KNOWNSEC

Degree of secrecy ★ | Project team open

Smart Contract Audit Report

Security status

Safe

★ ★ ★ ★ ★

Principal tester: Knownsec blockchain security team

Version Summary

Content	Date	Version
Editing Document	20210531	V2.0

Report Information

Title	Version	Document Number	Type
NBF Smart Contract Audit Report	V2.0	c470f04297494f55b37e552f6a6179b9	Open to project team

Copyright Notice

Table of Contents

1. Introduction

The effective test time of this report is from From May 7, 2021 to May 31, 2021 . During this period, the security and standardization of the **NBF smart contract controller, pledge mining pool, strategy and vault contract code** will be audited and used as the statistical basis for the report.

The scope of this smart contract security audit does not include external contract calls, new attack methods that may appear in the future, and code after contract upgrades or tampering. (With the development of the project, the smart contract may add a new pool , New functional modules, new external contract calls, etc.), does not include front-end security and server security.

In this audit report, engineers conducted a comprehensive analysis of the common vulnerabilities of smart contracts (Chapter 3). **The smart contract code of the NBF** is comprehensively assessed as **SAFE**.

Results of this smart contract security audit： SAFE

Since the testing is under non-production environment, all codes are the latest version. In addition, the testing process is communicated with the relevant engineer, and testing operations are carried out under the controllable operational risk to avoid production during the testing process, such as: Operational risk, code security risk.

Report information of this audit:

Report Number：c470f04297494f55b37e552f6a6179b9

Report query address link:

https://attest.im/attestation/searchResult?qurey=c470f04297494f55b37e552f6a6179b9

Target information of the NBF audit:

Target information	
Project name	NBF
Token address	DeFi protocol code, BSC smart contract code

Code type	Controller	0x8d6AdC4A7d6b87e37a8DC5Ee6ee51601d5ff0498
	Vault:bnb-cake	0x6eC9f515F0c8803f8b2D49dE61d794eDf2A0119d
	Strategy:bnb-cake	0xeB5355e96C6833b606897277689be666aB924e89
	Vault:BUSD-bnb	0x0B1a3904B519546C237f2FF34ba0f3b4eFB99919
	Strategy:BUSD-bnb	0xd8A35C2F7Cd0657AD5898Cc076bb98BE244Dd8Ec
	Vault:btcb-busd	0x5932665A95d97e8830ef48f0E11a640D6ceEEeA5
	Strategy:btcb-busd	0x107bbf40DCe15b5f98F408c91f7158d36B764721
	Vault:usdc-busd	0x4e759842E6eDE8df415432f2fd6b4CBb5e2557D3
	Strategy:usdc-busd	0x3cceFBd61158441E64686f6329C22da9e21621B6
	Vault:dai-busd	0x16638EF0C74906F8410e89645e1Da54C5c6E33E1
	Strategy:dai-busd	0x4f55E7677408f76F4fb12335DE8B87720072DEcF
	Vault:usdt-busd	0x8Ff463d46bA4697612D0Dac626eeBb3F0193e2a6
	Strategy:usdt-busd	0xbA7556f847E8f98EF56413c510F929d50A99c617

Code language	Solidity

Contract documents and hash:

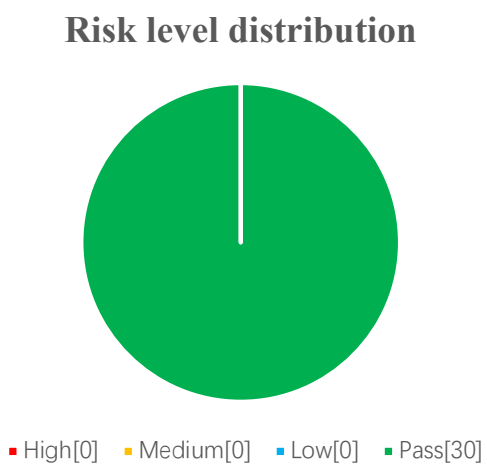
Contract documents	MD5
Controller.sol	97f652f62f7933e439bcf2fa1ffcd31d
StrategyLP.sol	c991182f4e46fb2ffc5541db468833f6
zVault.sol	01dce63e29d83f38bfc95b7966cc00d2

2. Code vulnerability analysis

2.1 Vulnerability Level Distribution

Vulnerability risk statistics by level：

Vulnerability risk level statistics table			
High	Medium	Low	Pass
0	0	0	30



2.2 Audit Result

Result of audit			
Audit Target	**Audit**	**Status**	**Audit Description**
Business security testing	**Controller contract**	Pass	After testing, there is no such safety vulnerability.
	Strategic contract	Pass	After testing, there is no such safety vulnerability.
	vault contract	Pass	After testing, there is no such safety vulnerability.
Basic code vulnerability detection	**Compiler version security**	Pass	After testing, there is no such safety vulnerability.
	Redundant code	Pass	After testing, there is no such safety vulnerability.
	Use of safe arithmetic library	Pass	After testing, there is no such safety vulnerability.
	Not recommended encoding	Pass	After testing, there is no such safety vulnerability.
	Reasonable use of require/assert	Pass	After testing, there is no such safety vulnerability.
	fallback function safety	Pass	After testing, there is no such safety vulnerability.
	tx.oriigin authentication	Pass	After testing, there is no such safety vulnerability.
	Owner permission control	Pass	After testing, there is no such safety vulnerability.

	Gas consumption detection	Pass	After testing, there is no such safety vulnerability.
	call injection attack	Pass	After testing, there is no such safety vulnerability.
	Low-level function safety	Pass	After testing, there is no such safety vulnerability.
	Vulnerability of additional token issuance	Pass	After testing, there is no such safety vulnerability.
	Access control defect detection	Pass	After testing, there is no such safety vulnerability.
	Numerical overflow detection	Pass	After testing, there is no such safety vulnerability.
	Arithmetic accuracy error	Pass	After testing, there is no such safety vulnerability.
	Wrong use of random number detection	Pass	After testing, there is no such safety vulnerability.
	Unsafe interface use	Pass	After testing, there is no such safety vulnerability.
	Variable coverage	Pass	After testing, there is no such safety vulnerability.
	Uninitialized storage pointer	Pass	After testing, there is no such safety vulnerability.
	Return value call verification	Pass	After testing, there is no such safety vulnerability.
	Transaction order dependency detection	Pass	After testing, there is no such safety vulnerability.

	Timestamp dependent attack	Pass	After testing, there is no such safety vulnerability.
	Denial of service attack detection	Pass	After testing, there is no such safety vulnerability.
	Fake recharge vulnerability detection	Pass	After testing, there is no such safety vulnerability.
	Reentry attack detection	Pass	After testing, there is no such safety vulnerability.
	Replay attack detection	Pass	After testing, there is no such safety vulnerability.
	Rearrangement attack detection	Pass	After testing, there is no such safety vulnerability.

3. Analysis of code audit results

3.1. Controller contract【PASS】

Audit analysis: The controller contract is implemented in the Controller.sol contract file, refer to the controller contract of the yearn protocol, delete some functions.

```
contract Controller {
    using SafeERC20 for IERC20;
    using Address for address;
    using SafeMath for uint256;

    address public governance;
    address public strategist;

    mapping(address => address) public vaults;
    mapping(address => address) public strategies;
    mapping(address => mapping(address => address)) public converters;

    mapping(address => mapping(address => bool)) public approvedStrategies;

    constructor() {
        governance = msg.sender;
        strategist = msg.sender;
    }

    function setStrategist(address _strategist) public {//
        require(msg.sender == governance, "!governance");
        require(_strategist != address(0), "ADDRESS ERROR!");
        strategist = _strategist;
    }

    function setGovernance(address _governance) public {//
```

```
        require(msg.sender == governance, "!governance");
        require(_governance != address(0), "ADDRESS ERROR!");
        governance = _governance;
    }

    function setVault(address _token, address _vault) public {//
        require(msg.sender == strategist || msg.sender == governance, "!strategist");
        require(_vault != address(0), "ADDRESS ERROR!");
        require(_token != address(0), "ADDRESS ERROR!");
        require(vaults[_token] == address(0), "vault");
        vaults[_token] = _vault;
    }

    function approveStrategy(address _token, address _strategy) public {
        require(msg.sender == governance, "!governance");
        require(_strategy != address(0), "ADDRESS ERROR!");
        require(_token != address(0), "ADDRESS ERROR!");
        approvedStrategies[_token][_strategy] = true;
    }

    function revokeStrategy(address _token, address _strategy) public {
        require(msg.sender == governance, "!governance");
        require(_strategy != address(0), "ADDRESS ERROR!");
        require(_token != address(0), "ADDRESS ERROR!");
        approvedStrategies[_token][_strategy] = false;
    }

    function setConverter(address _input, address _output, address _converter) public {
        require(msg.sender == strategist || msg.sender == governance, "!strategist");
        require(_input != address(0), "ADDRESS ERROR!");
        require(_output != address(0), "ADDRESS ERROR!");
        require(_converter != address(0), "ADDRESS ERROR!");
        converters[_input][_output] = _converter;
```

```
    }

    function setStrategy(address _token, address _strategy) public {//
        require(msg.sender == strategist || msg.sender == governance, "!strategist");
        require(approvedStrategies[_token][_strategy] == true, "!approved");
        require(_strategy != address(0), "ADDRESS ERROR!");
        require(_token != address(0), "ADDRESS ERROR!");

        address _current = strategies[_token];
        if (_current != address(0)) {//knownsec// If there is an existing strategy contract, first withdraw all
            Strategy(_current).withdrawAll();
        }
        strategies[_token] = _strategy;
    }

    function earn(address _token, uint _amount) public {
        require(_token != address(0), "ADDRESS ERROR!");
        address _strategy = strategies[_token];
        address _want = Strategy(_strategy).want();
        if (_want != _token) {
            address converter = converters[_token][_want];
            IERC20(_token).safeTransfer(converter, _amount);
            _amount = Converter(converter).convert(_strategy);
            IERC20(_want).safeTransfer(_strategy, _amount);
        } else {
            IERC20(_token).safeTransfer(_strategy, _amount);
        }
        Strategy(_strategy).deposit();
    }

    function balanceOf(address _token) external view returns (uint) {
        require(_token != address(0), "ADDRESS ERROR!");
```

```
        return Strategy(strategies[_token]).balanceOf();
    }

    function withdrawAll(address _token) public {
        require(msg.sender == strategist || msg.sender == governance, "!strategist");
        require(_token != address(0), "ADDRESS ERROR!");
        Strategy(strategies[_token]).withdrawAll();
    }

    function inCaseTokensGetStuck(address _token, uint _amount) public {
        require(msg.sender == strategist || msg.sender == governance, "!governance");
        require(_token != address(0), "ADDRESS ERROR!");
        IERC20(_token).safeTransfer(msg.sender, _amount);
    }

    function inCaseStrategyTokenGetStuck(address _strategy, address _token) public {
        require(msg.sender == strategist || msg.sender == governance, "!governance");
        require(_strategy != address(0), "ADDRESS ERROR!");
        require(_token != address(0), "ADDRESS ERROR!");
        Strategy(_strategy).withdraw(_token);
    }

    function withdraw(address _token, uint _amount) public {
        require(msg.sender == vaults[_token], "!vault");
        require(_token != address(0), "ADDRESS ERROR!");
        Strategy(strategies[_token]).withdraw(_amount);
    }

    function rewards() public view returns (address) {
        return strategist;
    }
}
```

Recommendation： nothing.

3.2. Strategic contract【PASS】

Audit analysis: The strategy contract is implemented in the StrategyLP.sol contract file. The strategy contract based on the year protocol implements a general strategy contract. The input assets and output assets of the strategy are passed in by the constructor during deployment.

```
contract StrategyLP{
    using SafeERC20 for IERC20;
    using Address for address;
    using SafeMath for uint256;


    address public constant newb = 0x545f90dC35CA1e6129f1fEd354b3e2DF12034261;
    address public constant busd = 0xe9e7CEA3DedcA5984780Bafc599bD69ADd087D56;
    address public constant usdt = 0x55d398326f99059fF775485246999027B3197955;
    address public constant uniRouter = 0x10ED43C718714eb63d5aA57B78B54704E256024E;

    uint256 public strategistReward = 10;
    uint256 public restake = 90;
    uint256 public withdrawalFee = 500;
    uint256 public constant FEE_DENOMINATOR = 10000;

    address public out;

    address public pool;
    uint256 public pid;
    address public want;

    address public token0Address;
    address public token1Address;
```

```
    address public governance;
    address public controller;
    address public strategist;

    mapping(address => bool) public farmers;

    constructor(
        address _controller,
        address _pool,
        uint _pid,
        address _want,
        address _out,
        address _token0Address,
        address _token1Address
    ) {
        governance = msg.sender;
        strategist = 0x243508EeF0ADc429A520c4a24f7AfdcC532edBC7;
        controller = _controller;
        pool = _pool;
        pid = _pid;
        want = _want;
        out = _out;
        token0Address = _token0Address;
        token1Address = _token1Address;

        doApprove();
    }
    ......
    function harvest() public {

        require(!Address.isContract(msg.sender),"!contract");//knownsec// Verify non-contract
calls
```

```
        dRewards(pool).withdraw(pid,0);

        uint256 _2reward = IERC20(out).balanceOf(address(this)).mul(strategistReward).div(100);//knownsec// 10%
        uint256 _2want = IERC20(out).balanceOf(address(this)).mul(restake).div(100);//knownsec// 90%
        if(_2reward > 0)
        {//knownsec// out<>busd<>newb
            address[] memory path = new address[](2);
                path[0] = out;
                path[1] = busd;
                Uni(uniRouter).swapExactTokensForTokens(
                    _2reward,
                    uint256(0),
                    path,
                    address(this),
                    block.timestamp.add(1800)
                );
                IERC20(busd).safeApprove(uniRouter, 0);
                IERC20(busd).safeApprove(uniRouter, uint256(-1));
                path[0] = busd;
                path[1] = newb;
                Uni(uniRouter).swapExactTokensForTokens(
                    IERC20(busd).balanceOf(address(this)),
                    uint256(0),
                    path,
                    address(this),
                    block.timestamp.add(1800)
                );
            uint256 _2strategistReward = IERC20(newb).balanceOf(address(this));

            IERC20(newb).safeTransfer(strategist, _2strategistReward);
```

```
}
if(_2want > 0)
{//knownsec// out, half to token0 and half to token1
    if (out != token0Address) {
        // Swap half earned to token0
        if(token0Address == busd)
        {//knownsec// out<>busd
            address[] memory path = new address[](2);
            path[0] = out;
            path[1] = busd;
            Uni(uniRouter).swapExactTokensForTokens(
                    _2want.div(2),
                    uint256(0),
                    path,
                    address(this),
                    block.timestamp.add(1800)
            );
        }else if(token0Address == usdt)
        {//knownsec// out<>busd<>usdt
            address[] memory path = new address[](3);
            path[0] = out;
            path[1] = busd;
            path[2] = usdt;
            Uni(uniRouter).swapExactTokensForTokens(
                    _2want.div(2),
                    uint256(0),
                    path,
                    address(this),
                    block.timestamp.add(1800)
            );
        }else{//knownsec// out<>busd<>usdt<>token0
            address[] memory path = new address[](4);
            path[0] = out;
```

```
                path[1] = busd;
                path[2] = usdt;
                path[3] = token0Address;
                Uni(uniRouter).swapExactTokensForTokens(
                        _2want.div(2),
                        uint256(0),
                        path,
                        address(this),
                        block.timestamp.add(1800)
                );
            }
        }

    if (out != token1Address) {
        // Swap half earned to token1
            if(token1Address == busd)
            {//knownsec// out<>busd
                address[] memory path = new address[](2);
                path[0] = out;
                path[1] = busd;
                Uni(uniRouter).swapExactTokensForTokens(
                        _2want.div(2),
                        uint256(0),
                        path,
                        address(this),
                        block.timestamp.add(1800)
                );
            }else if(token1Address == usdt)
            {//knownsec// out<>busd<>usdt
                address[] memory path = new address[](3);
                path[0] = out;
                path[1] = busd;
                path[2] = usdt;
```

```
            Uni(uniRouter).swapExactTokensForTokens(
                    _2want.div(2),
                    uint256(0),
                    path,
                    address(this),
                    block.timestamp.add(1800)
            );
        }else{//knownsec// out<>busd<>usdt<>token1
            address[] memory path = new address[](4);
            path[0] = out;
            path[1] = busd;
            path[2] = usdt;
            path[3] = token1Address;
            Uni(uniRouter).swapExactTokensForTokens(
                    _2want.div(2),
                    uint256(0),
                    path,
                    address(this),
                    block.timestamp.add(1800)
            );
        }
    }

    uint256 token0Amt = IERC20(token0Address).balanceOf(address(this));
    uint256 token1Amt = IERC20(token1Address).balanceOf(address(this));
    if (token0Amt > 0 && token1Amt > 0) {//knownsec// Add token0-token1 liquidity
        IERC20(token0Address).safeApprove(uniRouter, 0);
        IERC20(token0Address).safeApprove(uniRouter, uint256(-1));
        IERC20(token1Address).safeApprove(uniRouter, 0);
        IERC20(token1Address).safeApprove(uniRouter, uint256(-1));
        Uni(uniRouter).addLiquidity(
            token0Address,
            token1Address,
```

```
                    token0Amt,
                    token1Amt,
                    0,
                    0,
                    address(this),
                    block.timestamp.add(1800)
                );
            }
        }

        _deposit();
    }
    ......
}
```

Recommendation： nothing.

3.3. Vault contract【PASS】

Audit analysis: The vault contract is implemented in the zVault.sol contract file, which implements a general vault contract. The tokens stored in the vault are passed in by the constructor when deployed.

```
contract zVault is ERC20, ERC20Detailed {
    using SafeERC20 for IERC20;
    using Address for address;
    using SafeMath for uint256;

    IERC20 public token;

    uint public min = 10000;
    uint public constant max = 10000;
```

```
    uint public earnLowerlimit; // The free funds in the pool will be automatically earned at this value

    address public governance;
    address public controller;

    constructor (address _token,uint _earnLowerlimit ,address _controller) public ERC20Detailed(
        string(abi.encodePacked("newb ", ERC20Detailed(_token).name())),
        string(abi.encodePacked("n", ERC20Detailed(_token).symbol())),
        ERC20Detailed(_token).decimals()
    ) {
        token = IERC20(_token);
        governance = msg.sender;
        controller = _controller;
        earnLowerlimit = _earnLowerlimit;
    }
    .....
    }
```

Recommendation： nothing.

4. Basic code vulnerability detection

4.1. Compiler version security【PASS】

Check whether a safe compiler version is used in the contract code implementation.

Audit result: After testing, the smart contract code has formulated the compiler version ^0.5.16、>=0.7.0、^0.7.3 within the major version, and there is no such security problem.

Recommendation： nothing.

4.2. Redundant code【PASS】

Check whether the contract code implementation contains redundant code.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.3. Use of safe arithmetic library【PASS】

Check whether the SafeMath safe arithmetic library is used in the contract code implementation.

Audit result: After testing, the SafeMath safe arithmetic library has been used in the smart contract code, and there is no such security problem.

Recommendation： nothing.

4.4. Not recommended encoding【PASS】

Check whether there is an encoding method that is not officially recommended or abandoned in the contract code implementation

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.5. Reasonable use of require/assert【PASS】

Check the rationality of the use of require and assert statements in the contract code implementation.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.6. Fallback function safety【PASS】

Check whether the fallback function is used correctly in the contract code implementation.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.7. tx.origin authentication【PASS】

tx.origin is a global variable of Solidity that traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authentication in a smart contract makes the contract vulnerable to attacks like phishing.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.8. Owner permission control【PASS】

Check whether the owner in the contract code implementation has excessive authority. For example, arbitrarily modify other account balances, etc.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.9. Gas consumption detection【PASS】

Check whether the consumption of gas exceeds the maximum block limit.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.10. call injection attack【PASS】

When the call function is called, strict permission control should be done, or the function called by the call should be written dead.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.11. Low-level function safety【PASS】

Check whether there are security vulnerabilities in the use of low-level functions (call/delegatecall) in the contract code implementation

The execution context of the call function is in the called contract; the execution context of the delegatecall function is in the contract that currently calls the function.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.12. Vulnerability of additional token issuance【PASS】

Check whether there is a function that may increase the total amount of tokens in the token contract after initializing the total amount of tokens.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.13. Access control defect detection【PASS】

Different functions in the contract should set reasonable permissions.

Check whether each function in the contract correctly uses keywords such as public and private for visibility modification, check whether the contract is correctly defined and use modifier to restrict access to key functions to avoid problems caused by unauthorized access.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.14. Numerical overflow detection【PASS】

The arithmetic problems in smart contracts refer to integer overflow and integer underflow.

Solidity can handle up to 256-bit numbers (2^256-1). If the maximum number increases by 1, it will overflow to 0. Similarly, when the number is an unsigned type, 0 minus 1 will underflow to get the maximum digital value.

Integer overflow and underflow are not a new type of vulnerability, but they are especially dangerous in smart contracts. Overflow conditions can lead to incorrect

results, especially if the possibility is not expected, which may affect the reliability and safety of the program.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.15. Arithmetic accuracy error【PASS】

As a programming language, Solidity has data structure design similar to ordinary programming languages, such as variables, constants, functions, arrays, functions, structures, etc. There is also a big difference between Solidity and ordinary programming languages-Solidity does not float Point type, and all the numerical calculation results of Solidity will only be integers, there will be no decimals, and it is not allowed to define decimal type data. Numerical calculations in the contract are indispensable, and the design of numerical calculations may cause relative errors. For example, the same level of calculations: 5/2*10=20, and 5*10/2=25, resulting in errors, which are larger in data The error will be larger and more obvious.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.16. Incorrect use of random numbers【PASS】

Smart contracts may need to use random numbers. Although the functions and variables provided by Solidity can access values that are obviously unpredictable, such as block.number and block.timestamp, they are usually more public than they appear or are affected by miners. These random numbers are predictable to a certain extent, so malicious users can usually copy it and rely on its unpredictability to attack the function.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.17. Unsafe interface usage【PASS】

Check whether unsafe interfaces are used in the contract code implementation.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.18. Variable coverage【PASS】

Check whether there are security issues caused by variable coverage in the contract code implementation.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.19. Uninitialized storage pointer【PASS】

In solidity, a special data structure is allowed to be a struct structure, and the local variables in the function are stored in storage or memory by default.

The existence of storage (memory) and memory (memory) are two different concepts. Solidity allows pointers to point to an uninitialized reference, while uninitialized local storage will cause variables to point to other storage variables, leading to variable coverage, or even more serious As a consequence, you should avoid initializing struct variables in functions during development.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.20. Return value call verification【PASS】

This problem mostly occurs in smart contracts related to currency transfer, so it is also called silent failed delivery or unchecked delivery.

In Solidity, there are transfer(), send(), call.value() and other currency transfer methods, which can all be used to send BNB to an address. The difference is: When the transfer fails, it will be thrown and the state will be rolled back; Only 2300gas will be passed for calling to prevent reentry attacks; false will be returned when send fails; only 2300gas will be passed for calling to prevent reentry attacks; false will be

returned when call.value fails to be sent; all available gas will be passed for calling (can be Limit by passing in gas_value parameters), which cannot effectively prevent reentry attacks.

If the return value of the above send and call.value transfer functions is not checked in the code, the contract will continue to execute the following code, which may lead to unexpected results due to BNB sending failure.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.21. Transaction order dependency【PASS】

Since miners always get gas fees through codes that represent externally owned addresses (EOA), users can specify higher fees for faster transactions. Since the Ethereum blockchain is public, everyone can see the content of other people's pending transactions. This means that if a user submits a valuable solution, a malicious user can steal the solution and copy its transaction at a higher fee to preempt the original solution.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.22. Timestamp dependency attack【PASS】

The timestamp of the data block usually uses the local time of the miner, and this time can fluctuate in the range of about 900 seconds. When other nodes accept a new block, it only needs to verify whether the timestamp is later than the previous block and The error with local time is within 900 seconds. A miner can profit from it by setting the timestamp of the block to satisfy the conditions that are beneficial to him as much as possible.

Check whether there are key functions that depend on the timestamp in the contract code implementation.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.23. Denial of service attack【PASS】

In the world of Ethereum, denial of service is fatal, and a smart contract that has suffered this type of attack may never be able to return to its normal working state. There may be many reasons for the denial of service of the smart contract, including malicious behavior as the transaction recipient, artificially increasing the gas required for computing functions to cause gas exhaustion, abusing access control to access the private component of the smart contract, using confusion and negligence, etc. Wait.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.24. Fake recharge vulnerability【PASS】

The transfer function of the token contract uses the if judgment method to check the balance of the transfer initiator (msg.sender). When balances[msg.sender] <value, enter the else logic part and return false, and finally no exception is thrown. We believe that only if/else this kind of gentle judgment method is an imprecise coding method in sensitive function scenarios such as transfer.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.25. Reentry attack detection【PASS】

The **call.value()** function in Solidity consumes all the gas it receives when it is used to send BNB. When the **call.value()** function to send BNB occurs before the actual reduction of the sender's account balance, There is a risk of reentry attacks.

Audit results： After auditing, the vulnerability does not exist in the smart contract code.

Recommendation： nothing.

4.26. Replay attack detection【PASS】

If the contract involves the need for entrusted management, attention should be paid to the non-reusability of verification to avoid replay attacks

In the asset management system, there are often cases of entrusted management. The principal assigns assets to the trustee for management, and the principal pays a certain fee to the trustee. This business scenario is also common in smart contracts.

Audit results： After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.27. Rearrangement attack detection【PASS】

A rearrangement attack refers to a miner or other party trying to "compete" with smart contract participants by inserting their own information into a list or mapping, so that the attacker has the opportunity to store their own information in the contract. in.

Audit results： After auditing, the vulnerability does not exist in the smart contract code.

Recommendation： nothing.

5. Appendix A: Vulnerability rating standard

Smart contract vulnerability rating standards	
Level	Level Description
High	Vulnerabilities that can directly cause the loss of token contracts or user funds, such as: value overflow loopholes that can cause the value of tokens to zero, fake recharge loopholes that can cause exchanges to lose tokens, and can cause contract accounts to lose BNB or tokens. Access loopholes, etc.; Vulnerabilities that can cause loss of ownership of token contracts, such as: access control defects of key functions, call injection leading to bypassing of access control of key functions, etc.; Vulnerabilities that can cause the token contract to not work properly, such as: denial of service vulnerability caused by sending BNB to malicious addresses, and denial of service vulnerability caused by exhaustion of gas.
Medium	High-risk vulnerabilities that require specific addresses to trigger, such as value overflow vulnerabilities that can be triggered by token contract owners; access control defects for non-critical functions, and logical design defects that cannot cause direct capital losses, etc.
Low	Vulnerabilities that are difficult to be triggered, vulnerabilities with limited damage after triggering, such as value overflow vulnerabilities that require a large amount of BNB or tokens to trigger, vulnerabilities where attackers cannot

	directly profit after triggering value overflow, and the transaction sequence triggered by specifying high gas depends on the risk Wait.

6. Appendix B：Introduction to auditing tools

7.1 Manticore

Manticore is a symbolic execution tool for analyzing binary files and smart contracts. Manticore includes a symbolic Ethereum Virtual Machine (EVM), an EVM disassembler/assembler and a convenient interface for automatic compilation and analysis of Solidity. It also integrates Ethersplay, Bit of Traits of Bits visual disassembler for EVM bytecode, used for visual analysis. Like binary files, Manticore provides a simple command line interface and a Python for analyzing EVM bytecode API.

7.2 Oyente

Oyente is a smart contract analysis tool. Oyente can be used to detect common bugs in smart contracts, such as reentrancy, transaction sequencing dependencies, etc. More convenient, Oyente's design is modular, so this allows advanced users to implement and Insert their own detection logic to check the custom attributes in their contract.

7.3 securify.sh

Securify can verify common security issues of Ethereum smart contracts, such as disordered transactions and lack of input verification. It analyzes all possible execution paths of the program while fully automated. In addition, Securify also has a

specific language for specifying vulnerabilities, which makes Securify can keep an eye on current security and other reliability issues at any time.

7.4 Echidna

Echidna is a Haskell library designed for fuzzing EVM code.

7.5 MAIAN

MAIAN is an automated tool for finding vulnerabilities in Ethereum smart contracts. Maian processes the bytecode of the contract and tries to establish a series of transactions to find and confirm the error.

7.6 ethersplay

ethersplay is an EVM disassembler, which contains relevant analysis tools.

7.7 ida-evm

ida-evm is an IDA processor module for the Ethereum Virtual Machine (EVM).

7.8 Remix-ide

ida-evm is an IDA processor module for the Ethereum Virtual Machine (EVM).

7.9 Knownsec Penetration Tester Special Toolkit

Pen-Tester tools collection is created by KnownSec team. It contains plenty of Pen-Testing tools such as automatic testing tool, scripting tool, Self-developed tools etc.

KNOWNSEC

Beijing KnownSec Information Technology Co., Ltd.

Advisory telephone	+86(10)400 060 9587
E-mail	sec@knownsec.com
Website	www.knownsec.com
Address	wangjing soho T2-B2509,Chaoyang District, Beijing