of `Phoenix LevergedPool` is as follows:

Table 1.1: Basic Information of Phoenix LevergedPool

Item	Description
Issuer	Phoenix Protocol
Website	https://www.finnexus.io
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 5, 2020

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

- https://github.com/Phoenix-Finance/LeveragedPool.git (f915862)

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

- https://github.com/Phoenix-Finance/LeveragedPool.git (3fe3c28)

1.2   About PeckShield

PeckShield Inc. [14] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact	Likelihood High	Likelihood Medium	Likelihood 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 [13]:

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

- Impact measures the technical loss and business damage of a successful attack;

- Severity demonstrates the overall criticality of the risk.

PeckShield Audit Report #: 2021-185

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) [12], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4   Disclaimer

Note that this audit does not give any warranties on finding all possible security issues of the given smart contract(s), 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 1.3: 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

PeckShield Audit Report #: 2021-185

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.

PeckShield Audit Report #: 2021-185

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `Phoenix LevergedPool` 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	6	▪▪▪▪▪
Low	3	▪▪▪
Informational	1	▪
Total	10	

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 6 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Phoenix LevergedPool Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Potential Reentrancy Risk in rebaseToken::redeemToken()	Time and State	Fixed
PVE-002	Medium	Proper Accounting in PHXAccelerator::stake()	Business Logic	Fixed
PVE-003	Low	Possible Sandwich/MEV Attacks For Reduced Returns	Time and State	Confirmed
PVE-004	Medium	Possible Front-Running DoS Against Collateral Redemption	Business Logic	Confirmed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-006	Low	Proper Allowance Reset in setSwapRouterAddress()	Coding Practices	Fixed
PVE-007	Medium	Proper latestSettleTime Accounting	Business Logic	Fixed
PVE-008	Medium	Proper Accounting in accelerated-MinePool	Business Logic	Fixed
PVE-009	Medium	Proper Accounting in PHXAccelerator::unstake()	Business Logic	Fixed
PVE-010	Informational	Unused State/Code Removal	Coding Practices	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 Potential Reentrancy Risk in rebaseToken::redeemToken()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `rebaseToken`
- Category: Time and State [10]
- CWE subcategory: CWE-663 [4]

Description

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

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `rebaseToken` as an example, the `redeemToken()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 67) starts before effecting the update on the internal state (line 69), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the very same `redeemToken()` function.

```
64      function redeemToken() public {
65          uint256 amount = getRedeemAmount(msg.sender);
66          if(amount > 0){
67              _redeem(msg.sender,leftToken,amount);
68          }
```

```
69          userBeginRound[msg.sender] = Erc20InfoList.length -1;
70       }
```

Listing 3.1: `rebaseToken::redeemToken()`

In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`.

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

Status The issue has been fixed by this commit: `8e3a49a`.

3.2 Proper Accounting in PHXAccelerator::stake()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium

- Target: `PHXAccelerator, PHXVestingPool`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The `Phoenix LevergedPool` protocol has a built-in incentive mechanism that allows protocol users to staking the governance tokens `PHX` for additional mining rewards. While examining the current staking logic of `PHX`, we notice the current implementation can be improved.

To elaborate, we show below the related `stake()` function. This function has a rather straightforward logic in collecting the staked amount, and computing the accelerator balance from the staked assets, updating related accounting (including the user lockup period), and synchronizing with the mining pool. However, our analysis shows it fails to update the user's `tokenBalance` state. Moreover, it notifies the mining pool with an incorrect balance amount (line 40). In addition, the current implementation can be improved by applying an input validation on the given `maxLockedPeriod`.

```
32      function stake(address token,uint256 amount,uint128 maxLockedPeriod,address
            toMinePool) nonReentrant notHalted public {
33          amount = getPayableAmount(token,amount);
34          require(amount>0, "Stake amount is zero!");
35          uint256 rate = tokenAcceleratorRate[token];
36          require(rate>0 , "Stake token accelerate rate is zero!");
37          uint256 balance = amount.mul(rate);
38          userInfoMap[msg.sender].AcceleratorBalance  =  userInfoMap[msg.sender].
                AcceleratorBalance.add(balance);
39          serUserLockedPeriod(msg.sender,maxLockedPeriod);
```

```
40          _accelerateMinePool(toMinePool,balance);
41          emit Stake(msg.sender,token,amount,maxLockedPeriod);
42      }
```

<div align="center">Listing 3.2: <code>PHXAccelerator::stake()</code></div>

Recommendation Revise the above `stake()` logic to ensure proper accounting on `tokenBalance` and the protocol-wide mining pool. It can also be improved by validating the given `maxLockedPeriod`.

Status The issue has been fixed by this commit: `6c0d840`.

3.3   Possible Sandwich/MEV Attacks For Reduced Returns

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium

- Target: `Multiple Contracts`
- Category: Time and State [11]
- CWE subcategory: CWE-682 [5]

Description

The `Phoenix LevergedPool` protocol allows users to hedge risks and control excessive exposure from market fluctuation and dynamics. And there is a constant need to perform swaps from one token to another. To fulfill this need, the protocol has developed a contract `PHXSwapRouter`, which seamlessly integrates with `UniswapV2` and `OneSplit`.

To elaborate, we show below the `_swap()` function in `UniSwapRouter`. This routine essentially performs the swap between the given two tokens, i.e., `token0` and `token1`.

```
37      function _swap(address swapRouter,address token0,address token1,uint256 amount0)
            internal returns (uint256) {
38          IUniswapV2Router02 IUniswap = IUniswapV2Router02(swapRouter);
39          address[] memory path = new address[](2);
40          uint256[] memory amounts;
41          if(token0 == address(0)){
42              path[0] = IUniswap.WETH();
43              path[1] = token1;
44              amounts = IUniswap.swapExactETHForTokens.value(amount0)(0, path,address(this
                    ), now+30);
45          }else if(token1 == address(0)){
46              path[0] = token0;
47              path[1] = IUniswap.WETH();
48              amounts = IUniswap.swapExactTokensForETH(amount0,0, path, address(this), now
                    +30);
49          }else{
50              path[0] = token0;
51              path[1] = token1;
```

```
52            amounts = IUniswap.swapExactTokensForTokens(amount0,0, path, address(this),
                 now+30);
53         }
54         emit Swap(token0,token1,amounts[0],amounts[amounts.length-1]);
55         return amounts[amounts.length-1];
56     }
```

Listing 3.3: `UniSwapRouter::_swap()`

We notice the conversion is routed to the external `UniswapV2` without any slippage control. With that, it is possible for a malicious actor to launch a flashloan-assisted attack to claim the majority of swaps, resulting in a significantly less amount after the swap. This is possible if the `_swap()` function suffers from a sandwich attack.

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

Status   The issue has been confirmed.

3.4   Possible Front-Running DoS Against Collateral Redemption

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High

- Target: `PPTCoin`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

As mentioned in Section 3.2, the `Phoenix LevergedPool` protocol has a built-in incentive mechanism that allows protocol users to staking the governance tokens `PHX` for additional mining rewards. The staking operation enforces a lockup period that disallows users from unstaking until the lockup period expires.

In the following, we show a function named `redeemLockedCollateral()`, which is designed to redeem locked `PPT` tokens. We note this function has an associated modifier `OutLimitation(account)`, which essentially enforces the following requirement: `require(addressTimeMap[account].time+limitation<now`.

```
145     function redeemLockedCollateral(address account,uint256 tokenAmount,uint256
            leftCollateral)public onlyManager OutLimitation(account) returns (uint256,
            uint256){
146         if (leftCollateral == 0){
147             return(0,0);
148         }
149         uint256 lockedAmount = lockedBalances[account];
150         uint256 lockedWorth = lockedTotalWorth[account];
```

```
151          if (lockedAmount == 0  lockedWorth == 0){
152              return (0,0);
153          }
154          uint256 redeemWorth = 0;
155          uint256 lockedBurn = 0;
156          uint256 lockedPrice = lockedWorth/lockedAmount;
157          if (lockedAmount >= tokenAmount){
158              lockedBurn = tokenAmount;
159              redeemWorth = tokenAmount*lockedPrice;
160          }else{
161              lockedBurn = lockedAmount;
162              redeemWorth = lockedWorth;
163          }
164          if (redeemWorth > leftCollateral) {
165              lockedBurn = leftCollateral/lockedPrice;
166              redeemWorth = lockedBurn*lockedPrice;
167          }
168          if (lockedBurn > 0){
169              _subLockBalance(account,lockedBurn,redeemWorth);
170              return (lockedBurn,redeemWorth);
171          }
172          return (0,0);
173      }
```

Listing 3.4: `PPTCoin::redeemLockedCollateral()`

By examining the above modifier, we identify a possible front-running attack that may block an ongoing redeem attempt. Specifically, when a `transfer`() or `transferFrom`() action occurs, the lockup period of the receiver, i.e., `addressTimeMap[to]`, might be accordingly updated. Therefore, upon the observation of a `redeemLockedCollateral()` attempt from a victim, a malicious actor could intentionally transfer 1 `WEI` to the victim. By doing so, the `addressTimeMap` of the victim is updated with current timestamp. As a result, the specific redeem attempt is blocked as it occurs in the lockup period (line 145).

Recommendation A mitigation to the above front-running attacks need to prevent malicious actors from tampering with legitimate accounts. In the meantime, we acknowledge that front-running attacks are inherent in current DeFi system and there is still a need to search for more effective countermeasures.

Status The issue has been confirmed.

3.5   Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High

- Target: `Multiple Contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [2]

Description

In the `Phoenix LevergedPool` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., adding new roles and configuring various system parameters). In the following, we show the representative functions potentially affected by the privilege of the untrusted `owner` account.

```
16      function setPoolInfo(address PPTToken,address stakeToken,uint64 interestrate) public
            onlyOwner{
17          _PPTCoin = IPPTCoin(PPTToken);
18          _poolToken = stakeToken;
19          _interestRate = interestrate;
20          _defaultRate = interestrate;
21      }
22
23      ...
24
25      function setInterestRate(uint64 interestrate) public onlyOwner{
26          _interestRate = interestrate;
27          _defaultRate = interestrate;
28      }
29      function interestInflation(uint64 inflation)public onlyOwner{
30          if(_totalSupply > 0){
31              uint256 balance = poolBalance();
32              if(balance*100<_totalSupply){
33                  _interestRate = _interestRate*inflation/1e8;
34              }else{
35                  _interestRate = _defaultRate;
36              }
37          }
38      }
```

Listing 3.5: A number of representative `setters` in `stakePool`

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

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

Status This issue has been confirmed with the team. For the time being, the team has confirmed that these privileged functions should be called by a trusted multi-sig account, not a plain EOA account.

3.6   Proper Allowance Reset in setSwapRouterAddress()

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: leveragedPool
- Category: Coding Practices [8]
- CWE subcategory: CWE-1041 [1]

Description

At the core of Phoenix LevergedPool is the leveragedPool contract that supports the main functionality of leverage and hedging. It also allows the privileged owner to update the active swap router swapRouter for various token conversion. In the following, we examine this specific setSwapRouterAddress() function.

It comes to our attention that this function properly sets up the spending allowance to the new swapRouter. However, it forgets to cancel the previous spending allowance from the old swapRouter.

```
20     function setSwapRouterAddress(address _swapRouter)public onlyOwner{
21         require(swapRouter != _swapRouter,"swapRouter : same address");
22         swapRouter = _swapRouter;
23         if(leverageCoin.token != address(0)){
24             IERC20 oToken = IERC20(leverageCoin.token);
25             oToken.safeApprove(swapRouter,uint256(-1));
26         }
27         if(hedgeCoin.token != address(0)){
28             IERC20 oToken = IERC20(hedgeCoin.token);
29             oToken.safeApprove(swapRouter,uint256(-1));
30         }
31     }
```

Listing 3.6: leveragedPool::setSwapRouterAddress()

Recommendation Remove the spending allowance from the old swapRouter when it is updated.

Status This issue has been fixed in the following commit: 9e0d641.

3.7   Proper latestSettleTime Accounting

- ID: PVE-008
- Severity: Medium
- Likelihood: Low
- Impact: High

- Target: `acceleratedMinePool`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The mining support of `Phoenix LevergedPool` is mainly implemented in the `acceleratedMinePool` contract. While examining the mining support, we notice one key routine that settles on a specific mine coin needs to be improved.

Specifically, this key routine `_mineSettlement()` (as shown below) maintains the last settled timestamp (`latestSettleTime`) for the mine coin. However, the `latestSettleTime` state is not properly updated or maintained. In particular, the current update logic (lines $187 - 191$) depends on the `mineInterval` state, which should not be the case. In fact, regardless of the `mineInterval` state, the `latestSettleTime` needs to be updated always with `currentTime()` (line 190).

```
160     function _mineSettlement(address mineCoin)internal{
161         uint256 latestTime = mineInfoMap[mineCoin].latestSettleTime;
162         uint256 curIndex = getPeriodIndex(latestTime);
163         if (curIndex == 0){
164             latestTime = startTime;
165         }
166         uint256 nowIndex = getPeriodIndex(currentTime());
167         if (nowIndex == 0){
168             return;
169         }
170         for (uint256 i=0;i<_maxLoop;i++){
171             // If the fixed distribution is zero, we only need calculate
172             uint256 finishTime = getPeriodFinishTime(curIndex);
173             if (finishTime < currentTime()){
174                 _mineSettlementPeriod(mineCoin,curIndex,finishTime.sub(latestTime));
175                 latestTime = finishTime;
176             }else{
177                 _mineSettlementPeriod(mineCoin,curIndex,currentTime().sub(latestTime));
178                 latestTime = currentTime();
179                 break;
180             }
181             curIndex++;
182             if (curIndex > nowIndex){
183                 break;
184             }
185         }
186         uint256 _mineInterval = mineInfoMap[mineCoin].mineInterval;
187         if (_mineInterval>0){
```

```
188            mineInfoMap[mineCoin].latestSettleTime = latestTime/_mineInterval*
                   _mineInterval;
189        }else{
190            mineInfoMap[mineCoin].latestSettleTime = currentTime();
191        }
192    }
```

<div align="center">Listing 3.7: <code>acceleratedMinePool::_mineSettlement()</code></div>

Recommendation   Revise the above `_mineSettlement()` routine to properly update the `latestSettleTime`.

Status   The issue has been fixed by this commit: `3fe3c28`.

3.8   Proper Accounting in acceleratedMinePool

- ID: PVE-008
- Severity: Medium
- Likelihood: Low
- Impact: High

- Target: `acceleratedMinePool`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

As discussed in Section 3.7, the mining support of `Phoenix LevergedPool` is mainly implemented in the `acceleratedMinePool` contract. In the following, we examine two basic helper routines `addDistribution()` and `removeDistribution()`.

As the names indicate, these two routines are designed to add or remove an user's distribution amount. While their logic is rather straightforward, our analysis shows that the `removeDistribution()` routine can be improved by resetting `userInfoMap[account].maxPeriodID = 0` so that it can allow the user to stake again. [1].

```
418    function removeDistribution(address account,uint256 oldAcceleratedStake,uint256
           oldAcceleratedPeriod) internal {
419        uint256 addrLen = whiteList.length;
420        for(uint256 i=0;i<addrLen;i++){
421            _mineSettlement(whiteList[i]);
422            _settleUserMine(whiteList[i],account);
423        }
424        uint256 distri = calculateDistribution(account,oldAcceleratedStake,
               oldAcceleratedPeriod);
```

---

[1]An intermediate version of the `addDistribution()` routine is flawed when updating the `weightDistributionMap` state (line 442). In particular, the original implementation of `distri.mul(getPeriodWeight(nowId,endId)-1000)/1000` is proper and needs to restored back to replace the updated `distri.mul(getPeriodWeight(nowId,endId)-rateDecimal)/rateDecimal`

```
425        totalDistribution = totalDistribution.sub(distri);
426        uint256 nowId = getPeriodIndex(currentTime());
427        uint256 endId = userInfoMap[account].maxPeriodID;
428        for(;nowId<=endId;nowId++){
429            weightDistributionMap[nowId] = weightDistributionMap[nowId].sub(distri.mul(
                   getPeriodWeight(nowId,endId)-1000)/1000);
430        }
431        userInfoMap[account].distribution =  0;
432    }
433    /**
434     * @dev Auxiliary function. Add user's distribution amount.
435     * @param account user's account.
436     */
437    function addDistribution(address account,uint256 acceleratedStake,uint256
           acceleratedPeriod) internal {
438        uint256 distri = calculateDistribution(account,acceleratedStake,
               acceleratedPeriod);
439        uint256 nowId = getPeriodIndex(currentTime());
440        uint256 endId = userInfoMap[account].maxPeriodID;
441        for(;nowId<=endId;nowId++){
442            weightDistributionMap[nowId] = weightDistributionMap[nowId].add(distri.mul(
                   getPeriodWeight(nowId,endId)-1000)/1000);
443        }
444        userInfoMap[account].distribution =  distri;
445        userInfoMap[account].maxPeriodID = acceleratedPeriod;
446        totalDistribution = totalDistribution.add(distri);
447    }
```

Listing 3.8: `acceleratedMinePool::removeDistribution()/addDistribution()`

Recommendation   Correct the above two routines to achieve the intended purpose.

Status   The issue has been removed due to the re-design in the following commit: `3fe3c28`.

3.9   Proper Accounting in PHXAccelerator::unstake()

- ID: PVE-009
- Severity: Medium
- Likelihood: Low
- Impact: High

- Target: `PHXAccelerator`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The `PHXAccelerator` contract has two key routines `stake()` and `unstake()` to allow for protocol token holders to participate. While examining the `unstake()` routine, we identify an issue that does not properly notify mining pools for update.

To elaborate, we show below the `unstake()` implementation. It has properly adjusted internal accounting in decrementing the withdrawn `PHX` tokens from `AcceleratorBalance` (line 68) and `tokenBalance[token]` (line 69). However, it does not notify or update the associated mining pool `acceleratedMinePool`. The notification is necessary as it needs to timely adjust the user contribution.

```
57      /**
58       * @dev withdraw PHX coin.
59       * @param amount PHX amount that withdraw from mine pool.
60       */
61      function unstake(address token,uint256 amount)public nonReentrant notHalted
            periodExpired(msg.sender){
62          require(amount > 0, 'unstake amount is zero');
63          require(userInfoMap[msg.sender].tokenBalance[token] >= amount,
64              'unstake amount is greater than total user stakes');
65          uint256 rate = tokenAcceleratorRate[token];
66          require(rate>0 , "Stake token accelerate rate is zero!");
67          uint256 balance = amount.mul(rate);
68          userInfoMap[msg.sender].AcceleratorBalance = userInfoMap[msg.sender].
            AcceleratorBalance.sub(balance);
69          userInfoMap[msg.sender].tokenBalance[token] = userInfoMap[msg.sender].
            tokenBalance[token]-amount;
70          emit Unstake(msg.sender,token,amount);
71      }
```

Listing 3.9: `PHXAccelerator::unstake()`

Recommendation Timely notify the mining pool when a protocol user withdraws his/her `PHX` coins (via `unstake()`).

Status The issue has been fixed by this commit: `8e3a49a`.

3.10 Unused State/Code Removal

- ID: PVE-010
- Severity: Informational
- Likelihood: N/A
- Impact: N/A

- Target: `PPTCoin`
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [3]

Description

The `Phoenix LevergedPool` protocol makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `SafeMath`, and `ReentrancyGuard`, to facilitate its code implementation and organization. For example, the `leveragedPool` contract has so far imported at least five reference contracts. However,

we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the state variables defined in the `PPTCoin` contract, one of them is not used: `timeLimitWhiteList`. The unused state can be safely removed.

```
15  contract PPTCoin is SharedCoin {
16      using SafeMath for uint256;
17      mapping (address => bool) internal timeLimitWhiteList;
18      ...
19  }
```

Listing 3.10: The `PPTCoin` Contract

Recommendation   Consider the removal of the redundant code with a simplified, consistent implementation.

Status   The issue has been fixed by this commit: `8e3a49a`.

4 | Conclusion

In this audit, we have analyzed the Phoenix LevergedPool design and implementation. The system presents a unique offering in current DeFi ecosystem by proposing a decentralized leverage platform and enabling anyone anywhere to leverage or hedge their positions in a variety of crypto-assets. The current code base is well organized and those identified issues are promptly confirmed and fixed.

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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/ definitions/563.html.

[4] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe. mitre.org/data/definitions/663.html.

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

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

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

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

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

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

PeckShield Audit Report #: 2021-185

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

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

[13] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_ Rating_Methodology.

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

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

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