SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: DEFIAI
Date: SEP 14 , 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

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

Audit Result: Passed
Audit Date: Sep 14, 2021
Audit Team: ITSOGOO

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.

Detailed Results

Potential Front-Running For Migration Blocking

- ID: PVE-001
- Severity: High
- Likelihood: Medium
- Impact: High
- Category: Time and State [11]
- CWE subcategory: CWE-663 [3]

```
// Migrate lp token to another lp contract. Can be called by anyone. We trust that
migrator contract is good.
function migrate ( uint256 _pid ) public { require ( address ( migrator ) != address (0) , "migrate: no
migrator") ; PoolInfo storage pool = poolInfo [ _pid ] ;
IBEP20 lpToken = pool . lpToken ; uint256 bal = lpToken . balanceOf ( address ( this ) ) ; lpToken .
safeApprove ( address ( migrator ) , bal ) ; IBEP20 newLpToken = migrator . migrate ( lpToken ) ; require
( bal == newLpToken . balanceOf ( address ( this ) ) , "migrate: bad") ; pool . lpToken = newLpToken ;
}
```

Listing 3.1: MasterChef.sol

The actual bulk work of migration is performed by the `Migrator` contract in a function also named `migrate()` (we show the related code snippet below).

This assumption essentially reflects the code logic in lines 126*128. In other words, if an actor is able to front-run it to become the first one in successfully minting the new LP tokens, the actor will successfully block this migration (of this specific trading pair or the pool in `MasterChef`).

```
function mint ( address to ) external lock returns ( uint liquidity ) {
( uint112 _reserve0 , uint112 _reserve1 , ) = getReserves () ; // gas savings
uint balance0 = IBEP20 ( token0 ) . balanceOf ( address ( this ) ) ;
uint balance1 = IBEPC20 ( token1 ) . balanceOf ( address ( this ) ) ;
uint amount0 = balance0 . sub ( _reserve0 ) ;
uint amount1 = balance1 . sub ( _reserve1 ) ;

    bool feeOn = _mintFee ( _reserve0 , _reserve1 ) ;
    uint _totalSupply = totalSupply ; // gas savings, must be defined here
    since totalSupply can update in _mintFee
    i f ( _totalSupply == 0) {
    address migrator = IFactory ( factory ) . migrator () ;
    i f (msg. sender == migrator ) {
    l i q u i d i t y = IMigrator ( migrator ) . d e s i r e d L i q u i d i t y ();
    require ( l i q u i d i t y > 0 && l i q u i d i t y != uint256 (*1) , "Bad desired liquidity") ;
    } else {
    require ( migrator == address (0) , "Must not have migrator");
    l i q u i d i t y = Math . sqrt ( amount0 . mul ( amount1 ) ) . sub (MINIMUM_LIQUIDITY) ;
132 }
    _mint( address (0) , MINIMUM_LIQUIDITY) ; // permanently lock the
   first MINIMUM_LIQUIDITY tokens
    } else {
    l i q u i d i t y = Math . min ( amount0 . mul ( _totalSupply ) / _reserve0 ,
     amount1 . mul (
       _totalSupply ) / _reserve1 ) ;
136 }
    require ( l i q u i d i t y > 0 , ': INSUFFICIENT_LIQUIDITY_MINTED');
    _mint( to , l i q u i d i t y ) ;

_update ( balance0 , balance1 , _reserve0 , _reserve1 ) ;
i f ( feeOn ) kLast = uint ( reserve0 ) . mul ( reserve1 ) ; // reserve0 and reserve1 are up
    -to-date
emit Mint (msg. sender , amount0 , amount1 ) ;
143 }
```

Listing 3.3: Pair.sol

Recall the above migration check that essentially states the new LP token amount should equal to the old LP token amount. If the migration transaction is not the first to mint new LP tokens, the first transaction that successfully mints the new LP tokens will lead to `_totalSupply != 0`. In other words, the migration transaction will be forced to take the execution path in lines 135, not the intended lines 126*128. As a result, the minted amount is unlikely to be the same as the old pool token amount before migration, hence failing the migration check!

To ensure a smooth migration process, we need to guarantee the first minting of new LP tokens is launched by the migration transaction. To achieve that, we need to prevent any unintended minting (of new LP tokens) between the first step `deploy` and the third step `configure MasterChef`. A natural approach is to complete the initial three steps within the same transaction, best facilitated by a contract-coordinated deployment.

Recommendation Deploy these contracts in a coherent fashion and avoid the above-mentioned front-running to guarantee a smooth migration.

Status This issue has been confirmed and largely addressed by streamlining the entire deployment script (without the need of actually revising the smart contract implementation). This is indeed the approach the team plans to take and exercise with extra caution when deploying these contracts (by avoiding unnecessary exposure of vulnerable time window for front-running).

• Avoidance of Unnecessary (Small) Loss During Migration

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low

- Category: Business Logics [10]
- CWE subcategory: CWE-841 [8]

Description

We have discussed the four distinct migration steps in Section 3.1 and highlighted the need of being the first one for the migrator to mint the new liquidity pool (LP) tokens. In this section, we further elaborate another issue in current migration logic that could unnecessarily lead to a (small) loss of assets.

• Duplicate Pool Detection and Prevention

Description

DEFEAI provides incentive mechanisms that reward the staking BUSD tokens. The rewards are carried out by designating a number of staking pools. Each pool has its `allocPoint*100%/totalAllocPoint` share of scheduled rewards and the rewards these stakers in a pool will receive are proportional to the amount of LP tokens they have staked in the pool versus the total amount of LP tokens staked in the pool.

As of this writing, there are 2 pools that share the rewards tokens. To accommodate these new pools, DEFIAI has the necessary mechanism in place that allows for dynamic additions of new staking pools that can participate in being incentivized as well.

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

```
function add ( uint256 _allocPoint , IBEP20 _lpToken , bool _withUpdate ) public onlyOwner {
    i f ( _withUpdate ) { massUpdatePools () ;
    }
    uint256 lastRewardBlock = block . number > startBlock ? block . number : startBlock ;
    t o t a l A l l o c P o i n t = t o t a l A l l o c P o i n t . add ( _allocPoint ) ;
    poolInfo .
    push ( PoolInfo ({ lpToken : _lpToken ,
        a l l o c P o i n t : _allocPoint ,
        lastRewardBlock : lastRewardBlock , acciPerShare : 0
    ` ` `  `
```

Listing 3.6: MasterChef.sol

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

```
function checkPoolDuplicate ( IBEP20 _lpToken ) public { uint256 length = poolInfo . length ; for ( uint256 pid = 0; pid
        < length ; ++pid ) { require ( poolInfo [ _pid ] . lpToken != _lpToken , "add: existing pool?") ;
        }
}

function add ( uint256 _allocPoint , IBEP20 _lpToken , bool _withUpdate ) public onlyOwner
        { i f ( _withUpdate ) { massUpdatePools () ;
        } checkPoolDuplicate ( _lpToken ) ; uint256 lastRewardBlock = block . number > startBlock ? block . number :
        startBlock ; t o t a l A l l o c P o i n t = t o t a l A l l o c P o i n t . add ( _allocPoint ) ; poolInfo .
        push ( PoolInfo ({ lpToken : _lpToken ,
                a l l o c P o i n t :       _allocPoint ,
                                lastRewardBlock : lastRewardBlock , accPerShare :
                                        0 }) ) ;
                                        }
```

Listing 3.7: MasterChef.sol (revised)

We point out that if a new pool with a duplicate LP token can be added, it will likely cause a havoc in the distribution of rewards to the pools and the stakers. Worse, it will also bring great troubles for the planned migration!

Status We have discussed this issue with the team and the team is aware of it. Since the `MasterChef` contract is already live (with a huge amount of assets), the team prefers not modifying the code for the duplicate prevention, but instead takes necessary off-chain steps and exercises with extra caution to block duplicates when adding a new pool.

3.4 Recommended Explicit Pool Validity Checks

- ID: PVE-004 • Target: MasterChef
- Severity: Medium • Category: Security Features [9]
- Likelihood: Low • CWE subcategory: CWE-287 [2]
- Impact: High

Description

Central contract – MasterChef that has been tasked with not only the migration (Section 3.1), but also the pool management, staking/unstaking support, as well as the reward distribution to various pools and stakers.

In the following, we show the key pool data structure. Note all added pools are maintained in an array poolInfo.

```
// Info of each pool.
struct PoolInfo {
    IBEP20 lpToken;            // Address of LP token contract.
    uint256 allocPoint;    // How many allocation points assigned to this pool. to
        distribute per block.
    uint256 lastRewardBlock; // Last block number that distribution occurs.
        uint256 accPerShare; // Accumulated per share, times 1e12. See below
        .
}
...
// Info of each pool.
PoolInfo[] public poolInfo;
```

Listing 3.8: MasterChef.sol

When there is a need to add a new pool, set a new allocPoint for an existing pool, stake and unstake, query pending rewards, or migrate the pool assets, there is a constant need to perform sanity checks on the pool validity. The current implementation simply relies on the implicit, compiler-generated bound-checks of arrays to ensure the pool index stays within the array range [0, poolInfo.length-1]. However, considering the importance of validating given pools and their numerous occasions, a better alternative is to make explicit the sanity checks by introducing a new modifier, say validatePool. This new modifier essentially ensures the given _pool_id or _pid indeed points to a valid, live pool, and additionally give semantically meaningful information when it is not!

```
// Deposit LP tokens to MasterChef for allocation.
function deposit ( uint256 _pid , uint256 _amount) public
    PoolInfo storage pool = poolInfo
    UserInfo storage user = u s e r I n f o [ _pid ] [ msg. sender ] ; updatePool ( _pid ) ; i f ( user . amount > 0) { uint256 pending =
    user . amount . mul ( pool . accPerShare ) . div (1 e12 ) . sub ( user .
        rewardDebt ) ;
        safeTransfer (msg. sender , pending ) ;
    } pool . lpToken . safeTransferFrom ( address (msg. sender ) , address ( this ) , _amount) ; user .
    amount = user . amount . add (_amount) ; user . rewardDebt = user . amount . mul ( pool . accPerShare ) . div (1
    e12 ) ;
    emit Deposit (msg. sender , _pid , _amount) ;
}
```

Listing 3.9: MasterChef.sol

We highlight that there are a number of functions that can be benefited from the new pool-validating modifier, including `set ()`, `migrate ()`, `deposit ()`, `withdraw ()`, `emergencyWithdraw ()`, `pending ()` and `updatePool ()`.

Recommendation Apply necessary sanity checks to ensure the given `_pid` is legitimate. Accordingly, a new modifier `validatePool` can be developed and appended to each function in the above list.

```
modifier validatePool ( uint256 _pid ) { require ( _pid < poolInfo . length , "chef: pool
    exists?") ; _;
}

function deposit ( uint256 _pid , uint256 _amount) public validatePool ( _pid ) {
    PoolInfo storage pool = poolInfo
    UserInfo storage user = u s e r I n f o [ _pid ] [ msg. sender ] ; updatePool
    ( _pid ) ;

        uint256 pending = user . amount . mul ( pool . accPerShare ) . div (1 e12 ) . sub ( user .
        rewardDebt ) ;
        safeTransfer (msg. sender , pending ) ;
    } pool . lpToken . safeTransferFrom ( address (msg. sender ) , address ( this ) , _amount) ; user . amount = user .
    amount . add (_amount) ; user . rewardDebt = user . amount . mul ( pool . accPerShare ) . div (1 e12 ) ; emit Deposit
    (msg. sender , _pid , _amount) ;
}
```

Listing 3.10: MasterChef.sol

Status We have discussed this issue with the team. For the same reason as outlined in Section 3.3, because the `MasterChef` contract is already live (with a huge amount of assets), any change needs to be deemed absolutely necessary. In this particular case, the team prefers not modifying the code as the compiler-generated bounds-checking is already in place.

• Incompatibility With Deflationary Tokens

- ID: PVE-005 • Target: `MasterChef`
- Severity: Low • Category: Business Logics [10]
- Likelihood: Low • CWE subcategory: CWE-708 [5]
- Impact: Medium

Description

the `MasterChef` contract operates as the main entry for interaction with staking users. The staking users `deposit` LP tokens into the pool and in return get proportionate share of the pool's rewards. Later on, the staking users can `withdraw` their own assets from the pool. With assets in the pool, users can earn whatever incentive mechanisms proposed or adopted via governance.

Naturally, the above two functions, i.e., `deposit()` and `withdraw()`, are involved in transferring users' assets into (or out of) the protocol. Using the `deposit()` function as an example, it needs to transfer deposited assets from the user account to the pool (line 210). When transferring standard BEP20 tokens, these asset-transferring routines work as expected: namely the account's internal asset balances are always consistent with actual token balances maintained in individual BEP20 token contracts (lines 211*212).

```
201 // Deposit LP tokens to MasterChef for allocation.
202 function deposit ( uint256 _pid , uint256 _amount) public
203 {PoolInfo storage pool = poolInfo [ _pid ] ;
204 UserInfo storage user = u s e r I n f o [ _pid ] [ msg. sender ] ;
205 updatePool ( _pid ) ; i f ( user . amount > 0) { uint256 pending = user . amount . mul ( pool . accPerShare ) . div (1 e12 ) . sub
206 ( user .
207 rewardDebt ) ;
    safeTransfer (msg. sender , pending ) ;
208 } pool . lpToken . safeTransferFrom ( address (msg. sender ) , address ( this ) , _amount) ; user . amount = user .
209 amount . add (_amount) ; user . rewardDebt = user . amount . mul ( pool . accPerShare ) . div (1 e12 ) ; emit Deposit
210 (msg. sender , _pid , _amount) ;
211
212
213
214 }
```

Listing 3.11: MasterChef.sol

However, in the cases of deflationary tokens, as shown in the above code snippets, the input amount may not be equal to the received amount due to the charged (and burned) transaction fee. As a result, this may not meet the assumption behind these low-level 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 BEP20 token contracts in the cases of deflationary tokens. Apparently, these balance inconsistencies are damaging to accurate portfolio management of `MasterChef` and affects protocol-wide operation and maintenance. One mitigation is to query the asset change right before and after the asset-transferring routines. In other words, instead of automatically assuming the amount parameter in `transfer()` or `transferFrom ()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `transfer()`/`transferFrom()` is expected and aligned well with the intended operation. Though these additional checks cost additional gas usage, we feel that 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 BEP20 tokens that are permitted into `MasterChef` pools. With the single entry of adding a new pool (via `add()`), `MasterChef` is indeed in the position to effectively regulate the set of assets allowed into the protocol.

Fortunately, the LP tokens are not deflationary tokens and there is no need to take any action. However, it is a potential risk if the current code base is used elsewhere or the need to add other tokens arises (e.g., in listing new DEX pairs). Also, the current code implementation, including the path-supported `swap()` and thus similar `swap()`, is indeed not compatible with deflationary tokens.

Recommendation Regulate the set of LP tokens supported in `DEFIAI` and, if there is a need to support deflationary tokens, add necessary mitigation mechanisms to keep track of accurate balances.

Status This issue has been confirmed. As there is a central place to regulate the assets that can be introduction in the pool management, the team decides no change for the time being.

3.6 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MasterChef`
- Category: Time and State [11]
- CWE subcategory: CWE-663 [3]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Binance Smart Chain history, including the `DAO` [22] exploit, and the recent `Lendf.Me` hack [20]. We notice there are several occasions the `checks-effects-interactions` principle is violated. Using the `MasterChef` as an example, the `emergencyWithdraw()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 234) starts before effecting the update on internal states (lines 236*237), hence violating the principle. In this particular case, if the external contract has some hidden logic that may be capable of launching `re-entrancy` via the very same `emergencyWithdraw()` function.

```
230    // Withdraw without caring about rewards. EMERGENCY ONLY.
231    function emergencyWithdraw ( uint256 _pid ) public
232        PoolInfo storage pool = poolInfo
233        UserInfo storage user = u s e r I n f o [ _pid ] [ msg. sender ] ; pool . lpToken . safeTransfer
234        ( address (msg. sender ) , user . amount) ;
235        emit EmergencyWithdraw (msg. sender , _pid , user . amount ) ;
236        user . amount = 0;
237        user . rewardDebt = 0;
238    }
```

Listing 3.12: MasterChef.sol

Another similar violation can be found in the `deposit()` and `withdraw()` routines within the same contract.

```
// Deposit LP tokens to MasterChef for allocation.
function deposit ( uint256 _pid , uint256 _amount) public { PoolInfo storage pool =
poolInfo [ _pid ] ;
UserInfo storage user = u s e r I n f o [ _pid ] [ msg. sender ] ; updatePool ( _pid ) ; i f ( user . amount > 0) { uint256 pending =
user . amount . mul ( pool . accPerShare ) . div (1 e12 ) . sub ( user .
rewardDebt ) ;
safeTransfer (msg. sender , pending ) ;
} pool . lpToken . safeTransferFrom ( address (msg. sender ) , address ( this ) , _amount) ; user .
amount = user . amount . add (_amount) ; user . rewardDebt = user . amount . mul ( pool . acciPerShare ) . div (1
e12 ) ;
emit Deposit (msg. sender , _pid , _amount) ;
}

// Withdraw LP tokens from MasterChef.
function withdraw ( uint256 _pid , uint256 _amount) public
{PoolInfo storage pool = poolInfo [ _pid ] ;
UserInfo storage user = u s e r I n f o [ _pid ] [ msg. sender ] ; require ( user . amount >= _amount , "withdraw: not
good") ; updatePool ( _pid ) ; uint256 pending = user . amount . mul ( pool . accPerShare ) . div (1 e12 ) . sub ( user .
rewardDebt ) ;
safeTransfer (msg. sender , pending ) ; user . amount = user . amount . sub (_amount) ; user .
rewardDebt = user . amount . mul ( pool . accPerShare ) . div (1 e12 ) ; pool . lpToken . safeTransfer
( address (msg. sender ) , _amount) ;
emit Withdraw (msg. sender , _pid , _amount) ;
}
```

Listing 3.13: MasterChef.sol

In the meantime, we should mention that the LP tokens implement rather standard BEP20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`.

Recommendation Apply necessary reentrancy prevention by following the `checks-effectsinteractions` best practice. The above three functions can be revised as follows:

```
// Withdraw without caring about rewards. EMERGENCY ONLY.
function emergencyWithdraw ( uint256 _pid ) public
    PoolInfo storage pool = poolInfo
   UserInfo storage user = u s e r I n f o [ _pid ] [ msg. sender ] ; uint256 _amount=user .
   amount
   user . amount = 0; user . rewardDebt = 0; pool . lpToken . safeTransfer ( address (msg.
   sender ) , _amount) ;
   emit EmergencyWithdraw (msg. sender , _pid , _amount) ;
}

// Deposit LP tokens to MasterChef for allocation.
function deposit ( uint256 _pid , uint256 _amount) public
    PoolInfo storage pool = poolInfo [ _pid ] ;
    UserInfo storage user = u s e r I n f o [ _pid ] [ msg.


   updatePool ( _pid ) ; uint256 pending = user . amount . mul ( pool . accPerShare ) . div (1 e12 ) . sub ( user .
       rewardDebt ) ;

   user . amount = user . amount . add (_amount) ;
   user . rewardDebt = user . amount . mul ( pool . accPerShare ) . div (1 e12 ) ;


   safeTransfer (msg. sender , pending ) ; pool . lpToken . safeTransferFrom ( address (msg. sender ) ,
      address ( this ) , _amount) ;
   emit Deposit (msg. sender , _pid , _amount) ;
}


// Withdraw LP tokens from MasterChef.
function withdraw ( uint256 _pid , uint256 _amount) public
    PoolInfo storage pool = poolInfo
   UserInfo storage user = u s e r I n f o [ _pid ] [ msg. sender ] ; require ( user . amount >= _amount , "withdraw: not
   good") ; updatePool ( _pid ) ; uint256 pending = user . amount . mul ( pool . accPerShare ) . div (1 e12 ) . sub ( user .
       rewardDebt ) ;

   user . amount = user . amount . sub (_amount) ;
   user . rewardDebt = user . amount . mul ( pool . accPerShare ) . div (1 e12 ) ;

   safeTransfer (msg. sender , pending ) ; pool . lpToken .
   safeTransfer ( address (msg. sender ) , _amount) ;
   emit Withdraw (msg. sender , _pid , _amount) ;

}
```

Listing 3.14: MasterChef.sol (revised)

Status This issue has been confirmed. Due to the same reason as outlined in Section 3.3, the team prefers not modifying the live code and leaves the code as it is.

3.7 Improved Logic in getMultiplier()

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Medium

10. Target: `MasterChef`
11. Category: Status Codes [12]
12. CWE subcategory: CWE-682 [4]

Description

The early incentives are greatly facilitated by a helper function called `getMultiplier()`. This function takes two arguments, i.e., `_from` and `_to`, and calculates the reward multiplier for the given block range (`[_from, _to]`).

```
    // Return reward multiplier over the given _from to _to block.
    function getMultiplier(uint256 _from, uint256 _to) public view returns (uint256) {
        if (_to <= bonusEndBlock) { return _to.sub(_from).mul(BONUS_MULTIPLIER);
        } else if (_from >= bonusEndBlock) { return _to.sub(_from);
        } else { return bonusEndBlock.sub(_from).mul(BONUS_MULTIPLIER).add(_to.sub(bonusEndBlock)
            );
        }
    }
```

Listing 3.15: MasterChef.sol

For elaboration, the helper's code snippet is shown above. We notice that this helper does not take into account the initial block (startBlock) from which the inventive rewards start to apply. As a result, when a normal user gives arbitrary arguments, it could return wrong reward multiplier! A correct implementation needs to take startBlock into account and appropriately re-adjusts the starting block number, i.e., _from = _from >= startBlock ? _from : startBlock.

We also notice that the helper function is called by two other routines, e.g., pending() and updatePool(). Fortunately, these two routines have ensured _from >= startBlock and always use the correct reward multiplier for reward redistribution.

Recommendation Apply additional sanity checks in the getMultiplier() routine so that the internal _from parameter can be adjusted to take startBlock into account.

```
// Return reward multiplier over the given _from to _to block.
function getMultiplier(uint256 _from, uint256 _to) public view returns (uint256) {
    _from = _from >= startBlock ? _from : startBlock; if (_to <= bonusEndBlock)
    { return _to . sub (_from) . mul (BONUS_MULTIPLIER) ;
    } else if (_from >= bonusEndBlock) { return _to . sub (_from) ;
    } else {
    return bonusEndBlock . sub (_from) . mul (BONUS_MULTIPLIER) . add ( _to . sub
    ( bonusEndBlock )
    ) ; }
}
```

Listing 3.16: MasterChef.sol

Status This issue has been confirmed. Due to the same reason as outlined in Section 3.3, the team prefers not modifying the live code and leaves the implementation as it is. As discussed earlier, the current callers provide the arguments that have been similarly verified to always obtain correct reward multipliers. Meanwhile, the team has been informed about possible misleading results as external inquiries on the `getMultiplier()` routine may provide arbitrary arguments that do not take into account the initial block, i.e., `startBlock`.

3.8 Improved EOA Detection Against Front-Running of Revenue Conversion

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Maker`
- Category: Status Codes [12]
- CWE subcategory: CWE-682 [4]

```
function convert ( address token0 , address token1 ) public { // At least we try to
    make front-running harder to do.
    require (! Address . isContract (msg. sender ) , "do not convert from contract") ; Pair pair = I2Pair
    ( factory . getPair ( token0 , token1 ) ) ; pair . transfer ( address ( pair ) , pair . balanceOf ( address ( this ) ) ) ; pair .
    burn ( address ( this ) ) ;
    uint256 wBNBAmount = _toWBNB( token0 ) + _toWBNB( token1 ) ;
_to(wBNBAmount) ; }
```

Listing 3.17: Maker.sol

The conversion of collected revenues into `BUSD` is implemented in `convert()`. Due to possible revenues, this routine could be a target for front-running (and further facilitated by flash loans) to steal the majority of collected revenues, resulting in a loss for current stakers in `DEFIAI Pools`.

As a defense mechanism, `maker` takes a pro-active measure by only allowing EOA accounts when the revenues are being converted. The detection of whether the transaction sender is an EOA or contract is based on the `isContract()` routine borrowed from the `Address` library (shown below).

```
/**
 *     @dev Returns true if ‘account‘ is a contract.
 *
 *     [IMPORTANT]
 *     ====
 *     It is unsafe to assume that an address for which this function returns
 * false is an externally-owned account (EOA) and not a contract.
 *
 *     Among others, ‘isContract‘ will return false for the following * types
 of addresses:
 *
 *     - an externally-owned account
 *     - a contract in construction
 *     - an address where a contract will be created
 *     - an address where a contract lived, but was destroyed
 *     ====
 */ function isContract ( address account ) internal view returns ( bool ) { // This method relies in
extcodesize, which returns 0 for contracts in // construction, since the code
is only stored at the end of the // constructor execution.

    uint256 size;
    // solhint-disable-next-line no-inline-assembly
    assembly { size := extcodesize ( account ) } return size > 0;
}
```

Listing 3.18: Address.sol

The current `isContract()` could achieve its goal in most cases. However, as mentioned in the library documentation, *“it is unsafe to assume that an address for which this function returns false is an externally-owned account (EOA) and not a contract.”* Considering the specific context, we need a reliable method to detect the `convert()` transaction sender is an externally-owned account, i.e., `EOA`. With that, we can simply achieve our goal by `require(msg.sender==tx.origin, "do not convert from contract")`

3.9 No Pair Creation With Zero Migration Balance

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Migrator`
- Category: Business Logics [10]
- CWE subcategory: CWE-770 [6]

Description

As discussed in Section 3.1, the actual bulk work of migration is performed by the `Migrator` contract, specifically its `migrate()` routine (we show the related code snippet below). This specific routine basically burns the LP tokens to reclaim the underlying assets and then transfers them.

```
26
27
28     function migrate (Pair orig ) public returns (Pair ) { require (msg. sender == chef , "not from master
29         chef") ; require ( block . number >= notBeforeBlock , "too early to migrate") ; require
30         ( orig . factory () == oldFactory , "not from old factory") ; address token0 = orig . token0 () ;
31         address token1 = orig . token1 () ;
32         IPair pair = Pair ( factory . getPair ( token0 , token1 ) ) ; i f ( pair == IPair ( address (0) ) ) { pair = IPair ( factory .
33         createPair ( token0 , token1 ) ) ;
34         }
35          uint256        lp = orig . balanceOf (msg. sender ) ;
36         i f ( lp == 0) return pair ; d e s i r e d L i q u i d i t y = lp ; orig . transferFrom
37         (msg. sender , address ( orig ) , lp ) ; orig . burn ( address ( pair ) ) ; pair . mint
38         (msg. sender ) ; d e s i r e d L i q u i d i t y = uint256 (*1);
39          return pair ;
40     }
41
42
43
44
```

In the unlikely situation when the migrated pool does have any balance for migration, `migrate()` routine is expected to simply return. However, it is interesting to notice that the `return` (line 37) does not happen until the new pair is created. A new pair creation may cost more than 2,000,000 gasses. Considering the current congested Binance blockchain and the relatively prohibitive gas cost, it is inappropriate to spend the gas cost to create a new pair when the balance is `zero` and no migration actually occurs.

Recommendation Move the balance detection logic earlier so that we can simply return without migration and new pair creation if the balance is `zero` , i.e., `orig.balanceOf(msg.sender)== 0`. An example adjustment is shown below.

```
function migrate ( IPair orig ) public returns Pair ) { require (msg. sender == chef , "not from master
chef") ;

        uint256            lp = orig . balanceOf (msg.
        i f ( lp == 0) return pair ;

        require ( orig . factory () == oldFactory , "not from old factory") ; address token0 = orig .
        token0 () ; address token1 = orig . token1 () ;
        } orig . transferFrom (msg. sender ,     address ( orig ) ,        lp ) ; orig . burn
        ( address ( pair ) ) ;

        d e s i r e d L i q u i d i t y = lp ; pair . mint
        (msg. sender ) ; d e s i r e d L i q u i d i t y =
        uint256 (*1) ;
         return pair ;

}
```

3.10 Full Charge of Proposal Execution Cost From Accompanying msg.value

- ID: PVE-010 • Target: `GovernorAlpha`
- Severity: Low • Category: Business Logics [10]
- Likelihood: Low • CWE subcategory: CWE-770 [6]
- Impact: Low

Description

`DEFIAI` adopts the governance implementation from `Compound` by adjusting its governance token and related parameters, e.g., `quorumVotes()` and `proposalThreshold()`. The original governance has been successfully audited by `OpenZeppelin`.

In the following, we would like to comment on a particular issue regarding the proposal execution cost. Notice that the actual proposal execution is kicked off by invoking the governance's `execute()` function. This function is marked as `payable`, indicating the transaction sender is responsible for supplying required amount of `BNBs` as each inherent action (line 215) in the proposal may require accompanying certain `BNBS`, specified in `proposal.values[i]`, where *i* is the i^{th} action inside the proposal.

```
210
211     function execute ( uint proposalId ) public payable { require ( state ( proposalId ) == ProposalState . Queued ,
            "GovernorAlpha::execute:
212             proposal can only be executed if it is queued") ;
213         Proposal storage proposal = proposals [ proposalId ] ; proposal . executed = true ;
214         for ( uint i = 0; i < proposal . ta rge ts . length ; i++) {
215             timelock . executeTransaction . value ( proposal . values [ i ] ) ( proposal . ta rge ts [ i ] , proposal . values [ i ] ,
                    proposal . s i g n a tu r e s [ i ] , proposal . c a l l d a t a s [ i ] , proposal . eta ) ;
            }
216         emitProposalExecuted ( proposalId ) ; }
217
218
```

Listing 3.22: GovernorAlpha.sol

Though it is likely the case that a majority of these actions do not require any `BNBs`, i.e., `proposal.values[i] = 0`, we may be less concerned on the payment of required `BNBs` for the proposal execution. However, in the unlikely case of certain particular actions that do need `BNBs`, the issue of properly attributing the associated cost arises. With that, we need to better keep track of `BNB` charge for each action and ensure that the transaction sender (who initiates the proposal execution) actually pays the cost. In other words, we do not rely on the governance's balance of `BNB` for the payment.

Recommendation Properly charge the proposal execution cost by ensuring the amount of accompanying BNB deposit is sufficient. If necessary, we can also return possible leftover in `msgValue` back to the sender.

```
function execute ( uint proposalId ) public payable { require ( state ( proposalId ) == ProposalState . Queued ,
    "GovernorAlpha::execute:
        proposal can only be executed if it is queued") ;
    Proposal storage proposal = proposals [ proposalId ] ; proposal . executed = true ;
    uint msgValue = msg. value ;
    for ( uint i = 0; i < proposal . ta rge ts . length ; i++) { inValue = sub256 ( msgValue ,
        proposal . values [ i ] )
        timelock . executeTransaction . value ( proposal . values [ i ] ) ( proposal . ta rge ts [ i ] , proposal . values [ i ] ,
            proposal . s i g n a tu r e s [ i ] , proposal . c a l l d a t a s [ i ] , proposal . eta ) ;
    }
    emitProposalExecuted ( proposalId ) ; }
```

Listing 3.23: GovernorAlpha.sol

Status This issue has been confirmed.

3.11 Improved Handling of Corner Cases in Proposal Submission

- ID: PVE-011
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `GovernorAlpha`
- Category: Business Logics [10]
- CWE subcategory: CWE-837 [7]

Description

As discussed in Section 3.10, `DEFIAI` adopts the governance implementation from `Compound` by accordingly adjusting its governance token and related parameters, e.g., `quorumVotes()` and `proposalThreshold ()`. Previously, we have examined the payment of proposal execution cost. In this section, we elaborate one corner case during a proposal submission, especially regarding the proposer qualification.

To be qualified to be proposer, the governance subsystem requires the proposer needs to obtain a sufficient number of votes, including from the proposer herself and other voters. The threshold is specified by `proposalThreshold()`.

```
function propose ( address [ ] memory targets , uint [ ] memory values , string [ ] memory
    signatures , bytes [ ] memory calldatas , ( uint ) { string memory d e s c r i p t i o n ) public returns
```

```
155         require(getPriorVotes(msg.sender, sub256(block.number, 1)) > proposalThreshold(), "GovernorAlpha::propose: proposer votes below proposal threshold");
156         require(targets.length == values.length && targets.length == signatures.length && targets.length == calldatas.length, "GovernorAlpha::propose: proposal function information arity mismatch");
157         require(targets.length != 0, "GovernorAlpha::propose: must provide actions");
158         require(targets.length <= proposalMaxOperations(), "GovernorAlpha::propose: too many actions");

160         uint latestProposalId = latestProposalIds[msg.sender];
161         if (latestProposalId != 0) {
162           ProposalState proposersLatestProposalState = state(latestProposalId);
163           require(proposersLatestProposalState != ProposalState.Active, "GovernorAlpha::propose: one live proposal per proposer, found an already active proposal");
164           require(proposersLatestProposalState != ProposalState.Pending, "GovernorAlpha::propose: one live proposal per proposer, found an already pending proposal");
165         }
166         ...
167     }
```

Listing 3.24: GovernorAlpha.sol

If we examine the `propose()` logic, when a proposal is being submitted, the governance verifies upfront the qualification of the proposer (line 155): `require(getPriorVotes(msg.sender, sub256( block.number, 1))> proposalThreshold(), "GovernorAlpha::propose: proposer votes below proposal threshold")`. Notice that the number of prior votes is strictly higher than `proposalThreshold()`.

However, if we check the proposal cancellation logic, i.e., the `cancel()` function, a proposal can be canceled (line 225) if the number of prior votes (before current block) is strictly smaller than `proposalThreshold()`. The corner case of having an exact number prior votes as the threshold, though unlikely, is largely unattended. It is suggested to accommodate this particular corner case as well.

```
function cancel ( uint proposalId ) public { ProposalState state = state ( proposalId ) ;
    require ( state != ProposalState . Executed , "GovernorAlpha::cancel: cannot cancel executed
        proposal") ;

    Proposal storage proposal = proposals [ proposalId ] ;
    require (msg. sender == guardian | | . getPriorVotes ( proposal . proposer , sub256 ( block . number , 1) ) < proposalThreshold
        () , "GovernorAlpha::cancel: proposer above threshold");

    proposal . canceled = true ;
    for ( uint i = 0; i < proposal . ta rge ts . length ; i++) { timelock . cancelTransaction ( proposal . targ ets [ i ] , proposal . values
    [ i ] , proposal . s i g n a tu r e s [ i ] , proposal . c a l l d a t a s [ i ] , proposal . eta ) ; }

    emit ProposalCanceled ( proposalId ) ;
}
```

Status This issue has been confirmed.

3.12 Inconsistency Between Documented and Implemented Inflation

- ID: PVE-012
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `MasterChef`
- Category: Business Logics [10]
- CWE subcategory: CWE-837 [7]

Description

According to the documentation, *"At every block, 100 tokens will be created. These tokens will be equally distributed to the stakers of each of the supported pools."*

As part of the audit process, we examine and identify possible inconsistency between the documentation/white paper and the implementation. Based on the smart contract code, there is a system-wide configuration. This particular parameter is initialized as 100 when the contract is being deployed and it can only be changed at the contract's constructor. The initialized number of 100 seems consistent with the documentation

A further analysis about the inflation logic (implemented in `updatePool()`) shows certain inconsistency that needs to better articulated and clarified. For elaboration, we show the related code snippet below.

```
// Update reward variables of the given pool to be up-to-date.
function updatePool ( uint256 _pid ) public
{ PoolInfo storage pool = poolInfo [ _pid ] ; i f ( block .
number <= pool . lastRewardBlock ) { return ;
} uint256 lpSupply = pool . lpToken . balanceOf ( address
( this ) ) ; i f ( lpSupply == 0) { pool . lastRewardBlock = block .
number ;
return ;
}
uint256 m u l t i p l i e r = g e t M u l t i p l i e r ( pool . lastRewardBlock , block . number) ; uint256 Reward = m u l t i p
l i e r . mul ( PerBlock ) . mul ( pool . a l l o c P o i n t ) . div ( t o t a l A l l o c P o i n t ) ;
. mint ( devaddr ,Reward . div (10) ) ; . mint ( address
( this ) ,Reward ) ;
pool . acciPerShare = pool . accPerShare . add ( Reward . mul (1 e12 ) . div ( lpSupply ) ) ;
pool . lastRewardBlock = block . number ;
}
```

Listing 3.27: MasterChef.sol

The parameter indeed controls the number of `rewards` that are distributed to various pools (line 196). However, it further adds another 10% of the calculated `BUSD` to the development team- controlled account (line 195). With that, the number of new rewards per block should be 110, not 100!

Status This issue has been confirmed.

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.