SMART CONTRACT AUDIT REPORT

for

Muuu Protocol

Prepared By: Xiaomi Huang

PeckShield

May 6, 2022

Document Properties

Client	Muuu Finance
Title	Smart Contract Audit Report
Target	Muuu
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Shulin Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 6, 2022	Xuxian Jiang	Final Release
1.0-rc1	May 3, 2022	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

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

Muuu Finance is a platform for KGL token holders and Kagla liquidity providers to earn additional interest rewards and Kagla trading fees on their tokens. It is inspired from the Convex Finance with its own customized extensions. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Muuu Protocol

Item	Description
Issuer	Muuu Finance
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 6, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/muuu-finance/protocol.git (9e29ce1)

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

- https://github.com/muuu-finance/protocol.git (b3abef1)

1.2 About PeckShield

PeckShield Inc. [15] 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 [14]:

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

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

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

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

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) [13], 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 Muuu protocol 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 logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	3	■■■
Low	7	■■■■■■■
Informational	1	■
Undetermined	1	■
Total	11	

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 3 medium-severity vulnerabilities, 7 low-severity vulnerabilities, 1 informational suggestion, and 1 undetermined issue.

Table 2.1: Key Muuu Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Just-in-time Balance Inflation For Rewards	Business Logic	Resolved
PVE-002	Medium	Improved Logic in BasicMuuuHolder::execute()	Coding Practices	Resolved
PVE-003	Low	Overflow Mitigation in BaseRewardPool/VirtualBalanceRewardPool	Numeric Errors	Fixed
PVE-004	Low	Two-Step Transfer Of Privileged Account Ownership	Coding Practices	Fixed
PVE-005	Informational	Simplified Logic in getReward()	Business Logic	Fixed
PVE-006	Low	Duplicate Pool Detection and Prevention	Business Logic	Resolved
PVE-007	Low	Timely massUpdatePools During Pool Weight Changes	Business Logic	Resolved
PVE-008	Undetermined	Staking Incompatibility With Deflationary Tokens	Business Logic	Resolved
PVE-009	Low	Reentrancy Risk in BXHPool And airdroppool	Time and State	Resolved
PVE-010	Medium	Trust on Admin Keys	Security Features	Confirmed
PVE-011	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Just-in-time Balance Inflation For Rewards

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: MuuuStakingWrapperAbra, MuKglRari
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

The Muuu protocol has a number of built-in wrappers to tokenize a muuu staked position. For example, the MuuuStakingWrapperAbra contract implements the staking wrapper for the Abracadabra platform and the MuKglRari contract wraps the staking for the Rari's Fuse platform. While reviewing the staking-related rewards logic, we notice the current implementation needs to be improved.

To elaborate, we show below the related _getDepositedBalance() function from the MuuuStakingWrapperAbra contract. As the name indicates, this function is designed to query the current deposited balance, which is then used for various purposes, including the calculation of rewards. It comes to our attention that this deposited balance may interact with external contracts (e.g., cauldrons and/or collateralVault) for their respective balances. However, these external balances may be inflated right before the invocation of this _getDepositedBalance() function and returned back to normal after the invocation. This just-in-time balance inflation may be assisted with flashloans. However, the inflated balance may benefit the actor in largely occupying the rewards currently available for claims.

```
function _getDepositedBalance(address _account) internal view override returns (
    uint256) {
  if (_account == address(0)  _account == collateralVault) {
    return 0;
  }

  if (cauldrons.length == 0) {
    return balanceOf(_account);
```

```
    }

    //add up all shares of all cauldrons
    uint256 share;
    for (uint256 i = 0; i < cauldrons.length; i++) {
      try ICauldron(cauldrons[i]).userCollateralShare(_account) returns (uint256 _share)
          {
        share = share.add(_share);
      } catch {}
    }

    //convert shares to balance amount via bento box
    uint256 collateral = IBentoBox(collateralVault).toAmount(address(this), share, false
        );

    //add to balance of this token
    return balanceOf(_account).add(collateral);
  }
```

Listing 3.1: `MuuuStakingWrapperAbra::_getDepositedBalance()`

Recommendation Improve the above-mentioned functions to properly validate the balance to avoid being manipulated by flashloans. Note the `MuKglRari` contract shares the same issue.

Status This issue has been removed in the following PR: `72`.

3.2 Improved Logic in BasicMuuuHolder::execute()

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Coding Practices [9]
- CWE subcategory: CWE-628 [4]

Description

To flexibly support a variety of operations, the `Muuu` protocol has attached a special function `execute()` to its core contracts, including `BasicMuuuHolder`, `BoosterOwner`, `RewardDeposit`, and `TreasuryFunds`. This special function supports arbitrary input data and is guarded to validate that it can only be invoked by the trusted entity. While analyzing this special function, we notice the current implementation can be improved.

In the following, we use the full implementation of the `execute()` function. As the name indicates, this function is designed to be flexible in executing arbitrary function calls. The flexibility comes from the direct copy of the input data for the use of function invocation, including the destination, the transferred native coins, as well as the input data. It comes to our attention that the use of native coins may be limited due to the way the function is defined without the `payable` modifier. This modifier is necessary if the native coins need to be transferred from the caller to the intended callee! This issue is common in the current `execute()` functions among a number of core contracts, including `BasicMuuuHolder`, `BoosterOwner`, `RewardDeposit`, `TreasuryFunds`, `KaglaVoterProxy`, and `BoosterOwner`.

```
function execute(
  address _to,
  uint256 _value,
  bytes calldata _data
) external returns (bool, bytes memory) {
  require(msg.sender == operator, "!auth");

  (bool success, bytes memory result) = _to.call{ value: _value }(_data);

  return (success, result);
}
```

Listing 3.2: `BasicMuuuHolder::execute()`

Recommendation Correct the above logic by adding the `payable` modifier in the affected contracts.

Status This issue has been removed in the following PR: `64`.

3.3 Overflow Mitigation in BaseRewardPool/VirtualBalanceRewardPool

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Numeric Errors [12]
- CWE subcategory: CWE-190 [1]

Description

The `Muuu` protocol shares an incentivizer mechanism inspired from `Synthetix`. In this section, we focus on a routine, i.e., `rewardPerToken()`, which is responsible for calculating the reward rate for each staked token. And it is part of the `updateReward()` modifier that would be invoked up-front for almost every public function in `BaseRewardPool` to update and use the latest reward rate.

The reason is due to the known potential overflow pitfall when a new oversized reward amount is added into the pool. In particular, as the `rewardPerToken()` routine involves the multiplication of three `uint256` integer, it is possible for their multiplication to have an undesirable overflow (lines 141), especially when the `rewardRate` is largely controlled by an external entity, i.e., `operator` (through the `notifyRewardAmount()` function).

```
function rewardPerToken() public view returns (uint256) {
  if (totalSupply() == 0) {
    return rewardPerTokenStored;
  }
  return
    rewardPerTokenStored.add(
      lastTimeRewardApplicable().sub(lastUpdateTime).mul(rewardRate).mul(1e18).div(
          totalSupply())
    );
}
```

Listing 3.3: `BaseRewardPool::rewardPerToken()`

Apparently, this issue is made possible if the reward amount is given as the argument to `notifyRewardAmount()` such that the calculation of `rewardRate.mul(1e18)` always overflows, hence locking all deposited funds! Note that an authentication check on the caller of `notifyRewardAmount()` greatly alleviates such concern. Currently, only the `operator` address is able to call `notifyRewardAmount()` and this address is set when the contract is deployed. Apparently, if the `operator` is a normal address, it may put users' funds at risk. To mitigate this issue, it is necessary to have the ownership under the governance control and ensure the given reward amount will not be oversized to overflow and lock users' funds.

Recommendation Mitigate the potential overflow risk in the various reward pools, including `BaseRewardPool`, `muuuRewardPool`, and `VirtualBalanceRewardPool`.

Status This issue has been fixed by following the above suggestion in the following PR: `64`.

3.4 Two-Step Transfer Of Privileged Account Ownership

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Security Features [8]
- CWE subcategory: CWE-282 [2]

Description

The `Muuu` protocol has a number of contracts that implement a rather basic access control mechanism that allows a privileged account, i.e., `owner`, to be granted exclusive access to typically sensitive functions (e.g., the setting of protocol parameters). Because of the privileged access and the implications of these sensitive functions, the `owner` account is essential for the protocol-level safety and operation. In the following, we elaborate with the `owner` account.

Using the `MuuuStakingProxyV2` contract as an example, two related functions, i.e., `setPendingOwner()/applyPendingOwner()`, are provided to allow for possible `owner` updates. However, current implementation achieves its goal with these two function calls from the (same) current `owner`. This is reasonable under the assumption that the `pendingOwner` parameter is always correctly provided. However, in the unlikely situation, when an incorrect `pendingOwner` is provided, the contract owner may be forever lost, which might be devastating for the protocol-wide operation and maintenance.

As a common best practice, instead of achieving the owner update from the current owner, it is suggested to split the operation into two steps. The first step initiates the owner update intent from the current owner and the second step accepts and materializes the update from the pending owner. Both steps should be executed in two separate transactions. By doing so, it can greatly alleviate the concern of accidentally transferring the contract governor to an uncontrolled address. In other words, this two-step procedure ensures that a owner public key cannot be nominated unless there is an entity that has the corresponding private key. This is explicitly designed to prevent unintentional errors in the owner transfer process.

```
function setPendingOwner(address _po) external {
  require(msg.sender == owner, "!auth");
  pendingOwner = _po;
}

function applyPendingOwner() external {
  require(msg.sender == owner, "!auth");
  require(pendingOwner != address(0), "invalid owner");

  owner = pendingOwner;
  pendingOwner = address(0);
}
```

Listing 3.4: `MuuuStakingProxyV2::setPendingOwner()/applyPendingOwner()`

Recommendation Implement the two-step approach for owner update (or transfer) that involves both the current owner and the pending owner.

Status This issue has been fixed by taking the above two-step approach in the following PR: `64`.

3.5 Simplified Logic in getReward()

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Business Logic [10]
- CWE subcategory: CWE-770 [6]

Description

As mentioned earlier, the `Muuu` protocol shares an incentivizer mechanism inspired from `Synthetix`, which has the `getReward()` routine to obtain the calling user's staking rewards. The logic is rather straightforward in calculating possible reward, which, if not zero, is then allocated to the calling (staking) user.

Our examination shows that the current implementation logic can be further optimized. In particular, the `getReward()` routine has a modifier, i.e., `updateReward(_account))`, which timely updates the given user's (earned) rewards in `rewards[_account]` (line 125).

```
function getReward(address _account, bool _claimExtras)
  public
  updateReward(_account)
  returns (bool)
{
  uint256 reward = earned(_account);
  if (reward > 0) {
    rewards[_account] = 0;
    rewardToken.safeTransfer(_account, reward);
    IDeposit(operator).rewardClaimed(pid, _account, reward);
    emit RewardPaid(_account, reward);
  }

  //also get rewards from linked rewards
  if (_claimExtras) {
    for (uint i = 0; i < extraRewards.length; i++) {
      IRewards(extraRewards[i]).getReward(_account);
    }
  }
  return true;
}
```

Listing 3.5: `BasicRewardPool::getReward()`

```
121 modifier updateReward(address account) {
122   rewardPerTokenStored = rewardPerToken();
123   lastUpdateTime = lastTimeRewardApplicable();
124   if (account != address(0)) {
125     rewards[account] = earned(account);
```

```
            userRewardPerTokenPaid[account] = rewardPerTokenStored;
        }
        _;
    }
```

Listing 3.6: `BasicRewardPool::updateReward()`

Having the modifier `updateReward()`, there is no need to re-calculate the earned reward for the given user. In other words, we can simply re-use the calculated `rewards[_account]` and assign it to the `reward` variable (line 252). This issue is also applicable to the `VirtualBalanceRewardPool` contract.

Recommendation Avoid the duplicated calculation of the caller's reward in `getReward()`, which also leads to (small) beneficial reduction of associated gas cost.

```
247     function getReward(address _account, bool _claimExtras)
248         public
249         updateReward(_account)
250         returns (bool)
251     {
252         uint256 reward = rewards[_account];
253         if (reward > 0) {
254             rewards[_account] = 0;
255             rewardToken.safeTransfer(_account, reward);
256             IDeposit(operator).rewardClaimed(pid, _account, reward);
257             emit RewardPaid(_account, reward);
258         }

260         //also get rewards from linked rewards
261         if (_claimExtras) {
262             for (uint i = 0; i < extraRewards.length; i++) {
263                 IRewards(extraRewards[i]).getReward(_account);
264             }
265         }
266         return true;
267     }
```

Listing 3.7: Revised `BasicRewardPool::getReward()`

Status This issue has been fixed by following the above suggestion in the following PR: `64`.

3.6 Duplicate Pool Detection and Prevention

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: MuuuMasterChef
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

The Muuu protocol provides incentive mechanisms that reward the staking of supported assets with certain reward tokens. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. Each pool has its allocPoint*100%/totalAllocPoint share of scheduled rewards and the rewards for stakers are proportional to their share of LP tokens in the pool.

In current implementation, there are a number of concurrent pools that share the rewarded tokens and more can be scheduled for addition (via a privileged function). To accommodate these new pools, the design has the necessary mechanism in place that allows for dynamic additions of new staking pools that can participate in being incentivized as well.

The addition of a new pool is implemented in add(), whose code logic is shown below. It turns out it did not perform necessary sanity checks in preventing a new pool but with a duplicate token from being added. Though it is a privileged interface (protected with the modifier onlyOwner), it is still desirable to enforce it at the smart contract code level, eliminating the concern of wrong pool introduction from human omissions.

```
function add(
  uint256 _allocPoint,
  IERC20 _lpToken,
  IRewarder _rewarder,
  bool _withUpdate
) public onlyOwner {
  if (_withUpdate) {
    massUpdatePools();
  }
  uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
  totalAllocPoint = totalAllocPoint.add(_allocPoint);
  poolInfo.push(
    PoolInfo({
      lpToken: _lpToken,
      allocPoint: _allocPoint,
      lastRewardBlock: lastRewardBlock,
      accMuuuPerShare: 0,
      rewarder: _rewarder
    })
```

```
    );
  }
```

Listing 3.8: `MuuuMasterChef::add()`

Recommendation Detect whether the given pool for addition is a duplicate of an existing pool. The pool addition is only successful when there is no duplicate.

```
  function checkPoolDuplicate(IERC20 _lpToken) public {
    uint256 length = poolInfo.length;
    for (uint256 pid = 0; pid < length; ++pid) {
      require(poolInfo[_pid].lpToken != _lpToken, "add: existing pool?");
    }
  }

  function add(
    uint256 _allocPoint,
    IERC20 _lpToken,
    IRewarder _rewarder,
    bool _withUpdate
  ) public onlyOwner {
    if (_withUpdate) {
      massUpdatePools();
    }
    checkPoolDuplicate(_lpToken);
    uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    poolInfo.push(
      PoolInfo({
        lpToken: _lpToken,
        allocPoint: _allocPoint,
        lastRewardBlock: lastRewardBlock,
        accMuuuPerShare: 0,
        rewarder: _rewarder
      })
    );
  }
```

Listing 3.9: Revised `MuuuMasterChef::add()`

We point out that if a new pool with a duplicate LP token can be added, it will likely cause a havoc in the distribution of rewards to the pools and the stakers.

Status This issue has been removed in the following PR: `69`.

3.7 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: MuuuMasterChef
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

As mentioned earlier, the Muuu protocol provides incentive mechanisms that reward the staking of supported assets. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

The reward pools can be dynamically added via add() and the weights of supported pools can be adjusted via set(). When analyzing the pool weight update routine set(), we notice the need of timely invoking massUpdatePools() to update the reward distribution before the new pool weight becomes effective.

```
function set(
  uint256 _pid,
  uint256 _allocPoint,
  IRewarder _rewarder,
  bool _withUpdate,
  bool _updateRewarder
) public onlyOwner {
  if (_withUpdate) {
    massUpdatePools();
  }
  totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
  poolInfo[_pid].allocPoint = _allocPoint;
  if (_updateRewarder) {
    poolInfo[_pid].rewarder = _rewarder;
  }
}
```

Listing 3.10: MuuuMasterChef::set()

If the call to massUpdatePools() is not immediately invoked before updating the pool weights, certain situations may be crafted to create an unfair reward distribution. Moreover, a hidden pool without any weight can suddenly surface to claim unreasonable share of rewarded tokens. Fortunately, this interface is restricted to the owner (via the onlyOwner modifier), which greatly alleviates the concern.

Recommendation Timely invoke `massUpdatePools()` when any pool's weight has been updated. In fact, the third parameter (`_withUpdate`) to the `set()` routine can be simply ignored or removed.

```
function set(
  uint256 _pid,
  uint256 _allocPoint,
  IRewarder _rewarder,
  bool _withUpdate,
  bool _updateRewarder
) public onlyOwner {
  massUpdatePools();
  totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
  poolInfo[_pid].allocPoint = _allocPoint;
  if (_updateRewarder) {
    poolInfo[_pid].rewarder = _rewarder;
  }
}
```

Listing 3.11: Revised `MuuuMasterChef::set()`

Status This issue has been removed in the following PR: `69`.

3.8 Staking Incompatibility With Deflationary Tokens

- ID: PVE-008
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: `MuuuMasterChef`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

In the `Muuu` protocol, the `MuuuMasterChef` contract is designed to take users' assets and deliver rewards depending on their share. In particular, one interface, i.e., `deposit()`, accepts asset transfer-in and records the depositor's balance. Another interface, i.e, `withdraw()`, allows the user to withdraw the asset with necessary bookkeeping under the hood. For the above two operations, i.e., `deposit()` and `withdraw()`, the contract using the `safeTransfer()/safeTransferFrom()` routines to transfer assets into or out of its pool. This routine works as expected with standard ERC20 tokens: namely the pool's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
function deposit(uint256 _pid, uint256 _amount) public {
  PoolInfo storage pool = poolInfo[_pid];
  UserInfo storage user = userInfo[_pid][msg.sender];
  updatePool(_pid);
```

```
    if (user.amount > 0) {
      uint256 pending = user.amount.mul(pool.accMuuuPerShare).div(1e12).sub(user.
          rewardDebt);
      safeRewardTransfer(msg.sender, pending);
    }
    pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
    user.amount = user.amount.add(_amount);
    user.rewardDebt = user.amount.mul(pool.accMuuuPerShare).div(1e12);

    //extra rewards
    IRewarder _rewarder = pool.rewarder;
    if (address(_rewarder) != address(0)) {
      _rewarder.onReward(_pid, msg.sender, msg.sender, 0, user.amount);
    }

    emit Deposit(msg.sender, _pid, _amount);
  }
```

Listing 3.12: `MuuuMasterChef::deposit()`

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every transfer. As a result, this may not meet the assumption behind asset-transferring routines. In other words, the above operations, such as `deposit()` and `withdraw()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of the pool and affects protocol-wide operation and maintenance.

Specially, if we take a look at the `updatePool()` routine. This routine calculates `pool.accMuuuPerShare` via dividing the reward by `lpSupply`, where the `lpSupply` is derived from `pool.lpToken.balanceOf(address(this))` (line 163). Because the balance inconsistencies of the pool, the `lpSupply` could be 1 `Wei` and thus may yield a huge `pool.accMuuuPerShare` as the final result, which dramatically inflates the pool's reward.

```
158  function updatePool(uint256 _pid) public {
159    PoolInfo storage pool = poolInfo[_pid];
160    if (block.number <= pool.lastRewardBlock) {
161      return;
162    }
163    uint256 lpSupply = pool.lpToken.balanceOf(address(this));
164    if (lpSupply == 0) {
165      pool.lastRewardBlock = block.number;
166      return;
167    }
168    uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
169    uint256 muuuReward = multiplier.mul(rewardPerBlock).mul(pool.allocPoint).div(
         totalAllocPoint);
170    //muuu.mint(address(this), muuuReward);
171    pool.accMuuuPerShare = pool.accMuuuPerShare.add(muuuReward.mul(1e12).div(lpSupply));
172    pool.lastRewardBlock = block.number;
```

```
}
```

Listing 3.13: MuuuMasterChef::updatePool()

One mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in safeTransfer() or safeTransferFrom() will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the safeTransfer() or safeTransferFrom() is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into Muuu for support. However, certain existing stable coins may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation Check the balance before and after the safeTransfer() or safeTransferFrom() call to ensure the book-keeping amount is accurate. An alternative solution is using non-deflationary tokens as collateral but some tokens (e.g., USDT) allow the admin to have the deflationary-like features kicked in later, which should be verified carefully.

Status This issue has been removed in the following PR: 69.

3.9 Reentrancy Risk in MuuuMasterChef

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: MuuuMasterChef
- Category: Time and State [11]
- CWE subcategory: CWE-663 [5]

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 [17] exploit, and the recent Uniswap/Lendf.Me hack [16].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `MuuuMasterChef` as an example, the `emergencyWithdraw()` 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 241) starts before effecting the update on the internal state (lines 243 – 244), 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 same entry function.

```
238  function emergencyWithdraw(uint256 _pid) public {
239    PoolInfo storage pool = poolInfo[_pid];
240    UserInfo storage user = userInfo[_pid][msg.sender];
241    pool.lpToken.safeTransfer(address(msg.sender), user.amount);
242    emit EmergencyWithdraw(msg.sender, _pid, user.amount);
243    user.amount = 0;
244    user.rewardDebt = 0;
245
246    //extra rewards
247    IRewarder _rewarder = pool.rewarder;
248    if (address(_rewarder) != address(0)) {
249      _rewarder.onReward(_pid, msg.sender, msg.sender, 0, 0);
250    }
251  }
```

Listing 3.14: `MuuuMasterChef::emergencyWithdraw()`

Note that other routines share the same issue, including `deposit()`, `withdraw()`, and `emergencyWithdraw()`.

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible `re-entrancy`.

Status This issue has been removed in the following PR: `69`.

3.10 Trust Issue of Admin Keys

- ID: PVE-010
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [8]
- CWE subcategory: CWE-287 [3]

Description

The `Muuu` protocol has a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., pool addition, reward adjustment, and parameter setting). It also has the privilege to control or govern the flow of assets among various protocol components. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
function setRegistry(address _registry) external {
  require(msg.sender == owner(), "!auth");
  registry = _registry;
}

function setRewardMultiplier(uint256 _rewardMultiplier) external onlyOwner {
  require(_rewardMultiplier > 0 &&  _rewardMultiplier <= MULTIPLIER_DENOMINATOR , "rewardMultiplier should be 0-100000");
  rewardMultiplier = _rewardMultiplier;
}

function setVoteOwnership(address _voteOwnership) external onlyOwner {
  require(_voteOwnership != address(0), "voteOwnership should not be zero address");
  voteOwnership = _voteOwnership;
}

function setVoteParameter(address _voteParameter) external onlyOwner {
  require(_voteParameter != address(0), "voteParameter should not be zero address");
  voteParameter = _voteParameter;
}
```

Listing 3.15: Example Privileged Operations in `Booster`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Promptly transfer the `owner` privilege to the intended `DAO`-like governance contract. And 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.

3.11 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-011
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [9]
- CWE subcategory: CWE-628 [4]

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 the following, we examine the `transfer()` routine and related 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 3.16: 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. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `withdrawEmergency()` routine in the `LockerAdmin` contract. If the `USDT` token is supported as `tokenaddress`, the unsafe version of `IERC20(_tokenAddress).transfer(operator, _tokenAmount)` (line 77) may revert as there is no return value in the `USDT` token contract's `transfer()/transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
76  function transferToken(address _tokenAddress, uint256 _tokenAmount) public onlyOwner {
77    IERC20(_tokenAddress).transfer(operator, _tokenAmount);
78  }
```

Listing 3.17: LockerAdmin::transferToken()

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`. Note the `safeApprove()` counterpart may need to invoke twice: the first time resets the allowance to 0 and the second time sets the intended spending allowance.

Status This issue has been resolved in the following PR: 72.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Muuu` protocol, which is a platform for `KGL` token holders and `Kagla` liquidity providers to earn additional interest rewards and `Kagla` trading fees on their tokens. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

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

References

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

[2] MITRE. CWE-282: Improper Ownership Management. https://cwe.mitre.org/data/definitions/282.html.

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

[4] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

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

[6] MITRE. CWE-770: Allocation of Resources Without Limits or Throttling. https://cwe.mitre.org/data/definitions/770.html.

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

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

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

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

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

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

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

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

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

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

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