CYBERTIME.FINANCE

CyberTime Finance

Security Assessment

February 23, 2021

Prepared For:
Dmitriy Boshenyatov | CyberTime Finance

Prepared By:
Adesh Kolte | *Security Researcher*

TABLE OF CONTENTS

1. EXECUTIVE SUMMARY

1.1. Testing overview

The security review of the Cybertime Finance smart contracts was meant to verify whether the proper security mechanisms were in place to prevent users from abusing the contracts' business logic and to detect the vulnerabilities which could cause financial losses to the client or its customers.

Security tests were performed using the following methods:

- Manual source code review,
- Automatic scans using security verification tools,
- Verification of compliance with smart contract weakness classification

1.2. Basic information

Security Auditor	Adesh Kolte
Testing time period	2021-02-11. – 2021-02-20
Report date	2021-02-20 – 2021-02-22

1.3. Target in scope

The object being analyzed was CyberTime Finance platform accessible in the following GitHub repository:

https://github.com/cybertime-eth/smart-contracts

The contracts reviewed were the following:

- /farming/CTFfarming.sol
- /farming/NFTLfarming.sol
- /token/ CTFtoken.sol
- /token/NFTLtoken.sol

1.4. Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues), including, 2 medium-severity vulnerabilities, 1 low-severity vulnerabilities and 2 informational

ID	Severity	Title	Category	Status
CTFI-001	Medium	Duplicate Pool Detection and Prevention	Business Logics	Resolved
CTFI-002	Informational	Recommended Explicit Pool Validity Checks	Security Features	Resolved
CTFI-003	Informational	Incompatibility with Deflationary Tokens	Business Logics	Resolved
CTFI-004	Low	Suggested Adherence of Checks-Effects-Interactions	Time and State	Resolved
CTFI-005	Medium	Inconsistency Between Documented and Implemented CTF, NFTL Inflation	Business Logic	Resolved

2. DETAILED VULNERABILITIES

2.1. Duplicate Pool Detection and Prevention

***Severity**: Medium*

***Contract(s) affected**: CTFfarming.sol, NFTLFarming.sol*

Description

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 UniswapV2 LP 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 , IERC20 _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 , accCTFPerShare : 0
        }) )
        ; }
```

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 ( IERC20 _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 , IERC20 _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 ,
       accCTFPerShare : 0 }) ) ;

            }
```

.

2.2. Recommended Explicit Pool Validity Checks

Severity: *Medium*

Contract(s) affected*: CTFfarming.sol, NFTLfarming.sol*

Description

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  {
  IERC20  lpToken ;// Address of LP token contract.
     uint256 a l l o c P o i n t ;  // How many allocation points assigned to this pool. CTFs to
          distribute per block.
     uint256  lastRewardBlock ; // Last block number that CTFs distribution occurs.
     uint256 accCTFPerShare ; // Accumulated CTFs per share, times 1e12. See below
          .
}
...
```

When there is a need to add a new pool, set a new `allocPoint` for an existing pool, stake (by depositing the supported `UniswapV2`'s LP tokens), unstake (by redeeming previously deposited `UniswapV2`'s LP tokens), query pending CTF rewards, 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 Farming contracts for CTF, NFTL 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 . accCTFPerShare ) . div (1 e12 ) . sub ( user .rewardDebt ) ;
              safeCTFTransfer (msg. sender, pending ) ;
      } pool . lpToken . safeTransferFrom ( address (msg. sender ) ,address ( this ) ,_amount) ;
       user . amount = user . amount . add (_amount) ;
       user . rewardDebt = user . amount . mul ( pool . accCTFPerShare ) . div (1 e12 ) ;
                emit Deposit (msg. sender ,_pid ,_amount) ;
}
```

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()`, `pendingCTF ()` 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?");
  _;
}

// Deposit LP tokens to Farming contracts for CTF NFTL allocation.
Function deposit ( uint256 _pid ,uint256 _amount) public validatePool ( _pid ) {
       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 . accCTFPerShare ) . div (1 e12 ) . sub ( user .
                rewardDebt ) ;
            safeCTFTransfer (msg. sender , pending ) ;
        } pool . lpToken . safeTransferFrom ( address (msg. sender ) , address ( this ) , _amount) ;
        user . amount = user . amount . add (_amount) ;
        user . rewardDebt = user . amount . mul ( pool . accCTFPerShare ) . div (1 e12 ) ;
        emit Deposit (msg. sender , _pid , _amount) ;
```

2.3. Incompatibility with Deflationary Tokens

***Severity**: Medium*

***Contract(s) affected:** CTFfarming.sol, NFTLfarming.sol*

Description

Naturally, the above two functions in CTF/NFTL Farming contracts, i.e., deposit() and withdraw(), are involved in transferring users's assets into (or out of) the CyberTime Finance. Using the deposit() function as an example, it needs to transfer deposited assets from the user account to the pool).

When transferring standard ERC20 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 ERC20 token contracts

```
// Deposit LP tokens to Farming contracts for CTF, NFTL 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 . accCTFPerShare ) . div (1 e12 ) . sub ( user .rewardDebt ) ;
            safeCTFTransfer (msg. sender, pending ) ;
      } pool . lpToken . safeTransferFrom ( address (msg. sender ) ,address ( this ) ,_amount) ;
       user . amount = user . amount . add (_amount) ;
       user . rewardDebt = user . amount . mul ( pool . accCTFPerShare ) . div (1 e12 ) ;
                emit Deposit (msg. sender ,_pid ,_amount) ;
}
```

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 ERC20 token contracts in the cases of deflationary tokens. Apparently, these balance inconsistencies are damaging to accurate portfolio management of `farming contracts` and affects protocol-wide operation and maintenance.

One mitigation is to query the asset change right before and after the asset-transferring

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.

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

2.4. Suggested Adherence of Checks-Effects-Interactions

***Severity**: Low*

***Contract(s) affected**: CTFfarming.sol, NFTLfarming.sol*

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 Ethereum history, including the `DAO` [22] exploit, and the recent `Uniswap/Lendf.Me` hack .

We notice there are several occasions the `checks-effects-interactions` principle is violated. Using the `farming contracts` 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 starts before effecting the update on internal states , 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

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

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

```
// Deposit LP tokens to Farming contracts for CTF, NTFL 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 . accCTFPerShare ) . div (1 e12 ) .
            sub ( user .rewardDebt ) ;
      safeCTFTransfer (msg. sender ,pending ) ;
            } pool . lpToken . safeTransferFrom ( address (msg. sender ) , address ( this ) , _amount) ;
            user . amount = user . amount . add (_amount) ; user . rewardDebt = user . amount . mul ( pool . accCTFPerShare ) .
            div (1 e12 ) ;
      emit Deposit (msg. sender ,_ pid , _amount) ;
      }

      // Withdraw LP tokens from Farming contract.
      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 . accCTFPerShare ) . div (1 e12 ) . sub ( user .
                  rewardDebt ) ;
            safeCTFTransfer (msg. sender ,          pending ) ; user . amount = user . amount . sub (_amount) ; user .
            rewardDebt = user . amount . mul ( pool . accCTFPerShare ) . div (1 e12 ) ; pool . lpToken . safeTransfer (
            address (msg. sender ) , _amount) ;
            emit    Withdraw (msg. sender , _pid , _amount) ;
      }
```

Recommendation Apply necessary reentrancy prevention by following the `checks-effects interactions` best practices. 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 [ _pid ] ;
    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 Farming contacts for CTF, NFTL
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 ) ; uint256 pending = user . amount . mul ( pool . accCTFPerShare ) . div (1 e12 ) . sub ( user .
        rewardDebt ) ;

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


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


// Withdraw LP tokens from Farming contracts
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 . accCTFPerShare ) . div (1 e12 ) . sub ( user .
        rewardDebt ) ;

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

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

2.5. Inconsistency Between Documented and Implemented CTF or NFTL Inflation

***Severity**: Low*

***Contract(s) affected**: CTFfarming.sol, NFTLfarming.sol*

Description

According to the documentation of CyberTime Finance , ”At every block, 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, i.e., `CTFPerBlock`,`NFTLperBlock`. 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 and `CTFPerBlock`, `NFTLperBlock` is fixed forever (and cannot be adjusted even via a governance process).

A further analysis about the CTF, NFTL tokens 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 mul 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 ctfReward = m u l t i p l i e r . mul ( ctfPerBlock ) . 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 ) ;
      ctf . mint ( devaddr , ctfReward . div (10) ) ;
      ctf . mint ( address ( this ), ctfReward ) ;
      pool . accPerShare = pool . accCTFPerShare . add ( CTFReward . mul (1 e12 ) . div ( lpSupply ) ) ;
      pool . lastRewardBlock = block . number ;
}
```

The `ctfPerBlock`, `nftlperblock` parameter indeed controls the number of CTF, NFTL rewards that are distributed to various pools. However, it further adds another 10% of the calculated `ctfReward`, `nftlreward` to the development team-controlled account . With that, the number of new CTF, NFTL rewards per block should be 110, not 100!

Recommendation Clarify the inconsistency by clearly stating the number of new new CTF, NFTL tokens is 110, and the development team will be receiving about 1_11=9.09% of total new CTF, NFTL distribution.

3. CONCLUSION

In this audit, we thoroughly analyzed the CyberTime Finance design and implementation. Overall, Cybertime finance presents an evolutional improvement and provide extra incentives to liquidity providers. Our impression is that the current code base is well organized and those identified issues are promptly confirmed and fixed. The main concern, however, is related to the current deployment as its privilege management is not under the control of community-based governance. Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

4. APPENDIX A – REFERENCES

https://github.com/crytic/building-secure-contracts/tree/master/development-guidelines

https://swcregistry.io/

https://consensys.github.io/smart-contract-best-practices/

5. APPENDIX B – AUDITOR QUALIFICATIONS

Adesh is an Independent Information Security Consultant focusing on security assessments (applications, infrastructures and smart contracts) Previously worked as bug bounty hunter with Multiple organizations like Microsoft (MSRC), Apple, Google, Zoho, Siemens etc

Twitter - https://twitter.com/AdeshKolte
Medium - https://ad3sh.medium.com/

End of document