Smart contract security audit report

NewBFarm smart contract security audit report

Audit Team : Noneage security team

Audit Date : May 6 , 2021

NewBFarm Smart Contract Security Audit Report

1. Overview

On April 30, 2021, the security team of Noneage Technology received the security audit request of the **NewBFarm project**. The team completed the **NewBFarm smart contract** security audit on May 6. the security audit experts of Noneage Technology communicate with the relevant interface people of the NewBFarm project, maintain information symmetry, conduct security audits under controllable operational risks, and try to avoid project generation and operation during the test process. Cause risks.

Through communicat and feedback with NewBFarm project party, it is confirmed that the loopholes and risks found in the audit process have been repaired or within the acceptable range. The result of this NewBFarm smart contract security audit: **passed.**

Audit Report MD5： 3E99EDB883ECC82313DADB6D8426C793

2. Background

2.1 Project Description

> Project name: NewBFarm
>
> Contract type: DeFi Token contract
>
> Code language: Solidity
>
> Project official Github: https://github.com/NewBFarm/vault
>
> Contract documents: Controller.sol，StakePool.sol，StrategyBake.sol，StrategyLp.sol，V2 strategy.sol，V2 strategy BNB.sol，Vault.sol

2.2 Audit Range

The contract file and corresponding MD5 provided by NewBFarm:

Vault.sol	6EE9C5385A9588A6899E4C174CE57C3D
StakePool.sol	3F836AD06057B2FEDDF755F12AC7D34C
Controller.sol	F2F25EF96D07DF94A13C3DCD76BAC287
StrategyLp.sol	4F82E4FBFAB50B55DDF0A598114F469F
V2 strategy.sol	F3D4844FD8102B5068E9E391368866D9
StrategyBake.sol	E9A61DE2514F5894A920339EF5FECE0B

2.3 Security Audit List

The security experts of Noneage Technology conduct security audits on the security audit list within the agreement, The scope of this smart contract security audit does not include new attack methods that may appear in the future, does not include the code after contract upgrades or tampering, and is not included in the subsequent cross-country, does not include cross-chain deployment, does not include project front-end code security and project platform server security.

This smart contract security audit list includes the following:

- Integer overflow
- Reentry attack
- Floating point numbers and numerical precision
- Default visibility
- Tx.origin authentication
- Wrong constructor
- Return value not verified
- Insecure random numbers
- Timestamp dependency
- Transaction order is dependent
- Delegatecall
- Call
- Denial of service
- Logic design flaws
- Fake recharge vulnerability
- Short address attack
- Uninitialized storage pointer
- Additional token issuance
- Frozen account bypass
- Access control
- Gas usage

3. Contract Structure Analysis

3.1 Directory Structure

```
│
├─NewBFarm
│      Controller.sol
│      StakePool.sol
│      StrategyBake.sol
│      StrategyLp.sol
│      V2 strategy BNB.sol
│      V2 strategy.sol
│      Vault.sol
```

3.2 NewBFarm contract

Contract

Controller

- setStrategist(address _strategist)
- setGovernance(address _governance)
- setVault(address _token, address _vault)
- approveStrategy(address _token, address _strategy)
- revokeStrategy(address _token, address _strategy)
- setConverter(address _input, address _output, address _converter)
- setStrategy(address _token, address _strategy)
- earn(address _token, uint _amount)
- balanceOf(address _token)
- withdrawAll(address _token)
- inCaseTokensGetStuck(address _token, uint _amount)
- inCaseStrategyTokenGetStuck(address _strategy, address _token)
- withdraw(address _token, uint _amount)
- rewards()

IRewardDistributionRecipient

- setRewardDistribution(address _rewardDistribution)

LPTokenWrapper

- totalSupply()
- balanceOf(address account)
- stake(uint256 amount)
- withdraw(uint256 amount)

USDTLavaRewards2

- lastTimeRewardApplicable()
- rewardPerToken()
- earned(address account)
- stake(uint256 amount)
- withdraw(uint256 amount)
- exit()
- getReward()
- notifyRewardAmount(uint256 reward)

StrategyBakeWar

- doApprove ()
- deposit()
- withdraw(IERC20 _asset)
- withdraw(uint _amount)
- withdrawAll()
- _withdrawAll()
- harvest()
- getNumOfRewards()
- doswap()
- dosplit()
- _withdrawSome(uint256 _amount)

- balanceOfWant()
- balanceOfPool()
- balanceOf()
- setGovernance(address _governance)
- setController(address _controller)
- setFee(uint256 _fee)
- setStrategyFee(uint256 _fee)
- setCallFee(uint256 _fee)
- setBurnFee(uint256 _fee)
- setBurnAddress(address _burnAddress)
- setWithdrawalFee(uint _withdrawalFee)

StrategyLp

- doApprove ()
- deposit()
- withdraw(IERC20 _asset)
- withdraw(uint _amount)
- withdrawAll()
- _withdrawAll()
- harvest()
- getNumOfRewards()
- doswap()
- dosplit()
- _withdrawSome(uint256 _amount)
- balanceOfWant()
- balanceOfPool()
- balanceOf()
- setGovernance(address _governance)
- setController(address _controller)
- setFee(uint256 _fee)
- setStrategyFee(uint256 _fee)
- setCallFee(uint256 _fee)
- setBurnFee(uint256 _fee)
- setBurnAddress(address _burnAddress)
- setWithdrawalFee(uint _withdrawalFee)

StrategyBNB

- setStrategist(address _strategist)
- setWithdrawalFee(uint256 _withdrawalFee)
- setStrategistReward(uint256 _strategistReward)
- setReinvest(uint256 _reinvest)
- deposit()
- withdraw(IERC20 _asset)
- withdraw(uint256 _amount)
- _withdrawSome(uint256 _amount)
- withdrawAll()
- _withdrawAll()
- harvest()
- balanceOfWant()
- balanceOfPool()
- balanceOf()
- setGovernance(address _governance)

- setController(address _controller)

StrategyBUSD

- addFarmer(address f)
- removeFarmer(address f)
- setGovernance(address _governance)
- setStrategist(address _strategist)
- setWithdrawalFee(uint256 _withdrawalFee)
- setStrategistReward(uint256 _strategistReward)
- setReinvest(uint256 _reinvest)
- getSuppliedView()
- getBorrowedView()
- balanceOfPool()
- balanceOfWant()
- balanceOf()
- getSupplied()
- getBorrowed()
- harvest()
- deposit()
- _deposit()
- _withdrawSome(uint256 _amount)
- withdrawAll()
- _withdrawAll()
- withdraw(uint256 _amount)
- withdraw(IERC20 _asset)
- e_exit()
- e_redeem(uint amount)
- e_redeemAll()
- e_redeemCToken(uint amount)
- e_redeemAllCToken()
- e_repayAll()
- e_repay(uint amount)
- e_collect()

zVault

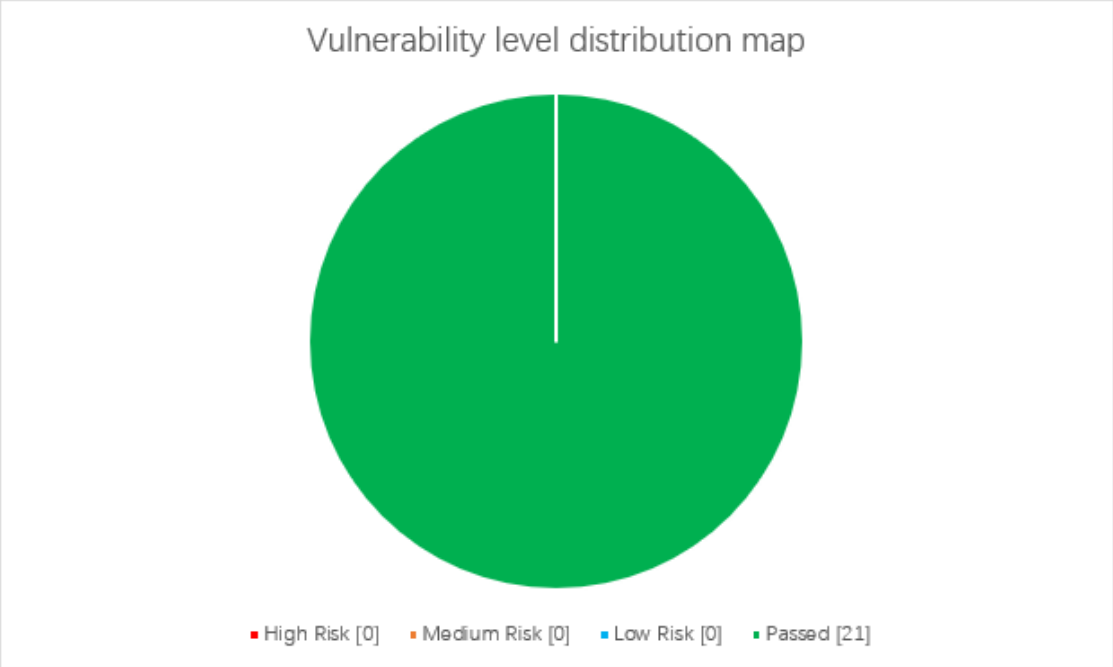
- balance()
- setMin(uint _min)
- setGovernance(address _governance)
- setController(address _controller)
- setEarnLowerlimit(uint256 _earnLowerlimit)
- available()
- earn()
- depositAll()
- deposit(uint _amount)
- withdrawAll()
- withdraw(uint _shares)
- getPricePerFullShare()

4. Audit Details

4.1 Vulnerabilities Distribution

Vulnerabilities in this security audit are distributed by risk level, as follows:

Vulnerability level distribution			
High risk	Medium risk	Low risk	Passed
0	0	0	21



This smart contract security audit has 0 high-risk vulnerabilities, 0 medium-risk vulnerabilities, 0 low-risk vulnerabilities, and 21 passed, with a high security level.

4.2 Vulnerabilities Details

A security audit was conducted on the smart contract within the agreement, and no security vulnerabilities that could be directly exploited and generated security problems were found, and the security audit was passed.

4.3 Other Risks

Other risks refer to the code that contract security auditors consider to be risky, which may affect the stability of the project under certain circumstances, but cannot constitute a security issue that directly endangers the security.

4.3.1 Safe use of tx.origin variables

- **Question detail**

The NewBFarm contract uses the global variable tx.origin to give the initial caller the administrator authority. Since tx.origin traverses the entire call stack and returns the address of the account that originally sent the call, it may happen that the current contract is called by the intermediate contract during the call.

```
    constructor() public {
        governance = tx.origin;
    }
```

- **Safety advice**

It is recommended to change the tx.origin variable to the caller's own msg.sender.

- **Update status**

Through communication with the NewBFarm team, Safety advice has been adopted.

4.3.2 Address validity check

- **Question detail**

In the NewBFarm contract, the input addresses of multiple methods have not been validated, and empty addresses may appear, as shown in the following figure:

```
    function setGovernance(address _governance)

    function setVault(address _token, address _vault)

    function setConverter(address _input, address _output, address _converter)

    function setStrategy(address _token, address _strategy)

    function earn(address _token, uint _amount)

    function balanceOf(address _token)

    function withdrawAll(address _token)

    function inCaseTokensGetStuck(address _token, uint _amount)

    function withdraw(address _token, uint _amount)
```

- **Safety advice**

It is recommended to add a verification code for checking address validity, as shown below:

```
    function setGovernance(address _governance)
        require(_governance != address(0) , 'ADDRESS ERROR!!!');
```

- **Update status**

Through communication with the NewBFarm team, Safety advice has been adopted.

4.3.3 Administrator authority is too large

- **Question detail**

In the NewBFarm contract, the administrator can set multiple values and there is a transfer function. If the administrator is manipulated by a malicious person, it can cause abnormal capital loss and shake the market stability.

```
    function withdrawAll(address _token) public {
        require(msg.sender == governance, "!governance");
        Strategy(strategies[_token]).withdrawAll();
    }

    function inCaseTokensGetStuck(address _token, uint _amount) public {
        require(msg.sender == governance, "!governance");
        IERC20(_token).safeTransfer(governance, _amount);
    }
```

- **Safety advice**

On the premise of ensuring security, keep multiple copies of the private key reasonably.

- **Update status**

Through communication with the NewBFarm team, Safety advice has been adopted.

4.3.4 Function attribute problem

- **Question detail**

In the NewBFarm contract, multiple methods have authorization functions. To avoid authorization conflicts, the authorization limit will be changed to 0 each time it is authorized, and then modified to the corresponding value. Since both methods are public attributes, whether there is any address call here, causing other functions of the project to be unstable.

```
    function doApprove () public{
        IERC20(output).safeApprove(unirouter, 0);
        IERC20(output).safeApprove(unirouter, uint(-1));
    }
```

- **Safety advice**

The above method is recommended to add an administrator decorator.

- **Update status**

Through communication with the NewBFarm team, Safety advice has been adopted.

4.3.5 The amount is valid and safe

- **Question detail**

In the NewBFarm contract, the _amount value input by multiple methods has not been judged for validity, and it should be judged whether the value is greater than zero.

```
    function deposit(uint _amount) public {
        uint _pool = balance();

    function withdraw(uint _shares) public {
        uint r = (balance().mul(_shares)).div(totalSupply());
```

- **Safety advice**

Determine whether _amount is valid within the function, as shown in the following figure:

```
require(_amount > 0, erroramount);
```

- **Update status**

Through communication with the NewBFarm team, Safety advice has been adopted.

4.3.6 Quantity of destruction

- **Question detail**

In the NewBFarm contract, in the withdraw() method, the user can enter any value to perform the destruction operation, and the correctness of the destroyed quantity should be verified.

```
    function withdraw(uint _shares) public {
        uint r = (balance().mul(_shares)).div(totalSupply());
        _burn(msg.sender, _shares);
```

- **Safety advice**

Strictly determine the amount of destruction.

- **Update status**

Through communication with the NewBFarm team, Safety advice has been adopted.

4.3.7 Normal minting problem

- **Question detail**

In the NewBFarm contract, in the deposit() method, after the user performs the deposit operation, the deposit code logic will perform the minting operation to the caller's address. The minting value is shares, which is determined by the total amount of minting totalSupply and the balance method balance(). It consists of a balance, whether totalSupply and balance are maliciously controlled, which leads to an increase in the value of shares and thus obtains a large amount of coins.

```
    function deposit(uint _amount) public {
        uint _pool = balance();
        uint _before = token.balanceOf(address(this));
        token.safeTransferFrom(msg.sender, address(this), _amount);
        uint _after = token.balanceOf(address(this));
        _amount = _after.sub(_before);
        uint shares = 0;
        if (totalSupply() == 0) {
            shares = _amount;
        } else {
            shares = (_amount.mul(totalSupply())).div(_pool);
        }
        _mint(msg.sender, shares);
        if (token.balanceOf(address(this))>earnLowerlimit){
          earn();
        }
    }
```

- **Safety advice**

Strictly determine the number of coins.

- **Update status**

By communicating with the NewBFarm team, there is no impact.

4.3.8 Safe use of the fallback function

- **Question detail**

The NewBFarm contract uses the fallback function fallback(), which is decorated with payable and has no other functions.

```
receive() payable external {}
```

- **Safety advice**

If the fallback function has no special logic implementation, it is recommended to remove the function to avoid security issues.

- **Update status**

Through communication with the NewBFarm team, Safety advice has been adopted.

5. Security Audit Tool

Tool name	Tool Features
Oyente	Can be used to detect common bugs in smart contracts
securify	Common types of smart contracts that can be verified
MAIAN	Multiple smart contract vulnerabilities can be found and classified
Noneage Internal Toolkit	Noneage(hawkeye system) self-developed toolkit + https://audit.noneage.com

6. Vulnerability assessment criteria

Vulnerability level	Vulnerability description
High risk	Vulnerabilities that can directly lead to the loss of contracts or users' digital assets, such as integer overflow vulnerabilities, false recharge vulnerabilities, re-entry vulnerabilities, illegal token issuance, etc. Vulnerabilities that can directly cause the ownership change of the token contract or verification bypass, such as: permission verification bypass, call code injection, variable coverage, unverified return value, etc. Vulnerabilities that can directly cause the token to work normally, such as denial of service vulnerabilities, insecure random numbers, etc.
Medium risk	Vulnerabilities that require certain conditions to trigger, such as vulnerabilities triggered by the token owner's high authority, and transaction sequence dependent vulnerabilities. Vulnerabilities that cannot directly cause asset loss, such as function default visibility errors, logic design flaws, etc.
Low risk	Vulnerabilities that are difficult to trigger, or vulnerabilities that cannot lead to asset loss, such as vulnerabilities that need to be triggered at a cost higher than the benefit of the attack, cannot lead to incorrect coding of security vulnerabilities.

NONEAGE

Telephone：86-17391945345 18511993344

Email ：support@noneage.com

Site ：www.noneage.com

Weibo ：weibo.com/noneage