SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Morbex

Date: July 26th, 2021

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

This document may contain confidential information about IT systems and the intellectual property of the Customer as well as information about potential vulnerabilities and methods of their exploitation.

The report containing confidential information can be used internally by the Customer, or it can be disclosed publicly after all vulnerabilities are fixed - upon a decision of the Customer.

Document

Name	Smart Contract Code Review and Security Analysis Report for Morbex - Initial audit
Approved by	Andrew Matiukhin | CTO Hacken OU
Type	BEP20 Token with minting by votes
Platform	Binance smart chain / Solidity
Methods	Architecture Review, Functional Testing, Computer-Aided Verification, Manual Review
Deployed in mainnet	https://bscscan.com/address/0x0352b52f4ddea5a4a25173796adf8a00de1dc5bd
Timeline	21 July 2021 – 24 July 2021
Changelog	24 July 2021 – Initial audit

Table of contents

Introduction

Hacken OÜ (Consultant) was contracted by Morbex (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contract and its code review conducted between July 21st, 2021 - July 24th, 2021.

Scope

The scope of the project is the smart contract deployed in the binance smart chain mainnet:

```
https://bscscan.com/address/0x0352b52f4ddea5a4a25173796adf8a00de1dc5bd
```

We have scanned these smart contracts for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that are considered:

Category	Check Item
Code review	▪ Reentrancy ▪ Ownership Takeover ▪ Timestamp Dependence ▪ Gas Limit and Loops ▪ DoS with (Unexpected) Throw ▪ DoS with Block Gas Limit ▪ Transaction-Ordering Dependence ▪ Style guide violation ▪ Costly Loop ▪ ERC20 API violation ▪ Unchecked external call ▪ Unchecked math ▪ Unsafe type inference ▪ Implicit visibility level ▪ Deployment Consistency ▪ Repository Consistency ▪ Data Consistency

Functional review	▪ Business Logics Review ▪ Functionality Checks ▪ Access Control & Authorization ▪ Escrow manipulation ▪ Token Supply manipulation ▪ Asset’s integrity ▪ User Balances manipulation ▪ Kill-Switch Mechanism ▪ Operation Trails & Event Generation

Executive Summary

According to the assessment, the Customer's smart contract is secured but has costly loops and some code styling issues.

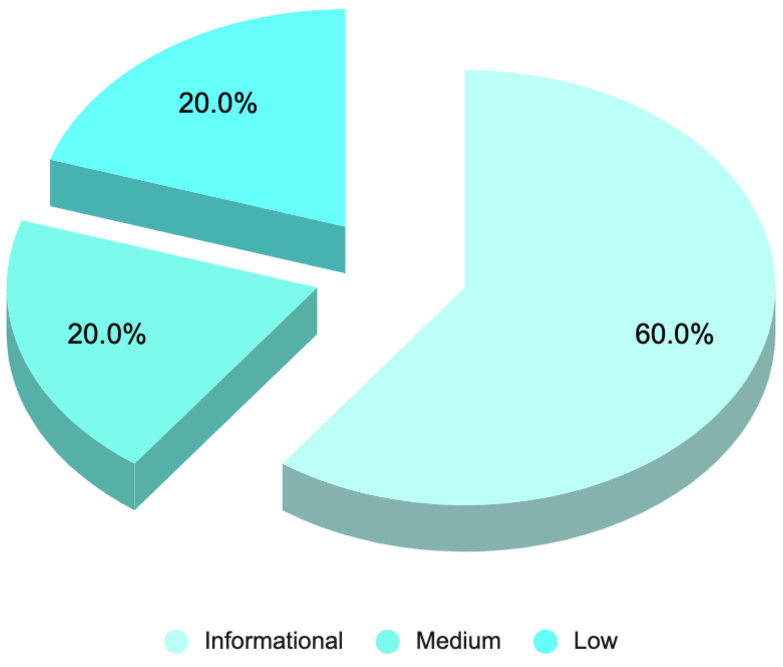
Insecure **Poor secured** **Secured** **Well-secured**

You are here

Our team performed an analysis of code functionality, manual audit, and automated checks with Mythril and Slither. All issues found during automated analysis were manually reviewed, and important vulnerabilities are presented in the Audit overview section. All found issues can be found in the Audit overview section.

Security engineers found **1** medium, **1** low and **3** informational issues during the first review.

Graph 1. The distribution of vulnerabilities after the first review.



Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to assets loss or data manipulations.
High	High-level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution, e.g., public access to crucial functions
Medium	Medium-level vulnerabilities are important to fix; however, they can't lead to assets loss or data manipulations.
Low	Low-level vulnerabilities are mostly related to outdated, unused, etc. code snippets that can't have a significant impact on execution
Lowest / Code Style / Best Practice	Lowest-level vulnerabilities, code style violations, and info statements can't affect smart contract execution and can be ignored.

Audit overview

■■■■ Critical

No Critical severity issues were found.

■■■ High

No High severity issues were found.

■■ Medium

1. **Vulnerability**: No return statement.

The **mint** function declared to return a **boolean** value but doesn't have a **return** statement in the body. That means the function will always return **false** which could be wrongly interpreted by the caller.

Lines: #310-322

```
function mint(address account) public returns (bool) {
   require(pendingMintAmount > 0, "there is no pending mint amount");
   require(getMintable(), "the vote count of validator members should be
greater than 10");
   super.mint(account, pendingMintAmount);
   for(uint i =0 ;i < VALIDATOR_NUMBERS; i++ )
   {
       if(enableMint[i])
       {
           enableMint[i] = false ;
       }
   }
   pendingMintAmount = 0;
}
```

■ Low

1. **Vulnerability**: Costly loops

Instead of building logic on loops, which is **costly** in the mean of gas, it's better to design the logic on the state and math.
For example, instead of looping through all validators to find a number of approves it's better to just keep this number in the state variable and update on the voting and minting request.
The same works for other places, so there's no need in loops at all.

■ Lowest / Code style / Best Practice

1. **Vulnerability:** Code layout.

   Solidity declares the code layout recommendations that should be followed. Such recommendations include:
   - Indentation
   - Blank Lines
   - Maximum Line Length
   - Order of Functions
   - Whitespace in Expressions
   - Control Structures
   - Function Declaration.

   Recommendation: Please follow code layout recommendations.

2. **Vulnerability:** Too many digits.

   Literals with many digits are difficult to read and review.

   Recommendation: Please use ether units suffixes and scientific notation. Ex.: **10e6 ether**

   Lines: #229

```
super.mint(_msgSender(), 10000000 * 10 ** 18);
```

3. **Vulnerability:** Public function that could be declared external.

   public functions that are never called by the contract should be declared **external** to save gas.

   Lines: #75

```
function totalSupply() public view override returns (uint256) {
```

   Lines: #78

```
function balanceOf(address account) public view override returns
(uint256) {
```

   Lines: #81

```
function transfer(address recipient, uint256 amount) public override
returns (bool) {
```

   Lines: #85

```
function allowance(address owner, address spender) public view override
returns (uint256) {
```

Lines: #88

```
function approve(address spender, uint256 amount) public override
returns (bool) {
```

Lines: #92

```
function transferFrom(address sender, address recipient, uint256
amount) public override returns (bool) {
```

Lines: #97

```
function increaseAllowance(address spender, uint256 addedValue) public
returns (bool) {
```

Lines: #101

```
function decreaseAllowance(address spender, uint256 subtractedValue)
public returns (bool) {
```

Lines: #145

```
function name() public view returns (string memory) {
```

Lines: #148

```
function symbol() public view returns (string memory) {
```

Lines: #151

```
function decimals() public view returns (uint8) {
```

Lines: #189

```
function addMinter(address account) public onlyMinter {
```

Lines: #192

```
function renounceMinter() public {
```

Lines: #213

```
function burn(uint256 amount) public {
```

Lines: #216

```
function burnFrom(address account, uint256 amount) public {
```

Lines: #235

```
function mintRequest(uint256 _amount) onlyMinter public {
```

Lines: #258

```
function getValidatorAddress(uint256 _index) public view
validIndex(_index) returns (address)
```

Lines: #262

```
function getValidatorIndex(address _account) public view
returns(uint256)
```

Lines: #273

```
function setMintEnable(uint256 _index, bool _mintEnable) public
validIndex(_index) validValidatorAddress(_index) {
```

Lines: #276

```
function transactValidatorRole(uint256 _index, address _account) public
validIndex(_index) validValidatorAddress(_index)
```

Lines: #298

```
function getMintEnableCount() public view returns (uint256)
```

Lines: #310

```
function mint(address account) public returns (bool) {
```

Conclusion

Smart contracts within the scope were manually reviewed and analyzed with static analysis tools.

The audit report contains all found security vulnerabilities and other issues in the reviewed code.

Security engineers found **1** medium, **1** low and **3** informational issues during the first review.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed in accordance with the best industry practices at the date of this report, in relation to cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only - we recommend proceeding with several independent audits and a public bug bounty program to ensure security of smart contracts.

Technical Disclaimer

Smart contracts are deployed and executed on the blockchain platform. The platform, its programming language, and other software related to the smart contract can have its vulnerabilities that can lead to hacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.