SMART CONTRACT AUDIT REPORT

for

MARLIN LABS

Prepared By: Shuxiao Wang

Hangzhou, China

December 27, 2020

Document Properties

Client	Marlin Labs
Title	Smart Contract Audit Report
Target	Marlin Stake
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Jeff Liu, Xuxian Jiang
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0	December 27, 2020	Xuxian Jiang	Final Release
1.0-rc1	December 18, 2020	Xuxian Jiang	Release Candidate #1
0.2	December 16, 2020	Xuxian Jiang	Additional Findings
0.1	December 14, 2020	Xuxian Jiang	Initial Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Stake` module in the `Marlin` 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 Marlin

`Marlin` is a secure platform-agnostic networking protocol that enables faster transmission of blocks and transactions between miners and incentivizes full nodes, making web 3 experiences smoother and cheaper. It is interoperable with a wide range of consensus algorithms. `Marlin` is backed by `Binance Labs`, `Electric Capital`, `ArringtonXRP`, `Fenbushi` and others in its mission to scale blockchains at layer-0. This audit covers the new `stake` module in the `Marlin` protocol.

The basic information of the Marlin Stake module is as follows:

Table 1.1: Basic Information of The Marlin Stake Module

Item	Description
Issuer	Marlin Labs
Website	https://www.marlin.pro/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 27, 2020

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note this repository contains a number of sub-directories (e.g., `Actors`, `Bridge`, and `Fund`) and this audit covers only the `Stake` sub-directory.

- https://github.com/marlinprotocol/Contracts/tree/staking-mvp/contracts/Stake (a84cd54)

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

- https://github.com/marlinprotocol/Contracts/tree/staking-mvp/contracts/Stake (34d3918)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- 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) [7], 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 1.4: 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 Marlin Stake implementation. During the first phase of our audit, we studied the smart contract source code and ran 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	0	
Low	4	■■■■
Informational	2	■■
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 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 4 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key Marlin Stake Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Complete Coverage of NetworkAdded Events	Coding Practices	Resolved
PVE-002	Informational	Explicit super Invocation	Coding Practices	Resolved
PVE-003	Low	Suggested Adherence of Checks-Effects-Interactions	Time and State	Resolved
PVE-004	Low	Sanity Checks For System/Function Parameters	Coding Practices	Resolved
PVE-005	Low	Simplified Business Logic in createStash()	Business Logics	Resolved
PVE-006	Low	Suggested safeTransfer()/safeTransferFrom() Replacement	Coding Practices	Resolved

Please refer to Section 3 for details.

3 | Detailed Results

3.1 Complete Coverage of NetworkAdded Events

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: ClusterRewards
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

In Ethereum, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we show the code snippets in two routines in ClusterRewards contract, i.e., initialize() and addNetwork(). The initialize() routine, as the name indicates, initializes the contract state, especially the beginning set of networks for rewards while the second routine supports the addition of new network at runtime.

```
    function initialize(
        address _owner,
        address _rewardDelegatorsAddress,
        bytes32[] memory _networkIds,
        uint256[] memory _rewardWeight,
        uint256 _totalRewardsPerEpoch,
        address _PONDAddress,
        uint256 _payoutDenomination)
        public
        initializer
    {
        require(
            _networkIds.length == _rewardWeight.length,
```

```
            "ClusterRewards:initialize - Each NetworkId need a corresponding
                RewardPerEpoch and vice versa"
        );
        initialize(_owner);
        uint256 weight = 0;
        rewardDelegatorsAddress = _rewardDelegatorsAddress;
        for(uint256 i=0; i < _networkIds.length; i++) {
            rewardWeight[_networkIds[i]] = _rewardWeight[i];
            weight = weight.add(_rewardWeight[i]);
        }
        totalWeight = weight;
        totalRewardsPerEpoch = _totalRewardsPerEpoch;
        POND = ERC20(_PONDAddress);
        payoutDenomination = _payoutDenomination;
    }
```

Listing 3.1: ClusterRewards:: initialize ()

```
    function addNetwork(bytes32 _networkId, uint256 _rewardWeight) public onlyOwner {
        require(rewardWeight[_networkId] == 0, "ClusterRewards:addNetwork - Network
            already exists");
        require(_rewardWeight != 0, "ClusterRewards:addNetwork - Reward can't be 0");
        rewardWeight[_networkId] = _rewardWeight;
        totalWeight = totalWeight.add(_rewardWeight);
        emit NetworkAdded(_networkId, _rewardWeight);
    }
```

Listing 3.2: ClusterRewards::addNetwork()

We notice that the `initialize()` routine does not emit related `NetworkAdded` events, while the `addNetwork()` routine properly emits the related event.

Recommendation Properly emit the `NetworkAdded` event with accurate information to timely reflect state changes. This is very helpful for external analytics and reporting tools.

Status The issue has been fixed by adding the corresponding `NetworkAdded` events.

3.2 Explicit super Invocation

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `ClusterRewards`, `RewardDelegators`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

Ethereum smart contracts are typically immutable by default. Once they are created, there is no way to alter them, effectively acting as an unbreakable contract among participants. In the meantime, there are several scenarios where there is a need to upgrade the contracts, either to add new functionalities or mitigate potential bugs.

The upgradeability support comes with a few caveats. One important caveat is related to the initialization of new contracts that are just deployed to replace old contracts. Due to the inherent requirement of any proxy-based upgradeability system, no constructors can be used in upgradeable contracts. This means we need to change the constructor of a new contract into a regular function (typically named `initialize()`) that basically executes all the setup logic.

However, a follow-up caveat is that during a contract's lifetime, its constructor is guaranteed to be called exactly once (and it happens at the very moment of being deployed). But a regular function may be called multiple times! In order to ensure that a contract will only be initialized once, we need to guarantee that the chosen `initialize` function can be called only once during the entire lifetime. This guarantee is typically implemented as a modifier named `initializer`.

The `Stake` module in `Marlin` is implemented with the above upgradeability support. In the following, we show again the `initialize()` of `ClusterRewards`. We notice that this routine calls another another function with the same function name, i.e., `initialize()`, from the inherited parent contract of `Ownable`. It is suggested to explicitly mark the inherited function call with `super` or the parent contract name.

```
    function initialize(
        address _owner,
        address _rewardDelegatorsAddress,
        bytes32[] memory _networkIds,
        uint256[] memory _rewardWeight,
        uint256 _totalRewardsPerEpoch,
        address _PONDAddress,
        uint256 _payoutDenomination)
        public
        initializer
    {
        require(
```

```
            _networkIds.length == _rewardWeight.length,
            "ClusterRewards:initialize - Each NetworkId need a corresponding
                RewardPerEpoch and vice versa"
        );
        initialize(_owner);
        uint256 weight = 0;
        rewardDelegatorsAddress = _rewardDelegatorsAddress;
        for(uint256 i=0; i < _networkIds.length; i++) {
            rewardWeight[_networkIds[i]] = _rewardWeight[i];
            weight = weight.add(_rewardWeight[i]);
        }
        totalWeight = weight;
        totalRewardsPerEpoch = _totalRewardsPerEpoch;
        POND = ERC20(_PONDAddress);
        payoutDenomination = _payoutDenomination;
    }
```

Listing 3.3: ClusterRewards:: initialize ()

Recommendation Make the function call to the parent contract explicitly, e.g., with `super`.

Status The issue has been fixed by this commit: `1d2ec4f`.

3.3 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RewardDelegators`
- Category: Time and State [6]
- CWE subcategory: CWE-663 [2]

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` [11] exploit, and the recent `Uniswap/Lendf.Me` hack [10].

We notice there are several occasions the `checks-effects-interactions` principle is violated. Using the `RewardDelegators` as an example, the `withdrawRewards()` 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 187) starts before effecting the update on internal states (lines 188 − 189), hence violating the principle. In this particular case, if the external contract has some hidden logic that may be capable of launching `re-entrancy` via the very same `withdrawRewards()` function.

```
176     function withdrawRewards(address _delegator, address _cluster) public returns(
            uint256) {
177         _updateRewards(_cluster);
178         Cluster memory clusterData = clusters[_cluster];
179         uint256 currentNonce = clusterData.lastRewardDistNonce;
180         Stake memory delegatorStake = clusters[_cluster].delegators[_delegator];
181         uint256 delegatorEffectiveStake = delegatorStake.pond.add(delegatorStake.mpond.
                mul(pondPerMpond));
182         uint256 totalRewards = delegatorStake.pond.mul(clusterData.accPondRewardPerShare
                )
183                                                     .add(delegatorStake.mpond.mul(
                                                            clusterData.
                                                            accMPondRewardPerShare));
184         if(delegatorEffectiveStake != 0 && clusters[_cluster].
                lastDelegatorRewardDistNonce[_delegator] < currentNonce) {
185             uint256 pendingRewards = totalRewards.div(10**30).sub(clusters[_cluster].
                    rewardDebt[_delegator]);
186             if(pendingRewards != 0) {
187                 transferRewards(_delegator, pendingRewards);
188                 clusters[_cluster].lastDelegatorRewardDistNonce[_delegator] =
                        currentNonce;
189                 clusters[_cluster].rewardDebt[_delegator] = totalRewards.div(10**30);
190             }
191             return pendingRewards;
192         }
193         return 0;
194     }
```

Listing 3.4: RewardDelegators::withdrawRewards()

Another similar violation can be found in the `delegate()` and `undelegate()` routines within the same contract.

In the meantime, we should mention that the related `PONDToken` token implements rather standard ERC20 interfaces and the related token contract is not vulnerable or exploitable for `re-entrancy`.

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` best practice. The above three functions can be revised as follows:

```
176     function withdrawRewards(address _delegator, address _cluster) public returns(
            uint256) {
177         _updateRewards(_cluster);
178         Cluster memory clusterData = clusters[_cluster];
179         uint256 currentNonce = clusterData.lastRewardDistNonce;
180         Stake memory delegatorStake = clusters[_cluster].delegators[_delegator];
181         uint256 delegatorEffectiveStake = delegatorStake.pond.add(delegatorStake.mpond.
                mul(pondPerMpond));
```

```
        uint256 totalRewards = delegatorStake.pond.mul(clusterData.accPondRewardPerShare)
                                                .add(delegatorStake.mpond.mul(clusterData.accMPondRewardPerShare));
        if(delegatorEffectiveStake != 0 && clusters[_cluster].lastDelegatorRewardDistNonce[_delegator] < currentNonce) {
            uint256 pendingRewards = totalRewards.div(10**30).sub(clusters[_cluster].rewardDebt[_delegator]);
            if(pendingRewards != 0) {
                clusters[_cluster].lastDelegatorRewardDistNonce[_delegator] = currentNonce;
                clusters[_cluster].rewardDebt[_delegator] = totalRewards.div(10**30);
                transferRewards(_delegator, pendingRewards);
            }
            return pendingRewards;
        }
        return 0;
    }
```

Listing 3.5: RewardDelegators::withdrawRewards()

Status The issue has been fixed by this commit: `18f574b`.

3.4 Improved Sanity Checks For System/Function Parameters

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The Marlin Stake protocol is no exception. Specifically, if we examine the `RewardDelegators` contract, it has defined a number of system-wide risk parameters: `undelegationWaitTime`, `PondRewardFactor`, `MPondRewardFactor`, and `minMPONDStake`.

To elaborate, we show below the related configuration routines in the `RewardDelegators` contract for the above risk parameters.

```
    function updateUndelegationWaitTime(uint256 _undelegationWaitTime) public onlyOwner {
        undelegationWaitTime = _undelegationWaitTime;
    }

```

```
    function updateMinMPONDStake(uint256 _minMPONDStake) public onlyOwner {
        minMPONDStake = _minMPONDStake;
    }

    function updateRewardFactors(uint256 _PONDRewardFactor, uint256 _MPONDRewardFactor)
        public onlyOwner {
        PondRewardFactor = _PONDRewardFactor;
        MPondRewardFactor = _MPONDRewardFactor;
    }
```

Listing 3.6: RewardDelegators.sol

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `undelegationWaitTime` may affect every unstake operation and further lead to unnecessarily long waiting time.

In addition, a number of functions can benefit from more rigorous validation on their arguments. For example, the `feed()` (see the code below) can be improved by requiring both `_clusters` and `_payouts` have the same length.

```
    function feed(bytes32 _networkId, address[] memory _clusters, uint256[] memory
        _payouts) public onlyOwner {
        for(uint256 i=0; i < _clusters.length; i++) {
            clusterRewards[_clusters[i]] = clusterRewards[_clusters[i]].add(
                                                totalRewardsPerEpoch
                                                .mul(rewardWeight[_networkId])
                                                .mul(_payouts[i])
                                                .div(totalWeight)
                                                .div(payoutDenomination)
                                            );
        }
        emit ClusterRewarded(_networkId);
    }
```

Listing 3.7: ClusterRewards::feed()

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. If necessary, also consider emitting relevant events for their changes.

Status The issue has been partially fixed by this commit: `18f574b`.

3.5 Simplified Business Logic in createStash()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `StakeManager`
- Category: Business Logics [5]
- CWE subcategory: CWE-837 [3]

Description

In Marlin Stake, the `StakeManager` contract is tasked with the `stash` creation as well as related delegation and un-delegation. While examining the `stash` creation routine, i.e., `createStash()`, we notice that its business logic can be improved and simplified.

```
    function createStash(
        bytes32[] memory _tokens,
        uint256[] memory _amounts
    ) public returns(bytes32) {
        require(
            _tokens.length == _amounts.length,
            "StakeManager:createStash - each tokenId should have a corresponding amount
                and vice versa"
        );
        uint stashIndex = indices[msg.sender];
        bytes32 stashId = keccak256(abi.encodePacked(msg.sender, stashIndex));
        stashes[stashId] = Stash(msg.sender, address(0), 0, new bytes32[](0));
        // TODO: This can never overflow, so change to + for gas savings
        indices[msg.sender] = stashIndex.add(1);
        uint256 index = stashes[stashId].tokensDelegated.length;
        for(uint256 i=0; i < _tokens.length; i++) {
            require(
                tokenAddresses[_tokens[i]] != address(0),
                "StakeManager:createStash - Invalid tokenId"
            );
            if(_amounts[i] != 0) {
                TokenData memory tokenData = stashes[stashId].amount[_tokens[i]];
                // if someone sends same token 2 times while creating stash
                if(tokenData.amount == 0) {
                    stashes[stashId].tokensDelegated.push(_tokens[i]);
                    stashes[stashId].amount[_tokens[i]] = TokenData(_amounts[i], index);
                    index++;
                } else {
                    stashes[stashId].amount[_tokens[i]].amount = tokenData.amount.add(
                        _amounts[i]);
                }
                _lockTokens(_tokens[i], _amounts[i], msg.sender);
            }
        }
        emit StashCreated(msg.sender, stashId, stashIndex, _tokens, _amounts);
```

```
152         return stashId;
153     }
```

Listing 3.8: StakeManager::createStash()

To elaborate, we show above the code implementation of `createStash()`. When a new `stash` is being created, the `index` (line 132) always starts from 0 and the `amount` member can be accordingly simplified as well.

Recommendation Properly revise `createStash()` for simplified and improved logic.

Status The issue has been fixed by this commit: `5cc87ea`.

3.6 Suggested safeTransfer()/safeTransferFrom() Replacement

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `StakeManager`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transfer()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }
```

```
    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.9: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. To use this library you can add a using SafeERC20 for IERC20. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the meantime, we should mention that the related `PONDToken` token implements rather standard ERC20 interfaces and the related token contract does not share the above-mentioned idiosyncrasy.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `transfer()` and `transferFrom()`.

Status The issue has been resolved. The team has confirmed that those related tokens for interaction are `POND`, `MPOND`, and pool tokens in either `Balancer` or `UniswapV2`. These tokens are rather standard ERC20-compliant tokens without the need of using `safeTransfer()` or `safeTransferFrom()`.

3.7 Other Suggestions

Due to the fact that compiler upgrades might bring unexpected compatibility or inter-version consistencies, it is always suggested to use fixed compiler versions whenever possible. As an example, we highly encourage to explicitly indicate the Solidity compiler version, e.g., `pragma solidity 0.6.0` instead of specifying a range, e.g., `pragma solidity >=0.4.21 <0.7.0`.

In addition, there is a known compiler issue that in all 0.5.x solidity prior to `Solidity 0.5.17`. Specifically, a private function can be overridden in a derived contract by a private function of the same name and types. Fortunately, there is no overriding issue in this code, but we still recommend using `Solidity 0.5.17` or above.

Last but not least, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet.

4 Conclusion

In this audit, we have analyzed the Marlin Stake design and implementation. The protocol aims to build a high-performance programmable network infrastructure that could benefit all kinds of blockchains and DApps. During the audit, we notice that the current `stake` module is well organized and those identified issues are promptly confirmed and fixed.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

[3] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

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

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

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

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

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

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

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

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