SMART CONTRACT AUDIT REPORT

for

Acet Finance (Farming)

Prepared By: Yiqun Chen

PeckShield

October 04, 2021

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	October 04, 2021	Jing Wang	Final Release
1.0-rc	October 04, 2021	Jing Wang	Release Candidate

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the farming support in the `Acet Finance` 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 Acet Finance

The `Acet Finance` is a decentralized liquidity mining platform that rewards the staking of supported assets with certain reward tokens. When compared with other framing protocols, the farming pool in `Acet` allows the `LP` provider to select different locking time periods when depositing their `LPs` tokens into the liquidity pool. The earned rewards will vary depending on the chosen locking time period and the actual locking time.

The basic information of the `Acet Farming` protocol is as follows:

Table 1.1: Basic Information of The `Acet Farming` Protocol

Item	Description
Name	Acet
Website	https://www.acet.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 04, 2021

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

- https://github.com/acetdefi/acet-farm.git (6d66070)

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.

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

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	1	■
Informational	2	■ ■
Total	4	

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 1 medium-severity vulnerability, 1 low-severity vulnerability, and 2 informational recommendations.

Table 2.1: Key Acet Finance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Information	Improved Logic of toPool()	Coding Practices	Confirmed
PVE-002	Information	Redundant State/Code Removal	Coding Practices	Confirmed
PVE-003	Low	Incompatibility with Deflationary Tokens	Business Logics	Confirmed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

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 Improved Logic of toPool()

- ID: PVE-001
- Severity: Information
- Likelihood: N/A
- Impact: N/A
- Target: AcetAdaptor
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The farming pool in Acet provides an AcetAdaptor contract to manage the list of farming pools and the actual mints or burns of the protocol token, i.e., AcetToken, when LPs tokens are deposited or withdrawn. When we examine the logic of an internal toPool() routine, it comes to our attention that its logic may be improved. To elaborate, we show below the related code snippet.

```
    function toPool(uint _amount, uint _funtion) public {
        uint checkAddress = 0;
        uint checkFlagEnable = 0;
         for (uint i = 0; i < candidateCount; i++) {
              Pool storage people = peoples[i];
                if (people.addr == msg.sender) {
                    checkAddress = 1;
                    checkFlagEnable = people.poolStatusFlag;
                }
        }
        require(checkAddress == 1, "Pool doesn't exist");
        if (checkFlagEnable == 1) {
            if (_funtion == 1) {
                revert("Pool has been limit");
            }
        }
        act._mint(address(msg.sender), _amount);
    }
```

Listing 3.1: AcetAdaptor::toPool()

We notice that there is a for-loop to find the specific pool by msg.sender and this for-loop may be stopped when the target pool is found. However, the current logic of the toPool() routine continues to check the pool's address until the end is reached. Note that other routines toDev(), toBurn() and updateSpecificPool() share the same issue.

Recommendation Add break when find the target pool to save gas.

Status This issue has been confirmed. Considering the contract has been deployed and is not upgradeable, the team decides to leave it as it is.

3.2 Redundant State/Code Removal

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

In the Acet protocol, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed. For example, the member _currentExtraReward and _extraRewards are defined but not used throughout the entire Pool contract. The currentExtraRewardSum() and currentExtraRewardByID() routines share the same issue as they only perform some calculations of the _extraRewards. If there is no actual uses of these variables, we suggest to simplify the contract by removing them.

```
    struct Pool {
        uint id;
        int256 contractID;
        uint256 blockStart;
        uint256 blockEnd;
        uint256 depositAmount;
        uint256 balanceAmount;
        uint256 havestBalance;
        uint256 currentExtraReward;
        uint256 packagePercent;
        address addr;
        uint256 extraRewards;
        uint256 prepareRewards;
        uint256 rewardPerBlock;
    }
```

Listing 3.2: The AcetAdaptor::Pool Struct

```
    function currentExtraRewardSum() external view returns (uint256) {
        uint256 sumExtra;
        for (uint i = 0; i < candidateCount; i++) {
              Pool storage people = peoples[i];
                if (people.addr == msg.sender) {
                    sumExtra = sumExtra.add(people.extraRewards);
                }
         }
        return sumExtra;
    }
```

Listing 3.3: `Pool::currentExtraRewardSum()`

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

Status This issue has been confirmed. Considering the contract has been deployed and is not upgradeable, the team decides to leave it as it is.

3.3 Incompatibility with Deflationary Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Pool`
- Category: Business Logics [6]
- CWE subcategory: CWE-841 [3]

Description

In the `Acet` protocol, the `Pool` contract is designed to take users' assets and deliver rewards depending on their deposit options. 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. For the above two operations, i.e., `deposit()` and `withdraw()`, the contract makes the use of `safeTransferFrom()` or `safeTransfer()` 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 _packagePercent,
        address _address,
        uint256 blockEstimated,
        uint256 _amount,
        IBEP20 syrupToken,
        int256 _contract
    ) private {
```

```
        uint256 fee = _amount.mul(StakeFees).div(100).div(100);
        syrupToken.safeTransferFrom(
            address(msg.sender),
            DeployerWalletAsdress,
            fee
        );
        ad.toPool(_amount.mul(_packagePercent).div(100).div(100), 1);
        uint256 toDev = _amount.mul(_packagePercent).div(100).div(100);
        ad.toDev(toDev.mul(AdditionalDeployerToken).div(100).div(100),
            DeployerWalletAsdress, 1);
        syrupToken.safeTransferFrom(
            address(msg.sender),
            address(this),
            _amount - fee
        );
        uint256 percent = _packagePercent;
        uint256 totalRewardsPrepare = _amount.mul(_packagePercent).div(100).div(100);
        uint256 rewardPerBlock =
        (_amount.mul(percent).div(100).div(100)).div(blockEstimated);
        addHolder(
            _contract,
            block.number,
            block.number.add(blockEstimated),
            _amount,
            _amount - fee,
            0,
            0,
            percent,
            address(msg.sender),
            0,
            totalRewardsPrepare,
            rewardPerBlock);

    }

    function _withdraw(
        IBEP20 fromRewardToken,
        IBEP20 syrupToken,
        uint256 packagePercent,
        uint256 _id
    ) private {

        if (peoples[_id].blockEnd > block.number) {
            ...
        }else {
            syrupToken.safeTransfer(address(msg.sender), peoples[_id].balanceAmount);
            peoples[_id].balanceAmount = peoples[_id].balanceAmount.sub(peoples[_id].
                balanceAmount);
            uint256 rewardReceive;
            if ( peoples[_id].blockEnd ==  peoples[_id].blockStart) {
                rewardReceive = 0;
            }else {
```

```
                rewardReceive = peoples[_id].prepareRewards.sub(peoples[_id].
                    havestBalance);
            }
            fromRewardToken.safeTransfer(DeployerWalletAsdress, rewardReceive.mul(
                HarvestFees).div(100).div(100));
            fromRewardToken.safeTransfer(address(msg.sender), rewardReceive.sub(
                rewardReceive.mul(HarvestFees).div(100).div(100)));
            peoples[_id].balanceAmount = 0;
            if (peoples[_id].blockStart == peoples[_id].blockEnd) {
                fromRewardToken.safeTransfer(adaptorAddress, peoples[_id].prepareRewards
                    .sub(peoples[_id].havestBalance));
                ad.toBurn(peoples[_id].prepareRewards.sub(peoples[_id].havestBalance));
            }
        }

    }
```

Listing 3.4: `Pool::deposit()/withdraw()`

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.

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 `Acet Farming` for support.

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 confirmed. Considering the contract has been deployed and is not upgradeable, the team decides to leave it as it is.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Pool
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the Acet protocol, there is a special administrative account, i.e., owner. This owner account plays a critical role in governing and regulating the system-wide operations (e.g., fees configuration, reward adjustment). Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

To elaborate, we show below the updateSpecificPool() function in the AcetAdaptor contract. This function allows the owner to change a key factor, poolStatusFlag, which greatly affects on if the pool could receive rewards or not.

```
    function updateSpecificPool(uint id, uint  status) public onlyOwner{
        uint checkID = 0;
        for (uint i = 0; i < candidateCount; i++) {
              Pool storage people = peoples[i];
                if (people.id == id) {
                    checkID = 1;
                }
        }
        require(checkID == 1, "ID doesn exist");
        require(status == 0  status == 1, "Error: The command was not found in the
            system");
        if (status == 1) {
            peoples[id].poolStatusFlag = 1;
        }else {
            peoples[id].poolStatusFlag = 0;
        }
    }
```

Listing 3.5: AcetAdaptor::updateSpecificPool()

Also, we notice in the emergencyUpdatePoolFee() routine, due to the lack of constraint of the changed PenaltyFees, the privileged account could even set the PenaltyFees to $100,000$ and then no funds could be withdrawn when the peoples[_id].blockEnd > block.number.

```
    function emergencyUpdatePoolFee(
        uint _StakeFees,
        uint _PenaltyFees,
        uint _AdditionalDeployerToken,
        uint _HarvestFees
```

```
        ) public onlyOwner {
         StakeFees =  _StakeFees;
         PenaltyFees =  _PenaltyFees;
         AdditionalDeployerToken  = _AdditionalDeployerToken;
         HarvestFees =  _HarvestFees;
    }
```

Listing 3.6: `Pool::emergencyUpdatePoolFee()`

Note that it could be worrisome if the privileged `owner` account is a plain EOA account. A revised multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

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. Considering the contract has been deployed and is not upgradeable, the team decides to leave it as it is and may consider to transfer the role to a community-governed DAO/multi-sig contract in the future.

4 Conclusion

In this audit, we have analyzed the design and implementation of the farming support in the `Acet Finance` protocol. The farming pool provides a decentralized liquidity platform for `LP` provider to deposit assets into the liquidity pool and earn rewards in return. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

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

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

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

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

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

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