SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: DEFIAI

Date: SEP 8, 2021

ITSOGOO received the application for a smart contract security audit of the DEFIAI on Sep 7, 2021. The following are the details and results of this smart contract security audit:

Project Name: DEFIAI

Contract address:0x7551F409a3ddFd63c3fBAc0fAa84A5a5e2939252

Link Address:
https://bscscan.com/token/0xddd840566a3a59db210fca1e1d4147c1be94a025

The audit items and results:
(Other unknown security vulnerabilities are not included in the audit responsibility scope)

Audit Result: Passed

Audit Date: Sep 8, 2021

Audit Team: ITSOGOO
https://www.itsogoo.org/

Table of Content

Introduction

This Audit Report mainly focuses on the overall security of DEFIAI Smart Contract. With this report, we have tried to ensure the reliability and correctness of their smart contract by complete and rigorous assessment of their system's architecture and the smart contract code base.

Auditing Approach and Methodologies applied

The ITSOGOO team has performed rigorous testing of the project starting with analyzing the code design patterns in which we reviewed the smart contract architecture to ensure it is structured and safe use of third-party smart contracts and libraries.

Our team then performed a formal line by line inspection of the Smart Contract to find any potential issue like race conditions, transaction-ordering dependence, timestamp dependence, and denial of service attacks.

In the Unit testing Phase, we coded/conducted custom unit tests written for each function in the contract to verify that each function works as expected.

In Automated Testing, we tested the Smart Contract with our in-house developed tools to identify vulnerabilities and security flaws.

The code was tested in collaboration of our multiple team members and this included ---

- Testing the functionality of the Smart Contract to determine proper logic has been followed throughout the whole process.
- Analyzing the complexity of the code in depth and detailed, manual review of the code, line-by-line.
- Deploying the code on testnet using multiple clients to run live tests.
- Analyzing failure preparations to check how the Smart Contract performs in case of any bugs and vulnerabilities.
- Checking whether all the libraries used in the code are on the latest version.
- Analyzing the security of the on-chain data.

Audit Details

Project Name: DEFIAI

Website:

https://dfai.finance/

Languages: Solidity (Smart contract)

Platforms and Tools: Remix IDE, Truffle, Truffle Team, Ganache, Solhint, VScode, Mythril, Contract Library

Audit Goals

The focus of the audit was to verify that the Smart Contract System is secure, resilient and working according to the specifications. The audit activities can be grouped in the following three categories:

Security

Identifying security related issues within each contract and the system of contract.

Sound Architecture

Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.

Code Correctness and Quality

A full review of the contract source code. The primary areas of focus include:

- Accuracy
- Readability
- Sections of code with high complexity
- Quantity and quality of test coverage

Issue Categories

Every issue in this report was assigned a severity level from the following:

High level severity issues

Issues on this level are critical to the smart contract's performance/functionality and should be fixed before moving to a live environment.

Medium level severity issues

Issues on this level could potentially bring problems and should eventually be fixed.

Low level severity issues

Issues on this level are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

Number of issues per severity

Critical	High	Medium	Low	Note
0	0	0	0	0

Issues Checking Status

№	Issue description.	Checking status
1	Compiler warnings.	Passed
2	Race conditions and Reentrancy. Cross-function race conditions.	Passed
3	Possible delays in data delivery.	Passed
4	Oracle calls.	Passed
5	Front running.	Passed
6	Timestamp dependence.	Passed
7	Integer Overflow and Underflow.	Passed
8	DoS with Revert.	Passed
9	DoS with block gas limit.	Passed
10	Methods execution permissions.	Passed
11	Economy model.	Passed
12	The impact of the exchange rate on the logic.	Passed
13	Private user data leaks.	Passed
14	Malicious Event log.	Passed
15	Scoping and Declarations.	Passed
16	Uninitialized storage pointers.	Passed
17	Arithmetic accuracy.	Passed
18	Design Logic.	Passed
19	Cross-function race conditions.	Passed
20	Safe Zeppelin module.	Passed
21	Fallback function security.	Passed

Manual Audit:

For this section the code was tested/read line by line by our developers. We also used Remix IDE's JavaScript VM and Kovan networks to test the contract functionality.

Critical Severity Issues

No critical severity issues found.

High Severity Issues

No high severity issues found.

Medium Severity Issues

No medium severity issues found.

Low Severity Issues

No low severity issues found.

Findings

1. Initializer input parameters

- ERC95-1
- Severity: Medium
- Likelihood: Low
- Impact: Medium

- Target: ERC95.sol
- Category: Low
- Finding Type: Dynamic
- Lines: 85-138

In the ERC95 contract, the constructor takes _percent as its input.

- This array input parameter should be checked for its item values less than 100 percent as it cannot be greater than 100 per;
- Line 102, the sum of all array items in _percent should be computed using SafeMath to avoid overflow that can result in _percent item values greater than 100 but the sum of all items percentTotal is still 100;
- Not using SafeMath can result in invalid percentage values (greater than 100), which can cause the contract function incorrectly. Due to not using SafeMath for summing percentage values, the transaction that initiates the contract can be succeeded but having invalid percentage values, which can open doors for exploitation later.

```
for (uint256 loop = 0; loop < _addresses.length; loop++) {
                // 0 % tokens cannnot be permitted
                require(_percent[loop] > 0 ERC95 : All wrapped tokens have to have at least 1%
of total'1);

                // we check the decimals of current token
                // decimals is not part of erc20 standard, and is safer to provide in the caller
                // tokenDecimals[loop] = IERC20(_addresses[loop]).decimals();
                decimalsMax   tokenDecimals[loop] > decimalsMax ? tokenDecimals[loop]
decimalsMax; // pick max

                percentTotal += _percent[loop]; // further for checking everything adds up
                //_numTokensWrapped++; // we might just assign this
                numTokensWrapped++;
                console. log ("loop one loop count:11 r loop);

            }
```

Action Recommended:

- Check _percent array item values less than or equal to 100;
- Use SafeMath for computing the sum of .percent;

```
for {uint256 loop = 0; loop < _addresses.length; loop++) (
                    // 0 % tokens cannnot be permitted
                    require(_percent[loop] > 0 ERC95 : All wrapped tokens have to have at least
1% of total"[1]);
                    require(_percent[loop] <= 100 ERC95 : All wrapped tokens have to have at most
100% of total");

                    // we check the decimals of current token
                    // decimals is not part of erc20 standard, and is safer to provide in the caller
                    // tokenDecimals[loop] = IERC20(_addresses[loop]).decimals();
                    decimalsMax ' tokenDecimals[loop] > decimalsMax ? tokenDecimals[loop]
decimalsMax; // pick max
                    percentTotal = percentTotal.add(_percent[loop]); // further for checking everything
adds up
                    //_numTokensWrapped++; //we might just assign this
                    numTokensWrapped++;
                    console. log ("loop one loop count:[11] r loop);
```

2. Redundant local variable numTokensWrapped

- ERC95-2
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: ERC95.sol
- Category: Informational
- Finding Type: Dynamic
- Lines: 92,104,109,110

In the function ___ERC95_init of the contract ERC95, the local variable numTokensWrapped is redundant. This is because if the contract initializer succeeds, the value of numTokensWrapped should be always _addresses , length. This leads to the redundancy of the statements at lines 104 and 109.

```
uint8 numTokensWrapped = 0;
            r (uint256 loop = 0; loop < _addresses.length; loop++) { // 0 % tokens cannnot be
              permitted
```

```
require(^percent[loop] > 0 ,"ERC95 : All wrapped tokens have to have at least 1% of
total");

// we check the decimals of current token
// decimals is not part of erc20 standard, and is safer to provide in the caller
// tokenDecimals[loop] = IERC20(_addresses[loop]).decimals();
decimalsMax    tokenDecimals[loop] > decimalsMax ? tokenDecimals[loop]
               decimalsMax;
// pick max

percentTotal += _j>ercent[loop]; // further for checking everything adds up
//_numTokensWrapped++; // we might just assign this
numTokensWrapped +;
console.log(Hloop one loop count:n, loop);


require(percentTotal == 100, "ERC95 : Percent of all wrapped tokens should equal
100"); require(numTokensWrapped = _addresses.length, "ERC95 : Length mismatch sanity check
fail"); // Is this sanity check needed? // No, but let's leave it anyway in case it becomes needed later
_numTokensWrapped        numTokensWrapped;
```

Action Recommended: For simplification, removing the local variable numTokensWrapped will be fine. This can be done by deleting lines *92,* 104, and 109, while changing line 110 to:

```
_numTokensWrapped = _addresses.length;
```

3. Missing checking tokenDecimals's length

- ERC95-3
- Severity: low
- Likelihood: Low
- Impact: Low

- Target: ERC95.sol
- Category: Low
- Finding Type: Dynamic
- Lines: 89

In the ERC95 contract's initializer, the latter should verify that the number of token decimals in the input array tokenDecimals is equal to the length of array .addresses. This can save gas cost in case the length Of tokenDecimals is less than that Of .addresses .

Action Recommended: Add a require statement to check the length of tokenDecimals must be equal to the length of .addresses.

4. Deposit and Unwrap can fail with tokens having fees on transfer

- ERC95-4
- Severity: High
- Likelihood: Medium
- Impact: Medium
- Target: ERC95.sol
- Category: Medium
- Finding Type: Dynamic
- Lines: 175-182, 204-212, 236-262

In the ERC95 contract, any user can trigger a wrap action that can create a corresponding amount of the ERC95 token with respective to _amountWrapperPerUnit. If one of the supported tokens of the ERC95 token contract has fees on transfer, the function sendUnderlyingTokens can fail as follows：

- When depositing _amt of ERC95 token, function _depositUnderlying computes the corresponding amounts of wrapped tokens, for example wrapping m amount of token x.
- If x has fees or burn rate on transfer, the ERC95 token contract will receive an amount less than m.
- When the user decides to unwrap, in function sendUnderlyingTokens, the ERC95 token contract must send m amount of x to the user address. This transaction will fail because the ERC95 token contract has less than m token of x.

Due to this mismatch between the expected deposit and the actual deposit amount, the function _updateReserves will also fail due to an underflow at line 245. This is because the total actual deposit will be less than the reserve, which is the sum of all expected deposits.

```
function sendUnderlyingTokens(address to, uint256 amt) internal {
        for (uint256 loop = 0; loop < _numTokensWrapped; loop++) {
            WrappedToken storage currentToken _wrappedTokens[loop]; uint256 amtToSend
                                amt.mul(currentToken._amountWrapperPerUnit);
            safeTransfer(currentToken._address, to, amtToSend); currentToken._reserve
                                    currentToken._reserve.sub(amtToSend);


// Loops over all tokens in the wrap and deposits them with allowance
    function depositunderlying(uint25& amt) internal (
        for {uint256 loop = 0; loop < _numTokensWrapped; loop++) {
            WrappedToken memory currentToken    _wrappedTokens[loop];
            // req successful transfer
            uint256 amtToSendamt.mul(currentToken._amountWrapperPerUnit);
            safeTransferFrom(currentToken._address, msg.sender, address(this) , amtToSend);
            // Transfer went OK this means we can add this balance we just took.
            _wrappedTokens[loop]—reserve          currentToken._reserve.add(amtToSend);
    }
```

```
function updateReserves() internal returns (uint256 qtyOfNewTokens) (
            // Loop through all tokens wrapped, and find the maximum quantity of wrapped tokens
that can be created, given the balance delta for this block
            console.log("_numTokensWrapped: ",  _numTokensWrapped);
            for (uint256 loop = 0; loop < _numTokensWrapped; loop++) {
                WrappedToken memory currentToken _wrappedTokens[loop];
                uint256 currentTokenBal
                                        IERC20(currentToken._address).balanceOf(address(this))
                ;
                console.log("currentTokenBal inside loop:     currentTokenBal,
                currentToken._address);
                console.log("currentToken._amountWrapperPerUnit: ",
currentToken._amountWrapperPerUnit);
                // TODO: update to not use percentages
                uint256 amtCurrent
currentTokenBal.sub(currentToken._reserve).div(currentToken._amountWrapperPerUnit); // math
                check pls console.log("amtCurrent: ", amtCurrent);
                qtyOfNewTokens qtyOfNewTokens > amtCurrent ? amtCurrent qtyOfNewTokens;
// logic check // pick lowest amount so dust attack doesn't work
                                                        // can*t skim in txs or they have non-
deterministic gas price
                console.log("qtyOfNewTokens: ", qtyOfNewTokens);
                if(loop == 0) {
                    qtyOfNewTokens amtCurrent;
                }
            }
            console. log ("Lowest common denominator for token mint:  qtyOfNewTokens);
            // second loop makes reserve numbers match from computed amount
            for (uint256 loop2 0; loop2 < _numTokensWrapped; loop2++) (
                WrappedToken memory currentToken    _wrappedTokens[loop2];
                uint256 amtDelta
                                    qtyOfNewTokens.mul(currentToken._amountWrapperPerUnit);/
                / math
check pls
                _wrappedTokens[loop2],_reserve currentToken._reserve.add(amtDelta); / / math
                check pls
```

Action Recommended:

- Because there is no standard for fees or burn on transfer tokens, it's hard, even impossible to check whether a token has fees or burns on transfer. This is because a token might support fees using a threshold-based, percentage-based, or flat fee mechanism. Thus, it is very hard to support those types of tokens. Therefore, before deploying an ERC95 token, the deployer should verify that all supported tokens in the initializer should not have any fees or burns on transfer.
- A possible solution the team can consider is to recompute _amountwrapperPerUnit every time a new deposit is made. In a typical case where all supported tokens are no fees or burns on transfer, _amountwrapperPerUnit will be constant. In cases of tokens with fees or burns on transfer, _amountWrapperPerUnit will be adjusted with the fees or burns on transfers. The following is a suggested code portion for function _depositUnderlying.
- Because _amountWrapperPerUnit is adjusted every deposit (it's worth noting that the adjustment will be small because fees or burns on transfers are usually small), the user of _reserve is unnecessary. This method also improves on the aspect of excessive token quantity in function skim. The latter is only not useful for normal users as it will be called by any programmed bots written by developers to claim any excessive token quantity. By using the function syncAmountPerUnit, if there is any excessive token quantity, _amountwrapperPerUnit will be increased after any deposit or unwrap. The increase of _amountWrapperPerUnit will benefit all users that deposit tokens to the contract.

```
function syncAmountPerUnit(uint256 tokenindex, uint256 amt) internal {
    WrappedToken storage currentToken          _wrappedTokens[tokenIndex];
    uint256 currentTokenBal  IERC20(currentToken._address).balanceOf(address(this));
    uint256 supplyAfterMint  _totalSupply.add(amt);
    currentToken._amountWrapperPerUnit        currentTokenBal.div(supplyAfterMint);
}

// Loops over all tokens in the wrap and deposits them with allowance function
_depositUnderlying(uint256 amt) internal {
    for (uint256 loop = 0; loop < _numTokensWrapped; loop++) {
        WrappedToken memory currentToken = _wrappedTokens[loop];
        // req successful transfer
        uint256 amtToSend amt.mul(currentToken._amountWrapperPerUnit);
        safeTransferFrom(currentToken._address, msg.sender, address(this) , amtToSend);
        syncAmountPerUnit(loop, amt);
    }

function sendUnderlyingTokens(address to, uint256 amt) internal { for (uint256 loop = 0;
    loop < _numTokensWrapped; loop++) (
        syncAmountPerUnit(loop, 0); //sync _amountWrapperPerUnit before unwrapping
        WrappedToken storage currentToken _wrappedTokens[loop];
```

```
            uint256 amtToSendamt.mul(currentToken._amountWrapperPerUnit);
            safeTransfer(currentToken._address, to, amtToSend);
        )
    }
```

5. Inherited Vulnerable Voting Code

- CORE-6
- Severity: Medium
- Likelihood: Low
- Impact: Low

- Target: NBUNIERC20.sol
- Category: Code Clarity
- Finding Type: Dynamic
- Lines: *

This issue is a duplicate of CORE-5 except affecting the NBUNIERC20.sol contract
Action Recommended: As the ERC-20 contract has already been deployed there is very little immediate action possible, however in the event of any redeployments or new token deployments, the unused and vulnerable voting code should be removed from the token contract before any new deployments.

6. Code Structure

- CORE-7
- Severity: Notice
- Likelihood: Notice
- Impact: Notice

- Target: NBUNIERC20.sol
- Category: Code Clarity
- Finding Type: Dynamic
- Lines: *

Action Recommended: Ensure that you try to order your storage variables and struct members such that they can be packed tightly. For example, declaring your storage variables in the order of uint128, uint128, uint256 instead of uint128, uint256, uint128, as the former will only take up two slots of storage whereas the latter will take up three. This is a good way to improve codebase health, gas cost and clarity.

7. Token Uniswap Pair Location

- CORE-8
- Severity: Medium
- Likelihood: Low
- Impact: Low

- Target: FeeApprover.sol & NBUNIERC20.sol
- Category: Code Clarity
- Finding Type: Dynamic
- Lines: 382

The tokenUniswapPair state variable is obtained through uniswap but it appears independently in two different contracts (FeeApprover.sol and NBUNIERC20.sol). This function should check whether tokenUniswapPair is the same in FeeApprover.sol and NBUNIERC20.sol

```
    function initialize(
        address _COREAddress,
        address _WETHAddress,
        address _uniswapFactory
    ) public initializer {
        OwnableUpgradeSafe.__Ownable_init();
        coreTokenAddress = _COREAddress;
        WETHAddress = _WETHAddress;
        tokenUniswapPair =
IUniswapV2Factory(_uniswapFactory).getPair(WETHAddress,coreTokenAddress);
        feePercentX100 = 10;
        paused = true; // We start paused until sync post LGE happens.
        _editNoFeeList(0xC5cacb708425961594B63eC171f4df27a9c0d8c9, true);
// corevault proxy
        _editNoFeeList(tokenUniswapPair, true);
        sync();
        minFinney = 5000;
    }
```

Action Recommended: Add sanity check to ensure that the trading pair is the same across both contracts. This does not impact current contracts much as the pairs have been set correctly here, but to avoid unnecessary issues in the future, a sanity check would greatly improve code reusability.

8. Redundant Check

- CORE-9
- Severity: Notice
- Likelihood: Notice
- Impact: Notice
- Target: NBUNIERC20.sol
- Category: Code Clarity
- Finding Type: Dynamic
- Lines: 448

There is a redundant check for feeDistributor in the contract, it is not an issue, however the overall deployment and utilization cost will be lowered if this redundant check is removed.

```
if(transferToFeeDistributorAmount > 0 && feeDistributor != address(0)){
            _balances[feeDistributor] =
_balances[feeDistributor].add(transferToFeeDistributorAmount);
            emit Transfer(sender, feeDistributor,
transferToFeeDistributorAmount);
```

```
            if(feeDistributor  != address(0)){

ICoreVault(feeDistributor).addPendingRewards(transferToFeeDistributorAmount
);
            }
        }
    }
```

Action Recommended: Remove redundant check when possible.

10. Readibility

- CORE-10
- Severity: Notice
- Likelihood: Notice
- Impact: Notice
- Target: NBUNIERC20.sol
- Category: Readibility
- Finding Type: Dynamic
- Lines: 221

```
mapping (address => uint)  public ethContributed;
```

Action Recommended: Mapping should be moved to a more appropriate location in the contract.

CoreVault.sol

11. Gas Optimization

- CORE-11
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: NBUNIERC20.sol
- Category: Gas
- Finding Type: Dynamic
- Lines: 133

The add function could potentially run out of gas as the for-loop can cost a large amount of gas in the event there are a large number of pools. While it is unlikely that CORE will have that number of pools in the near future, it is better to have future-proofed than to not have.

```
for (uint256 pid = 0; pid < length; ++pid) {
            require(poolInfo[pid].token != _token,"Error pool already
added");
        }
```

Action Recommended: Add a mapping or other solution that searches for the pool, in order to minimize the gas cost for the for-loop.

Automated Audit

Remix Compiler Warnings

It throws warnings by Solidity's compiler. If it encounters any errors the contract cannot be compiledand deployed. No issues found.

Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice as at the date of this report, in relation to cyber security vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the below disclaimer below – please make sure to read it in full.

DISCLAIMER: By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis, and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and ITSOGOO and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers and other representatives) (ITSOGOO) owe no duty of care towards you or any other person, nor does ITSOGOO make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties or other terms of any kind except as set out in this disclaimer, and ITSOGOO hereby excludes all representations, warranties, conditions and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, ITSOGOO hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against ITSOGOO, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of use of this report, and any reliance on this report.

The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed.

Summary

Smart contracts do not contain any high severity issues.

Note:
Please check the disclaimer above and note, the audit makes no statements or warranties on business model, investment attractiveness or code sustainability. The report is provided for the only contract mentioned in the report.