SMART CONTRACT AUDIT REPORT

for

DeFiAI

PeckShield

Prepared By: Patrick Lou

PeckShield

April 19, 2022

Document Properties

Client	DeFiAI Finance
Title	Smart Contract Audit Report
Target	DeFiAI
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Patrick Lou, Jing Wang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 19, 2022	Xuxian Jiang	Final Release
1.0-rc	April 6, 2022	Xuxian Jiang	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

PeckShield

1 Introduction

Given the opportunity to review the design document and related source code of the DeFiAi 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 is well engineered, though it can be improved by further addressing the issues identified in this report. This document outlines our audit results.

1.1 About DeFiAi

DeFiAi is the latest protocol of DeFi2.0 with the goal of changing DeFi through the new CPA protocol, lossless and stable APY. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of the audited protocol

Item	Description
Name	DeFiAI Finance
Website	https://dfai.app/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 19, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note this contract has been deployed at 0xc919CD67a5Ff77e55B050f6A29Aa31Ec9869bf68.

- https://github.com/DEFIAI2021/defiai-v2.git (bfb7b19)

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

- https://github.com/DEFIAI2021/defiai-v2.git (0e2ba44)

1.2 About PeckShield

PeckShield Inc. [7] 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 [6]:

- 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) [5], 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 implementation of the `DeFiAi` 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	0	
Low	2	■
Informational	1	■
Total	3	

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 low-severity vulnerabilities and 1 informational recommendation.

Table 2.1: Key Audit Findings of DeFiAI Protocol

ID	Severity	Title	Category	Status
PVE-001	Informational	Meaningful Events For Important State Changes	Coding Practices	Resolved
PVE-002	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-003	Low	Trust on Admin Keys	Security Features	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.

PeckShield

3 | Detailed Results

3.1 Generation of Meaningful Events For Important State Changes

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `DeFiAIFarmV2`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

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

In the following, we use the `DeFiAIFarmV2` contract as an example. This contract has public functions that are used to configure important protocol-level parameters. While examining the events that reflect the `earlyExitFee` changes, we notice there is a lack of emitting important events that reflect important state changes. Specifically, when the `earlyExitFee` is being updated in `setEarlyExitFee()`, there is no respective event being emitted to reflect the update of `earlyExitFee` (line 186).

```
170     function set(
171         uint256 _pid,
172         uint256 _minFee
173     )
174         external
175         onlyGovernance
176         validatePid(_pid)
177     {
178         poolInfo[_pid].minFee = _minFee;
179     }
```

```
    function setEarlyExitFee(uint256 _earlyExitFee)
        external

        onlyGovernance
    {
        earlyExitFee = _earlyExitFee;
    }
```

Listing 3.1: `DeFiAIFarmV2::set()/setEarlyExitFee()`

Recommendation Properly emit respective events when protocol-wide parameters are updated.

Status This issue has been fixed in the following commit: `0f501e6`.

3.2 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `DeFiAIFarmV2`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

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

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
```

```
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.2: USDT Token **Contract**

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()` , 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 `transfer()` as well, i.e., `safeTransfer()`.

```
    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IERC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(
        IERC20 token,
        address spender,
        uint256 value
    ) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        require(
            (value == 0)  (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
            spender, value));
    }
```

Listing 3.3: `SafeERC20::safeApprove()`

In current implementation, if we examine the `DeFiAIFarmV2::deposit()` routine, it is designed to deposit the supported tokens into the pool for farming. To accommodate the specific idiosyncrasy, there is a need to use `safeApprove()`, instead of `safeIncreaseAllowance()` (line 119). Moreover, the `safeApprove()` call needs to be invoked twice: the first time resets the allowance to 0 and the second time sets the intended allowance amount.

```
104     function deposit(uint256 _pid, uint256 _wantAmt)
105         external
106
```

```
        validatePid(_pid)
        nonReentrant
    {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        if (_wantAmt > 0) {
            pool.want.safeTransferFrom(
                address(msg.sender),
                address(this),
                _wantAmt
            );

            pool.want.safeIncreaseAllowance(pool.strat, _wantAmt);
            uint256 sharesAdded = IDeFiAIMultiStrat(pool.strat).deposit(msg.sender,
                _wantAmt);
            user.shares = user.shares + sharesAdded;
            user.lastDepositedTime = block.timestamp;
        }
    }
```

Listing 3.4: `DeFiAIFarmV2::deposit()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/tranfer()/transferFrom()`.

Status This issue has been resolved as the team confirms the support of ERC20-compliant tokens.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `DeFiAIFarmV2`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [2]

Description

In the `DeFiAi` protocol, there are certain privileged accounts, i.e., `owner`. When examining the related contracts, we notice an inherent trust on these privileged accounts. For example, this `owner` account plays a critical role in governing and regulating the system-wide operations (e.g., configure various settings). It also has the privilege to control or govern the flow of assets within the protocol contracts . In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    function add(
        IERC20 _want,
        address _strat,
        uint256 _minFee
    )
        external
        onlyGovernance
    {
        require(_strat != address(0), "MinoFarm::add: Strat can not be zero address.");
        poolInfo.push(
            PoolInfo({
                want: _want,
                minFee: _minFee,
                strat: _strat
            })
        );
    }

    function set(
        uint256 _pid,
        uint256 _minFee
    )
        external
        onlyGovernance
        validatePid(_pid)
    {
        poolInfo[_pid].minFee = _minFee;
    }
```

Listing 3.5: Example Privileged Operations in DeFiAIFarmV2

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the owner 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 Make the list of extra privileges granted to owner explicit to the protocol users.

Status This issue has been resolved as the owner is now only used to initialize the protocol.

4 Conclusion

In this audit, we have analyzed the design and implementation of the DeFiAi protocol, which is the latest protocol of DeFi2.0 with the goal of changing DeFi through the new CPA protocol, lossless and stable APY. The current code base is well organized and those identified issues are promptly confirmed and addressed.

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.

PeckShield

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

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

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

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

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

PeckShield