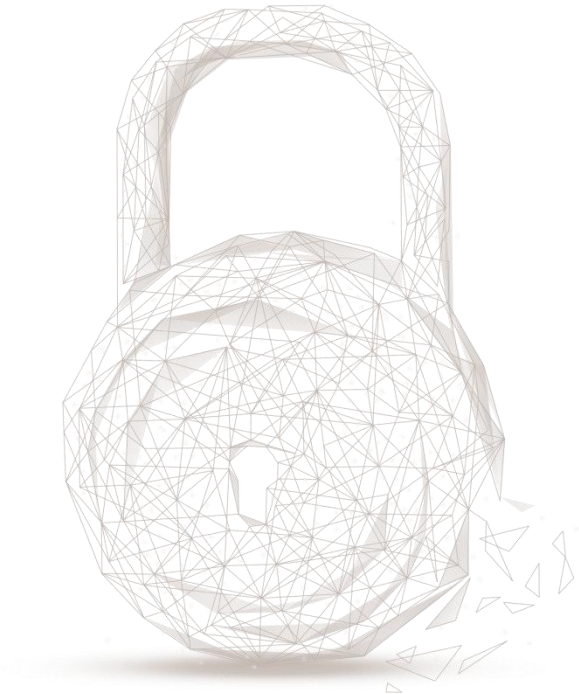
BEOSIN
Blockchain Security

Smart contract security audit report



Audit Number: 202110111802

Project Contract Name: OptionTrade

Project Deployment Platform: Binance Smart Chain

Project Contract Address: 0xd5d6e68f1e0c75c4ea7ece3317a865825189a7e6

Audit Start Date: 2021.09.10

Audit Completion Date: 2021.10.11

Audit Result: Pass

Audit Team: Beosin Technology Co. Ltd.

Audit Results Explained

Beosin Technology has used several methods including Formal Verification, Static Analysis, Typical Case Testing and Manual Review to audit three major aspects of OptionTrade smart contracts, including Coding Standards, Security, and Business Logic. **After auditing, the OptionTrade project was found to have 5 risk items: 3 Critical-risk, 2 Info. As of the completion of the audit, part of the risk items have been fixed or properly handled. The overall result of the OptionTrade smart contract is Pass.** The following is the detailed audit information for this project.

Index	Risk description	Risk level	Fix results
OptionTrade-1	The *withdraw* function has too much authority	Critical	Fixed
OptionTrade-2	The *ProfitDistribution*, *InsuredRefundand* and *StackWithdraw* function has too much authority	Critical	Ignored
OptionTrade-3	*sendETHDividends* function design flaws	Critical	Fixed
OptionTrade-4	currentMBNBPrice parameter redundancy	Info	Ignored
OptionTrade-5	*InStake* function design flaws	Info	Ignored

Table 1. Risk Statistics

Risk explained:

Item OptionTrade-2 is not fixed and may cause the loss of tokens staked by the user in the contract.

Risk descriptions and fix results explained

[OptionTrade-1 Critical] The *withdraw* function has too much authority

Description: The *withdraw* function can be used by the owner to withdraw the user's stake tokens.

```
function withdraw() public {
    require(msg.sender == _owner, "Can not send without owner");
    msg.sender.transfer(address(this).balance);
}
```

Figure 1 Source code screenshot of *withdraw* function (Unfixed)

Fix recommendations: Suggested deletion.

Fix results: The function has been deleted.

[OptionTrade-2 Critical] The *ProfitDistribution*, *InsuredRefundand* and *StackWithdraw* function has too much authority

Description: The *ProfitDistribution*, *InsuredRefundand* and *StackWithdraw* functions can all be called by the owner to extract tokens staked by the user in the contract.

```
function StackWithdraw(address[] memory user,uint[] memory amount) public
onlyOwner {
        for(uint i=0;i<user.length;i++){
            sendETHDividends(user[i],amount[i]);
        }
}

function ProfitDistribution(address[] memory user,uint[] memory amount)
public onlyOwner {
        for(uint i=0;i<user.length;i++){
            sendETHDividends(user[i],amount[i]);
        }
}

function InsuredRefund(address[] memory user,uint[] memory amount) public
onlyOwner {
        for(uint i=0;i<user.length;i++){
            sendETHDividends(user[i],amount[i]);
        }
}


 function sendETHDividends(address receiver,uint amount ) private {
        if (!address(uint160(receiver)).send(amount)) {
            return address(uint160(receiver)).transfer(address(this).
            balance);
        }
}
```

Figure 2 Screenshot of the related code (Unfixed)

Fix recommendations: Suggested deletion.

Fix results: Project Parties Description: This owner is called by other contracts.

```
function ProfitDistribution(address[] memory user,uint[] memory amount)
public onlyOwner {
        for(uint i=0;i<user.length;i++){
            payable(user[i]).transfer(amount[i]);
        }
}

function InsuredRefund(address[] memory user,uint[] memory amount) public
onlyOwner {
        for(uint i=0;i<user.length;i++){
            payable(user[i]).transfer(amount[i]);
        }
}
```

Figure 3 Screenshot of the related code (fixed)

[OptionTrade-3 Critical] *sendETHDividends* function design flaws

Description: The design logic of the token transfer is flawed, assuming the caller is the contract, rejecting the tokens sent by send in the fallback in the contract, and then being able to empty the pool of all tokens by transfer.

```
function sendETHDividends(address receiver,uint amount ) private {
        if (!address(uint160(receiver)).send(amount)) {
            return address(uint160(receiver)).transfer(address(this).
            balance);
        }
}
```

Figure 4 Source code screenshot of *sendETHDividends* function (Unfixed)

Fix recommendations: When a transfer is made, the balance in the contract is first recorded with a variable, and if the number of transfers is greater than the contract balance, Then the remaining tokens in the contract will be sent.

Fix results: fixed.

```
}
function sendBNBDividends(address payable receiver,uint amount) private {
    uint256 balance = address(this).balance;
    if(amount>balance){
        receiver.transfer(address(this).balance);
    }else
        receiver.transfer(amount);
}
```

Figure 5 Source code screenshot of *sendETHDividends* function (fixed)

[OptionTrade-4 Info] currentMBNBPrice parameter redundancy

Description: The currentMBNBPrice parameter is not used in the code.

```
contract MBNBPool is
    Ownable,
    BEP20("MBNB LP", "MBNB")
{
    using SafeMath for uint256;
    uint256 private minimumMBNBPrice = 0.001 ether;
    uint256 public currentMBNBPrice;

    /*
     * @nonce Provider burns OBNB and receives BNB from the pool
     * @param amount Amount of BNB to receive
     * @return burn Amount of tokens to be burnt
     */
```

Figure 6 Screenshot of the related code (Unfixed)

Fix recommendations: Suggested deletion.

Fix results: Ignored.

[OptionTrade-5 Info] *InStake* function design flaws.

Description: In the *InStake* function, when calculating the price, the division is done before the multiplication, which will result in the MBNB obtained by the user being small.

```
function InStake() external payable{
    uint256 MBNBPrice = 0;
    if(this.totalSupply() > 0 && (address(this).balance).mul(10**18) > 0){
        MBNBPrice = (((address(this).balance).sub(msg.value)).mul(10**18))/
        this.totalSupply();
    }
    uint256 payMBNB = 0;
    if(this.totalSupply() == 0 || MBNBPrice == 0){
        payMBNB = msg.value/minimumMBNBPrice;
    }else{
        payMBNB = msg.value/MBNBPrice;
    }
    if(payMBNB>0){
        payMBNB = payMBNB.mul(10**18);
    }
    _mint(msg.sender, payMBNB);
    // emit Withdraw(msg.sender, amount, burn);
}
```

Figure 7 Screenshot of the related code (Unfixed)

Fix recommendations: Multiplication before division is recommended.

Fix results: Ignored.

Other audit items explained

1. Other audit recommendations

- Users should note that the owner can call the *ProfitDistribution* and *InsuredRefundand* functions to withdraw the principal amount staked by the user.

Appendix 1 Description of Vulnerability Level

Vulnerability Level	Description	Example
Critical	Vulnerabilities that lead to the complete destruction of the project and cannot be recovered. It is strongly recommended to fix.	Malicious tampering of core contract privileges and theft of contract assets.
High	Vulnerabilities that lead to major abnormalities in the operation of the contract due to contract operation errors. It is strongly recommended to fix.	Unstandardized docking of the USDT interface, causing the user's assets to be unable to withdraw.
Medium	Vulnerabilities that cause the contract operation result to be inconsistent with the design but will not harm the core business. It is recommended to fix.	The rewards that users received do not match expectations.
Low	Vulnerabilities that have no impact on the operation of the contract, but there are potential security risks, which may affect other functions. The project party needs to confirm and determine whether the fix is needed according to the business scenario as appropriate.	Inaccurate annual interest rate data queries.
Info	There is no impact on the normal operation of the contract, but improvements are still recommended to comply with widely accepted common project specifications.	It is needed to trigger corresponding events after modifying the core configuration.

Appendix 2 Audit Categories and Details

No.	Categories	Subitems
1	Coding Conventions	Compiler Version Security
		Deprecated Items
		Redundant Code
		require/assert Usage
		Gas Consumption
2	General Vulnerability	Integer Overflow/Underflow
		Reentrancy
		Pseudo-random Number Generator (PRNG)
		Transaction-Ordering Dependence
		DoS (Denial of Service)
		Function Call Permissions
		call/delegatecall Security
		Returned Value Security
		tx.origin Usage
		Replay Attack
		Overriding Variables
3	Business Security	Business Logics
		Business Implementations

1. Coding Conventions

1.1. Compiler Version Security

The old version of the compiler may cause various known security issues. Developers are advised to specify the contract code to use the latest compiler version and eliminate the compiler alerts.

1.2. Deprecated Items

The Solidity smart contract development language is in rapid iteration. Some keywords have been deprecated by newer versions of the compiler, such as throw, years, etc. To eliminate the potential pitfalls they

may cause, contract developers should not use the keywords that have been deprecated by the current compiler version.

1.3. Redundant Code

Redundant code in smart contracts can reduce code readability and may require more gas consumption for contract deployment. It is recommended to eliminate redundant code.

1.4. SafeMath Features

Check whether the functions within the SafeMath library are correctly used in the contract to perform mathematical operations, or perform other overflow prevention checks.

1.5. require/assert Usage

Solidity uses state recovery exceptions to handle errors. This mechanism will undo all changes made to the state in the current call (and all its subcalls) and flag the errors to the caller. The functions assert and require can be used to check conditions and throw exceptions when the conditions are not met. The assert function can only be used to test for internal errors and check non-variables. The require function is used to confirm the validity of conditions, such as whether the input variables or contract state variables meet the conditions, or to verify the return value of external contract calls.

1.6. Gas Consumption

The smart contract virtual machine needs gas to execute the contract code. When the gas is insufficient, the code execution will throw an out of gas exception and cancel all state changes. Contract developers are required to control the gas consumption of the code to avoid function execution failures due to insufficient gas.

1.7. Visibility Specifiers

Check whether the visibility conforms to design requirement.

1.8. Fallback Usage

Check whether the Fallback function has been used correctly in the current contract.

2. General Vulnerability

2.1. Integer overflow

Integer overflow is a security problem in many languages, and they are especially dangerous in smart contracts. Solidity can handle up to 256-bit numbers (2**256-1). If the maximum number is increased by 1, it will overflow to 0. Similarly, when the number is a uint type, 0 minus 1 will underflow to get the maximum number value. Overflow conditions can lead to incorrect results, especially if its possible results are not expected, which may affect the reliability and safety of the program. For the compiler version after Solidity 0.8.0, smart contracts will perform overflow checking on mathematical operations by default. In the previous compiler versions, developers need to add their own overflow checking code, and SafeMath library is recommended to use.

2.2. Reentrancy

The reentrancy vulnerability is the most typical Ethereum smart contract vulnerability, which has caused the DAO to be attacked. The risk of reentry attack exists when there is an error in the logical order of calling the call.value() function to send assets.

2.3 Pseudo-random Number Generator (PRNG)

Random numbers may be used in smart contracts. In solidity, it is common to use block information as a random factor to generate, but such use is insecure. Block information can be controlled by miners or obtained by attackers during transactions, and such random numbers are to some extent predictable or collidable.

2.4. Transaction-Ordering Dependence

In the process of transaction packing and execution, when faced with transactions of the same difficulty, miners tend to choose the one with higher gas cost to be packed first, so users can specify a higher gas cost to have their transactions packed and executed first.

2.5. DoS(Denial of Service)

DoS, or Denial of Service, can prevent the target from providing normal services. Due to the immutability of smart contracts, this type of attack can make it impossible to ever restore the contract to its normal working state. There are various reasons for the denial of service of a smart contract, including malicious revert when acting as the recipient of a transaction, gas exhaustion caused by code design flaws, etc.

2.6. Function Call Permissions

If smart contracts have high-privilege functions, such as coin minting, self-destruction, change owner, etc., permission restrictions on function calls are required to avoid security problems caused by permission leakage.

2.7. call/delegatecall Security

Solidity provides the call/delegatecall function for function calls, which can cause call injection vulnerability if not used properly. For example, the parameters of the call, if controllable, can control this contract to perform unauthorized operations or call dangerous functions of other contracts.

2.8. Returned Value Security

In Solidity, there are transfer(), send(), call.value() and other methods. The transaction will be rolled back if the transfer fails, while send and call.value will return false if the transfer fails. If the return is not correctly judged, the unanticipated logic may be executed. In addition, in the implementation of the transfer/transferFrom function of the token contract, it is also necessary to avoid the transfer failure and return false, so as not to create fake recharge loopholes.

2.9. tx.origin Usage

The tx.origin represents the address of the initial creator of the transaction. If tx.origin is used for permission judgment, errors may occur; in addition, if the contract needs to determine whether the caller is the contract address, then tx.origin should be used instead of extcodesize.

2.10. Replay Attack

A replay attack means that if two contracts use the same code implementation, and the identity authentication is in the transmission of parameters, the transaction information can be replayed to the other contract to execute the transaction when the user executes a transaction to one contract.

2.11. Overriding Variables

There are complex variable types in Solidity, such as structures, dynamic arrays, etc. When using a lower version of the compiler, improperly assigning values to it may result in overwriting the values of existing state variables, causing logical exceptions during contract execution.

Appendix 3 Disclaimer

This report is made in response to the project code. No description, expression or wording in this report shall be construed as an endorsement, affirmation or confirmation of the project. This audit is only applied to the type of auditing specified in this report and the scope of given in the results table. Other unknown security vulnerabilities are beyond auditing responsibility. Beosin Technology only issues this report based on the attacks or vulnerabilities that already existed or occurred before the issuance of this report. For the emergence of new attacks or vulnerabilities that exist or occur in the future, Beosin Technology lacks the capability to judge its possible impact on the security status of smart contracts, thus taking no responsibility for them. The security audit analysis and other contents of this report are based solely on the documents and materials that the contract provider has provided to Beosin Technology before the issuance of this report, and the contract provider warrants that there are no missing, tampered, deleted; if the documents and materials provided by the contract provider are missing, tampered, deleted, concealed or reflected in a situation that is inconsistent with the actual situation, or if the documents and materials provided are changed after the issuance of this report, Beosin Technology assumes no responsibility for the resulting loss or adverse effects. The audit report issued by Beosin Technology is based on the documents and materials provided by the contract provider, and relies on the technology currently possessed by Beosin. Due to the technical limitations of any organization, this report conducted by Beosin still has the possibility that the entire risk cannot be completely detected. Beosin disclaims any liability for the resulting losses.

The final interpretation of this statement belongs to Beosin Technology.

BEOSIN

Blockchain Security

Official Website

https://lianantech.com

E-mail

market@lianantech.com

Twitter

https://twitter.com/Beosin_com