SMART CONTRACT AUDIT REPORT

for

Amara Finance

Prepared By: Patrick Lou

Hangzhou, China
April 4, 2022

Document Properties

Client	Amara Foundation
Title	Smart Contract Audit Report
Target	Amara Finance
Version	1.0
Author	Shulin Bie
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 4, 2022	Shulin Bie	Final Release
1.0-rc	March 18, 2022	Shulin Bie	Release Candidate

Contact

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

Name	Patrick Lou
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction 4
- 1.1 About Amara Finance . . . 4
- 1.2 About PeckShield . . . 5
- 1.3 Methodology . . . 5
- 1.4 Disclaimer . . . 7

2 Findings 9
- 2.1 Summary . . . 9
- 2.2 Key Findings . . . 10

3 Detailed Results 11
- 3.1 Possible Costly *AlpToken* From Improper Deposit Initialization . . . 11
- 3.2 Timely massUpdateRewards In Multiple Routines . . . 13
- 3.3 Accommodation Of Non-ERC20-Compliant Tokens . . . 14
- 3.4 Potential Sandwich/MEV Attack For _safeSwap() . . . 16
- 3.5 Duplicate Pool Detection and Prevention . . . 17
- 3.6 Trust Issue Of Admin Keys . . . 18

4 Conclusion 20

References 21

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Amara Finance, 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 Amara Finance

Amara Finance is a yield farming aggregator running on BNB Chain (previously BSC), with the goal of optimizing DeFi users' yield farming at the lowest possible cost. The protocol has a number of built-in farming strategies and supports multiple farming pools (e.g., PancakeSwap). The protocol also has its utility token MARA, which is distributed to protocol users according to their engagement or contribution.

Table 1.1: Basic Information of Amara Finance

Item	Description
Target	Amara Finance
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	April 4, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Please note that this audit only covers the contracts/farm sub-directory.

- https://github.com/xizho10/mara-auto.git (d2792d8)

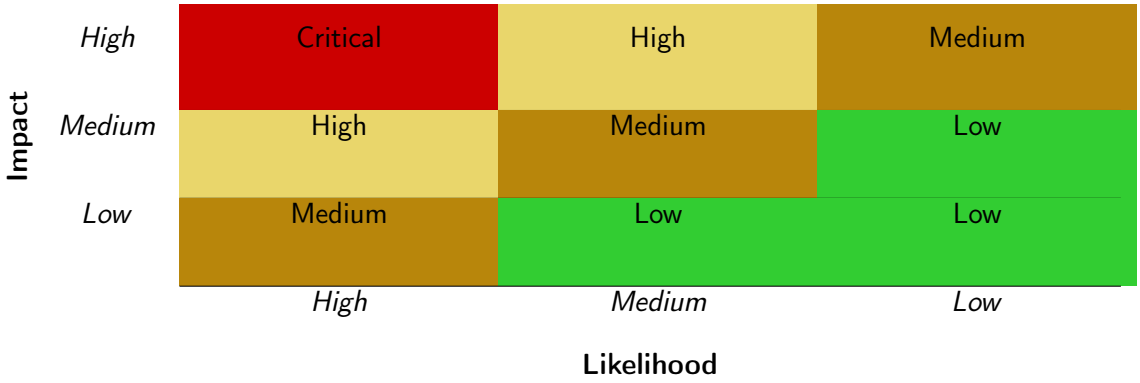
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/xizho10/mara-auto.git (b507feb)

1.2 About PeckShield

PeckShield Inc. [13] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification



1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [12]:

- 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

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

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) [11], 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 Amara Finance implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■■
Low	4	■■■■
Informational	0	
Total	6	

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities and 4 low-severity vulnerabilities.

Table 2.1: Key Amara Finance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Possible Costly *AlpToken* From Improper Deposit Initialization	Time and State	Confirmed
PVE-002	Low	Timely massUpdateRewards In Multiple Routines	Business Logic	Fixed
PVE-003	Low	Accommodation Of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-004	Low	Potential Sandwich/MEV Attack For _safeSwap()	Time and State	Confirmed
PVE-005	Low	Duplicate Pool Detection and Prevention	Business Logic	Fixed
PVE-006	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed

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 Possible Costly *AlpToken* From Improper Deposit Initialization

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `FarmManagerV1/Farm_PCS/Farm_Token`
- Category: Time and State [7]
- CWE subcategory: CWE-362 [3]

Description

By design, the `Amara Finance` protocol organizes a number of farming pools into which supported assets can be staked and implements an incentive mechanism that rewards the staking of different farming pools with the `MARA` token and `rewardToken`. The `FarmManagerV1` contract is one of the main entries for interaction with users. Meanwhile, the `FarmBase` contract implements the standard farming strategy, and various `Farm_PCS/Farm_Token` contracts inherit from it to implement specific farming strategies.

In particular, one routine, i.e., `Farm_PCS::deposit()`, is called inside the `FarmManagerV1::deposit()` routine when the user stakes his assets. And the corresponding `AlpToken` is minted to the staker as his pool shares. While examining the share calculation with the given stakes in the `Farm_PCS::deposit()` routine, we notice an issue that may unnecessarily make the pool share extremely expensive and bring hurdles (or even causes loss) for later stakers. Specifically, the issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
39      function deposit(uint256 _pid, uint256 _lpTokenAmount) public nonReentrant {
40          FarmManagerInfo(managerInfoAddress).massUpdateRewards();
41          if (_lpTokenAmount > 0) {
42              PoolInfo memory pool = _getPool(_pid);
43              require(pool.status == PoolStatus.ACTIVE, "can't deposit to this pool");
```

```
44              pool.lpToken.safeTransferFrom(address(msg.sender), address(this),
                    _lpTokenAmount);
45          }
46          _depositInternal(_pid, _lpTokenAmount);
47      }
48
49      function _depositInternal(uint256 _pid, uint256 _lpTokenAmount) internal {
50          PoolInfo memory pool = _getPool(_pid);
51
52          if (_lpTokenAmount > 0) {
53              pool.lpToken.safeIncreaseAllowance(pool.farmAddress, _lpTokenAmount);
54              uint256 aLPAdded = IFarmBase(pool.farmAddress).deposit(msg.sender,
                    _lpTokenAmount);
55              pool.aLPToken.mint(msg.sender, aLPAdded);
56              emit Deposit(msg.sender, _pid, _lpTokenAmount);
57          }
58      }
```

Listing 3.1: `FarmManagerV1::deposit()`

```
54      function deposit(
55          address _userAddress,
56          uint256 _lpTokenAmount
57      ) public override onlyOwner nonReentrant whenNotPaused returns (uint256) {
58          if (userLastDepositedTimestamp[_userAddress] == 0) {
59              users.push(_userAddress);
60          }
61          userLastDepositedTimestamp[_userAddress] = block.timestamp;
62
63          IERC20(lpTokenAddress).safeTransferFrom(address(msg.sender), address(this),
                _lpTokenAmount);
64
65          uint256 aLPAdded = _lpTokenAmount;
66          uint256 aLPTotal = IERC20(aLPTokenAddress).totalSupply();
67          if (lpAmountTotal > 0 && aLPTotal > 0) {
68              aLPAdded = _lpTokenAmount * aLPTotal * entranceFeeFactor / lpAmountTotal /
                    entranceFeeFactorMax;
69          }
70
71          if (isAuto) {
72              _farm();
73          } else {
74              lpAmountTotal = lpAmountTotal + _lpTokenAmount;
75          }
76          return aLPAdded;
77      }
```

Listing 3.2: `Farm_PCS::deposit()`

Specifically, when the pool is being initialized, the share value directly takes the value of `_lpTokenAmount` (line 65), which is under control by the malicious actor. As this is the first stake, the current total supply equals the calculated `uint256 aLPAdded = _lpTokenAmount = 1WEI`. With that, the actor can fur-

ther transfer a huge amount of `lpTokenAddress` tokens to `Farm_PCS` contract with the goal of making per share extremely expensive.

An extremely expensive share can be very inconvenient to use as a small number of $1WEI$ may denote a large value. Furthermore, it can lead to precision issue in truncating the computed share value for staked assets. If truncated to be zero, the staked assets are essentially considered dust and kept by the pool with returning `0`. That is to say, the staker will get `0 AlpToken` for this stake.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

Recommendation Revise current execution logic of `Farm_PCS::deposit()/Farm_Token::deposit()` to defensively calculate the share amount when the pool is being initialized. An alternative solution is to ensure guarded launch that safeguards the first stake to avoid being manipulated. Note that another routine, i.e., `Farm_Token::deposit()`, shares the similar issue.

Status The issue has been confirmed by the team.

3.2 Timely massUpdateRewards In Multiple Routines

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `FarmManagerInfo`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

The `Amara Finance` protocol implements an incentive mechanism that rewards the staking of supported assets with the `MARA` token. The rewards are carried out by designating a number of staking pools. The staking users are rewarded in proportional to their staking assets in the pool.

The reward rate (per block) of the `MARA` token can be adjusted via the `FarmManagerInfo::setMaraPerBlock()` routine. When analyzing its logic, we notice the lack of timely invoking `massUpdateRewards()` to update each pool reward status before the new reward-related configuration becomes effective. If the call to `massUpdateRewards()` is not immediately invoked before updating the reward rate, certain situations may be crafted to create an unfair reward distribution.

```
57      function setMaraPerBlock(uint256 _maraPerBlock) public onlyOwner {
58          maraPerBlock = _maraPerBlock;
```

```
59          emit SetMaraPerBlock(_maraPerBlock);
60      }
```

Listing 3.3: FarmManagerInfo::setMaraPerBlock()

Note that other routines, i.e., AlpToken::_transfer(), can be similarly improved.

Recommendation Timely invoke massUpdateRewards() in above-mentioned routines.

Status The issue has been addressed by the following commit: f207433.

3.3 Accommodation Of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: AlpToken
- Category: Coding Practices [8]
- CWE subcategory: CWE-1109 [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."*

```
64      function transfer(address _to, uint _value) returns (bool) {
65          //Default assumes totalSupply can't be over max (2^256 - 1).
66          if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
67          balances[msg.sender] -= _value;
68          balances[_to] += _value;
69          Transfer(msg.sender, _to, _value);
70          return true;
71          } else { return false; }
72      }
73      function transferFrom(address _from, address _to, uint _value) returns (bool) {
74          if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
                balances[_to] + _value >= balances[_to]) {
75              balances[_to] += _value;
```

```
76              balances[_from] -= _value;
77              allowed[_from][msg.sender] -= _value;
78              Transfer(_from, _to, _value);
79              return true;
80          } else { return false; }
81      }
```

Listing 3.4: 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 transferFrom() as well, i.e., safeTransferFrom().

In the following, we show the AlpToken::withdraw() routine. If the USDT token is supported as _token, the unsafe version of _token.transfer(address(msg.sender), _amount) (line 68) may revert as there is no return value in the USDT token contract's transfer() implementation (but the IERC20 interface expects a return value). We may intend to replace _token.transfer(address(msg.sender), _amount) (line 68) with safeTransfer().

```
67      function withdraw(ERC20 _token, uint256 _amount) public onlyOwner {
68          _token.transfer(address(msg.sender), _amount);
69      }
```

Listing 3.5: AlpToken::withdraw()

Note that other routines, i.e., AlpToken::receiveToken()/_claimCurrentReward()/_safeTransferRewardToken(), can be similarly improved.

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related transfer() and transferFrom().

Status The issue has been addressed by the following commit: 430a7e0.

3.4 Potential Sandwich/MEV Attack For _safeSwap()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Farm_PCS/FarmBase
- Category: Time and State [10]
- CWE subcategory: CWE-682 [4]

Description

While examining the FarmBase contract, we notice there is a routine (i.e., _safeSwap()) that can be improved with effective slippage control. To elaborate, we show below the related code snippet of the FarmBase contract. According to the design, the _safeSwap() function is used to swap a certain token (specified by the input _path) to another one. In the function, the swapExactTokensForTokensSupporting-FeeOnTransferTokens() function of PancakeSwap is called (lines 214 - 221) to swap the exact token to another one. However, we observe the second input amountOutMin parameter is calculated according to the current state of the PancakeSwap pool (line 211), which may have been price-manipulated. In other words, the slippage control is ineffective and is therefore vulnerable to possible front-running attacks.

```
202     function _safeSwap(
203         address _uniRouterAddress,
204         uint256 _amountIn,
205         uint256 _slippageFactor,
206         address[] memory _path,
207         address _to,
208         uint256 _deadline
209     ) internal virtual {
210         uint256[] memory amounts =
211         IPancakeRouter02(_uniRouterAddress).getAmountsOut(_amountIn, _path);
212         uint256 amountOut = amounts[amounts.length - 1];
213
214         IPancakeRouter02(_uniRouterAddress)
215         .swapExactTokensForTokensSupportingFeeOnTransferTokens(
216             _amountIn,
217             amountOut * _slippageFactor / 1000,
218             _path,
219             _to,
220             _deadline
221         );
222     }
```

Listing 3.6: FarmBase::_safeSwap()

Note that another routine, i.e, Farm_PCS::earn(), that uses addLiquidity() also lacks necessary slippage control.

Recommendation Improve the above-mentioned routines by adding effective slippage control.

Status The issue has been confirmed by the team.

3.5 Duplicate Pool Detection and Prevention

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `FarmManagerInfo`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

The `Amara Finance` protocol organizes a number of farming pools into which supported assets can be staked and implements an incentive mechanism that rewards the staking of different farming pools with the `MARA` token and `rewardToken`. The `FarmManagerInfo` contract is designed to manage the farming pools information.

In current implementation, there are a number of concurrent pools that share the rewarded tokens and more can be scheduled for addition (via a proper governance procedure or moderated by a privileged account). 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 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.

```
66      function add(
67          ERC20 _lpToken,
68          address _farmAddress
69      ) public onlyOwner {
70          require(manager != address(0), "manager not init");
71          uint256 lastMaraRewardBlock = block.number > startBlock ? block.number :
                startBlock;
72          string memory alpTokenSymbol = string(abi.encodePacked("a", _lpToken.symbol()));
73          string memory alpTokenName = string(abi.encodePacked("A", _lpToken.name()));
74          AlpToken alpToken = alpTokenDeployer.createNewAlpToken(
75              alpTokenName,
76              alpTokenSymbol,
77              _lpToken,
78              manager,
```

```
79              address(msg.sender)
80          );
81          PriceHelper(IFarmBase(_farmAddress).priceHelperAddress()).addAlpTokens(address(
                alpToken));
82          IFarmBase(_farmAddress).setAlpToken(address(alpToken));
83
84          poolInfos.push(
85              PoolInfo({
86          status : PoolStatus.INACTIVE,
87          lpToken : _lpToken,
88          aLPToken : alpToken,
89          lastMaraRewardBlock : lastMaraRewardBlock,
90          farmAddress : _farmAddress,
91          lendingPoolAddress : address(0)
92          })
93          );
94
95          emit Add(address(_lpToken), _lpToken.symbol(), address(alpToken));
96      }
```

Listing 3.7: `FarmManagerInfo::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.

Status The issue has been addressed by the following commit: `f92a8aa`.

3.6 Trust Issue Of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

In the `Amara Finance` protocol, there is a privileged account that plays a critical role in governing and regulating the protocol-wide operations (e.g., transfer the staked token out of the contract). In the following, we show the representative functions potentially affected by the privilege of the account.

```
193     function inCaseTokensGetStuck(
194         address _token,
195         uint256 _amount,
196         address _to
197     ) public virtual onlyOwner {
198         IERC20(_token).safeTransfer(_to, _amount);
```

```
199     }
```

Listing 3.8: FarmBase::inCaseTokensGetStuck()

```
98      function setPoolStatus(uint256 _pid, PoolStatus _status) external onlyOwner {
99          massUpdateRewards();
100         PoolInfo storage pool = poolInfos[_pid];
101         pool.status = _status;
102         emit SetPoolStatus(_pid, _status);
103     }
104
105     function setFarmAddress(uint256 _pid, address _farmAddress) public onlyOwner {
106         poolInfos[_pid].farmAddress = _farmAddress;
107         emit SetFarmAddress(_pid, _farmAddress);
108     }
```

Listing 3.9: FarmManagerInfo::setPoolStatus()&&setFarmAddress()

```
32      function setProxy(address token, address proxy) public onlyOwner {
33          tokenToUsdaProxys[token] = proxy;
34          emit SetProxy(token, proxy);
35      }
```

Listing 3.10: PriceHelper::setProxy()

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

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

Status This issue has been confirmed by the team. The team intends to introduce multi-sig mechanisms to mitigate this issue when the protocol is deployed on the mainnet.

4 Conclusion

In this audit, we have analyzed the Amara Finance design and implementation. Amara Finance is a yield farming aggregator running on BNB Chain (previously BSC), with the goal of optimizing DeFi users' yield farming at the lowest possible cost, which provides a number of built-in farming strategies and supports multiple farming pools (e.g., PancakeSwap). 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

[3] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

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

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

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

[7] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.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: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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