北京链安
Chains Guard Technology

SafeHaven Smart Contract Audit Report

Beijing Chains Guard Technology

■ Documentation

File name	SafeHaven Smart Contract Audit Report		
Serial number	CG-YMSJ-202008021		
Confidentiality	Business secret	Version	V1.1
Author	ChainsGuard Security Center	Date	2020-08-25

■ Scope of application

This security assessment was authorized by the authorized party, and The Beijing ChainsGuard Network Technology Co., Ltd. (hereinafter referred to as "ChainsGuard") conducted an in-depth evaluation of the security risks of the SafeHaven Smart Contract. security assessment and make recommendations to the reinforcement of the security situation in the main network is limited to Beijing ChainsGuard security and authorized party insiders circulated.

■ Version change record

Date	Version	Description	Modify by
2020-08-25	V1.1	Document creation	ChainsGuard Security Center

Catalogue

Disclaimer . . . 1

1 Introduction . . . 2

1.1 Overview . . . 2

1.2 Audit Time . . . 2

1.3 Audit Unit . . . 2

1.4 Audit Object . . . 3

2 Security Audit Summary . . . 3

2.1 Vulnerability Statistics . . . 3

3 Checklist . . . 4

3.1 Reentrancy Attack . . . 4

3.2 Permission Control . . . 5

3.3 Overflow/Underflow . . . 6

3.4 Call Injection . . . 6

3.5 Race Conditions . . . 7

3.6 Design Flaw . . . 7

3.7 DDoS Attack . . . 7

3.8 Pseudo-random Number . . . 8

3.9 Front-running . . . 8

3.10 Short Address Attack . . . 8

3.11 Data Privacy . . . 9

3.12 Malicious backdoor 9

3.13 Code Optimization 9

4 Summary of Security Audit 10

5 Optimization suggestions 10

5.1 Controller private-key protection 10

6 Smart Contract UML 11

Appendix A. Explanation of Security Risk Status Levels 12

Disclaimer

The audit report is a technical security audit for the authorized party. The purpose of this audit is to provide the authorized party with a reference basis for conducting its business security assessment and optimization, The regulatory regime of the business model, or any other statement about the applicability of the application, as well as a statement or warranty that the application is in error-free behavior. This report cannot be used as a proof that these tested systems and codes are absolutely secure and there are no other security risks.

The audit report only covers the code, installation packages and other materials provided by the authorized party, and its conclusion is only applicable to the corresponding version of the application. Once the relevant code, configuration, and operating environment change, the corresponding conclusion will no longer be applicable.

1 Introduction

1.1 Overview

This document includes the results of the audit performed by the Chains Guard Team on the SafeHaven project, at the request of the SafeHaven tea m. The goal of this audit is to review the smart contract code solidity impl ementation, study potential security vulnerabilities, its general designand arc hitecture, and uncover bugs that could compromise the software in product ion.

1.2 Audit Time

Evaluation test time	
Start time	2020-08-25
End time	2020-08-25

1.3 Audit Unit

Company Name	The Beijing ChainsGuard Network Technology Co., Ltd.
Web Site	https://www.chainsguard.com/

1.4 Audit Object

Project Name：SafeHaven

Github Link：https://github.com/Safehaven-io/Contracts

Git Branch：master

Commit Hash：51f1966eb447be8ec8c0532f8febe0e61f3ee101

Review Files:

```
├── control
|   ├── Controlled.sol
|   └── TokenController.sol
├── factory
|   └── MiniMeTokenFactory.sol
├── ownership
|   └── Owned.sol
└── token
    ├── ApproveAndCallFallback.sol
    ├── ERC20.sol
    ├── MiniMeToken.sol
    └── SafeHavenToken.sol
```

2 Security Audit Summary

2.1 Vulnerability Statistics

Vulnerabilities	High risk	Medium risk	Low risk
0	0	0	1

[Note] A brief description of the hazard classification method is as follows

High: It directly causes the system to be controlled or the data to be destroyed. Once it occurs, it is a serious security event.

Medium: It may lead to the leakage of important information or may cause the system to be controlled.

Low: Non-critical information leaks or minor security issues generally do not lead to serious security incidents.

3 Checklist

For smart contract source code audit, we focus on the detection of the following security points:

3.1 Reentrancy Attack

One of the main dangers of invoking external contracts is that they can take over the control process. In a reentrancy attack (also called a recursive call attack), a malicious contract will call back the calling contract before t he first call to the function is completed. This may cause different calls of t he function to interact in an undesirable way

Test result: **Pass**

Security reminder: It is recommended to use ReentrancyGuard technolog y to protect the contract from reentrancy attacks when calling external unt rusted contracts.

ReentrancyGuard: https://github.com/OpenZeppelin/openzeppelin-contract s/blob/master/contracts/utils/ReentrancyGuard.sol

3.2 Permission Control

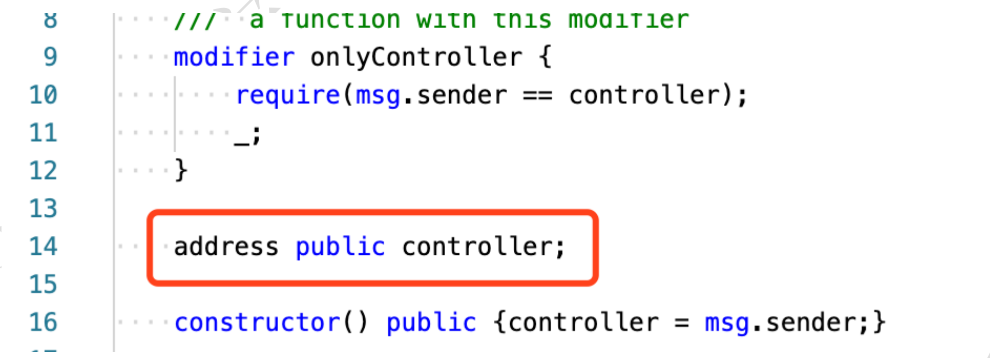
Check whether the functions in the contract use public, private and othe r keywords correctly for visibility modification. Check whether the contract i s correctly defined and use the modifier to restrict access to key functions to avoid unauthorized access.

Test result: **Low risk (not found exploit)**

Risk Detailes: The controller (state variable) of the contract uses public vi sibility modification, which will cause the variable to leak into the inheritanc e system of this contract. If the attacker can intervene in the execution con text of the sub-contract, he can tamper with the value of this field to achie ve control The purpose of the contract.

Fix suggestion: Change public visibility to private visibility to make it visi ble only in this contract.

Note: The owner (state variable) of the unused Owned contract also has the same problem.



```
    /// a function with this modifier
    modifier onlyController {
        require(msg.sender == controller);
        _;
    }

    address public controller;

    constructor() public {controller = msg.sender;}
```

3.3 Overflow/Underflow

The numerical processing in the smart contract needs to strictly check the arithmetic overflow/underflow problem, the conventional addition and subtraction arithmetic processing is easy to cause integer overflow or underflow, especially when processing the account amount of similar tokens, it is necessary to strictly judge the size of the account amount. Normally numerical calculation is recommended to use Zeppelin open source module SafeMath for processing.

Test result: **pass**

3.4 Call Injection

When using a low-level interface call() for contract interaction in a smart contract, the interface is improperly used. An attacker can control the parameters, method selectors, or execution byte contents of the call interface to perform dangerous operations such as unauthorized transfer, mint, and token burn.

Test result: **pass**

3.5 Race Conditions

Every transaction and smart contract calls requires miner to mine to con firm. Sharing a state between different functions may cause different proces sing results due to changes in the execution order.

Test result: **pass**

3.6 Design Flaw

Unreasonable code logic processing or wrong coding will lead to abnor mal contract execution, functional or security does not meet expectations, e tc. Before the contract is officially deployed, the entire code needs to be c hecked for integrity and multi-faceted to ensure that the contract code logi c is correct.

Test result: **pass**

3.7 DDoS Attack

"Unrecoverable malicious operations or controllable unlimited resource c onsumption" can all be called denial of service in smart contracts. An attac ker may call a contract function by constructing malicious parameters, whic h may lead to logical processing problems that cannot be recovered by the deployed contract, and the service is denied The smart contract usually m akes the entire code logic unable to continue execution and "stop service".

Test result: **pass**

3.8 Pseudo-random Number

The values such as block.timestamp, block.number, etc. in the smart cont ract are predictable. When the contract involves random number generatio n, do not use easily accessible variables or parameters as seeds to generate random numbers. A better method is to use The external service Oraclize performs random number generation and processing.

Test result: **pass**

3.9 Front-running

Every transaction and smart contract calls requires miners to confirm the mining. During this time period, it is extremely easy for an attacker to mal iciously read the transaction or call details, and then use the high Gas min er incentive mechanism to give priority to their transactions or calls. form F ront-Running.

Test result: **pass**

3.10 Short Address Attack

When the virtual machine parses the smart contract bytes, it will automa tically filling the number of parameter bytes. The attacker can construct mal formed data to make smart contract calls. Automatic byte filling will cause unintended parameter values to change.

Test result: **pass**

3.11 Data Privacy

The smart contract code needs to encrypt and protect the stored user's private information. Anyone can obtain the relevant information stored in th e contract by querying the data on the chain. If the private information is not encrypted or the encryption is not strict, it is easy to cause privacy lea kage.

Test result: **pass**

3.12 Malicious backdoor

In the blockchain ecosystem, some project parties themselves do not ha ve the ability to develop smart contracts. They usually choose some smart contract automation generation tools for one-click generation and automati c deployment on the blockchain. This opens up opportunities for hackers t o insert backdoor codes into smart contracts. Once the backdoor code is in sert into the smart contract, the hacker can use its backdoor to manipulate the contract arbitrarily, which will cause fatal harm to the project party.

Test result: **pass**

3.13 Code Optimization

Each step of smart contract code execution needs to spend correspondin g gas. Non-standard code writing will cause unnecessary gas waste. Code l ogic optimization can help the smart contract logic to be clearer, the execu tion efficiency is higher, and the Gas costs less.

Test result: pass

4 Summary of Security Audit

This contract, code style is good, no exploitable security issues found. The overall estimated safety status is **warning status.**

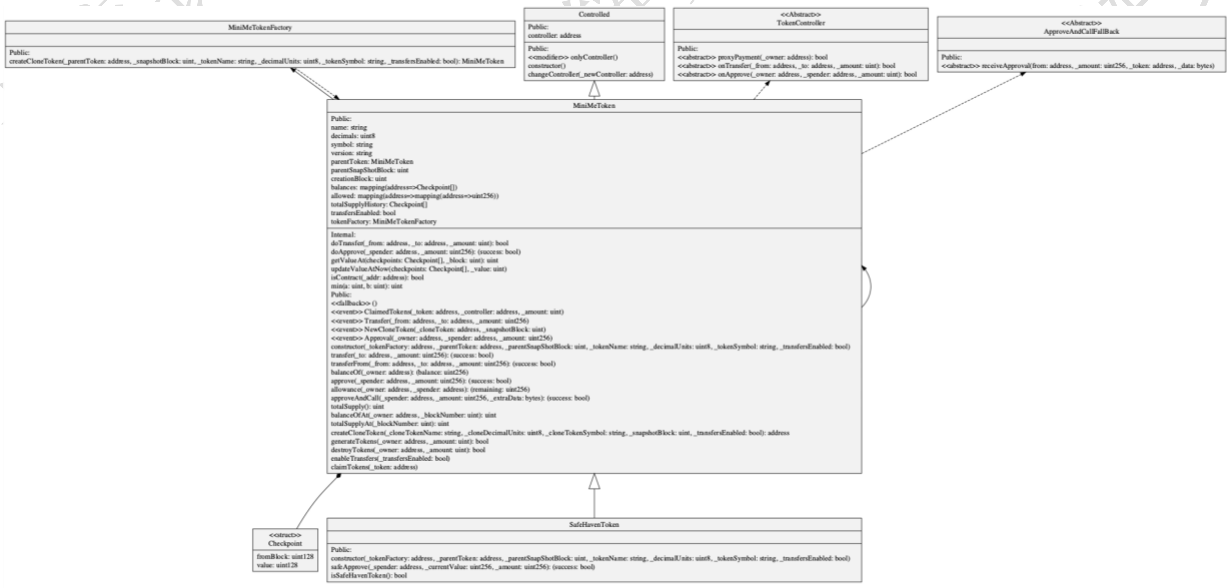
The intelligent contract audit results only provide the actual basis for the authorizer to formulate corresponding security measures and solutions.

5 Optimization suggestions

5.1 Controller private-key protection

The controller of this token contract has too much authority. Please protect the private key of the controller to prevent the token contract from being maliciously damaged due to the leakage of the private key.

6 Smart Contract UML


MiniMeTokenFactory
Public:
createCloneToken(_parentToken: address, _snapshotBlock: uint, _tokenName: string, _decimalUnits: uint8, _tokenSymbol: string, _transfersEnabled: bool): MiniMeToken
Controlled
Public:
controller: address
Public:
<<modifier>> onlyController()
constructor()
changeController(_newController: address)
<<Abstract>>
TokenController
Public:
<<abstract>> proxyPayment(_owner: address): bool
<<abstract>> onTransfer(_from: address, _to: address, _amount: uint): bool
<<abstract>> onApprove(_owner: address, _spender: address, _amount: uint): bool
<<Abstract>>
ApproveAndCallFallBack
Public:
<<abstract>> receiveApproval(from: address, _amount: uint256, _token: address, _data: bytes)
MiniMeToken
Public:
name: string
decimals: uint8
symbol: string
version: string
parentToken: MiniMeToken
parentSnapShotBlock: uint
creationBlock: uint
balances: mapping(address=>Checkpoint[])
allowed: mapping(address=>mapping(address=>uint256))
totalSupplyHistory: Checkpoint[]
transfersEnabled: bool
tokenFactory: MiniMeTokenFactory
Internal:
doTransfer(_from: address, _to: address, _amount: uint): bool
doApprove(_spender: address, _amount: uint256): (success: bool)
getValueAt(checkpoints: Checkpoint[], _block: uint): uint
updateValueAtNow(checkpoints: Checkpoint[], _value: uint)
isContract(_addr: address): bool
min(a: uint, b: uint): uint
Public:
<<fallback>> ()
<<event>> ClaimedTokens(_token: address, _controller: address, _amount: uint)
<<event>> Transfer(_from: address, _to: address, _amount: uint256)
<<event>> NewCloneToken(_cloneToken: address, _snapshotBlock: uint)
<<event>> Approval(_owner: address, _spender: address, _amount: uint256)
constructor(_tokenFactory: address, _parentToken: address, _parentSnapShotBlock: uint, _tokenName: string, _decimalUnits: uint8, _tokenSymbol: string, _transfersEnabled: bool)
transfer(_to: address, _amount: uint256): (success: bool)
transferFrom(_from: address, _to: address, _amount: uint256): (success: bool)
balanceOf(_owner: address): (balance: uint256)
approve(_spender: address, _amount: uint256): (success: bool)
allowance(_owner: address, _spender: address): (remaining: uint256)
approveAndCall(_spender: address, _amount: uint256, _extraData: bytes): (success: bool)
totalSupply(): uint
balanceOfAt(_owner: address, _blockNumber: uint): uint
totalSupplyAt(_blockNumber: uint): uint
createCloneToken(_cloneTokenName: string, _cloneDecimalUnits: uint8, _cloneTokenSymbol: string, _snapshotBlock: uint, _transfersEnabled: bool): address
generateTokens(_owner: address, _amount: uint): bool
destroyTokens(_owner: address, _amount: uint): bool
enableTransfers(_transfersEnabled: bool)
claimTokens(_token: address)
<<struct>>
Checkpoint
fromBlock: uint128
value: uint128
SafeHavenToken
Public:
constructor(_tokenFactory: address, _parentToken: address, _parentSnapShotBlock: uint, _tokenName: string, _decimalUnits: uint8, _tokenSymbol: string, _transfersEnabled: bool)
safeApprove(_spender: address, _currentValue: uint256, _amount: uint256): (success: bool)
isSafeHavenToken(): bool

Appendix A. Explanation of Security Risk Status Levels

Security risk status statement	
1	**good status** The contract is in good running condition, and there are no or only sporadic low-risk security problems. At this time, as long as the existing security policy is maintained, the safety level requirements of the system can be met.
2	**warning status** There are some loopholes or security risks in the smart contract, which have not been used on a large scale. At this time, targeted reinforcement or improvement should be carried out according to the problems found in the evaluation, and then redeployment.
3	**serious status** Smart contract has been widely used. Serious loopholes or security problems that may seriously threaten the normal operation of the contract are found in the intelligent contract. At this time, measures should be taken immediately to redeploy the strengthened intelligent contract.
4	**emergency status** The tokens related to the intelligent contract have been opened for trading. Serious loopholes or security problems that may seriously threaten the normal operation of the contract have been found in the intelligent contract, which may cause serious damage to economic interests. At this point, should immediately stop the contract related token trading, immediately take measures to redeploy the strengthened intelligent contract.