KNOWNSEC

Smart Contract Audit Report

Safe State

LOW RISK

★★★★★

Version description

Revised man	Revised content	Revised time	version number	Reviewer
	Document creation and editing	2020/9/18	V1.0	

Document information

Name	Document version number	Number	Privacy level
NTON Smart Contract Audit Report	V1.0	【NTON-DMSJ-20200918】	Open project team

Copyright statement

目录

1. Review

The effective testing time of this report is from September 17,2020 to September 18,2020. During this period, the Knownsec engineers audited the safety and regulatory aspects of NTON smart contract code.

In this test, engineers comprehensively analyzed common vulnerabilities of smart contracts (Chapter 3) and It was discovered that the order of transactions depends on the risk, which is common in token contracts, and the point of increasing tokens was also discovered; so it's evaluated as Low-risk.

The result of the safety auditing: PASS

Since the test process is carried out in a non-production environment, all the codes are the latest backups. We communicates with the relevant interface personnel, and the relevant test operations are performed under the controllable operation risk to avoid the risks during the test..

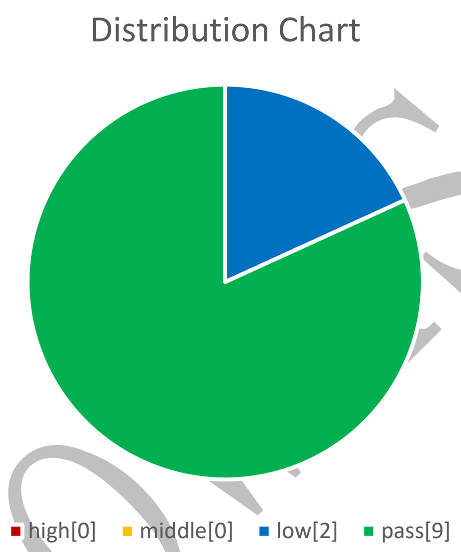
Target information for this test:

Module name	
Token name	NTON
Code type	Token code
Contract address	0xcfb152e5b93fc2c9906d4ff41fc8407dfa5e8851
chained address	https://etherscan.io/address/0xcfb152e5b93fc2c9906d4ff41fc8407dfa5e8851
Code language	solidity

2. Analysis of code vulnerability

2.1. DISTRIBUTION OF VULNERABILITY LEVELS

Vulnerability statistics			
high	Middle	low	pass
0	0	2	9



2.2. AUDIT RESULT SUMMARY

Result			
Test project	Test content	status	description
Smart Contract	Reentrancy	Pass	Check the call.value() function for security
	Arithmetic Issues	Pass	Check add and sub functions
	Access Control	Pass	Check the operation access control
	Unchecked Return Values For Low Level Calls	Pass	Check the currency conversion method.
	Bad Randomness	Pass	Check the unified content filter
	Transaction ordering dependence	Low	Check the transaction ordering dependence
	Denial of service attack detection	Pass	Check whether the code has a resource abuse problem when using a resource
	Logic design Flaw	Pass	Examine the security issues associated with business design in intelligent contract codes.
	USDT Fake Deposit Issue	Pass	Check for the existence of USDT Fake Deposit Issue
	Adding tokens	Low	It is detected whether there is a function in the token contract that may increase the total amounts of tokens
	Freezing accounts bypassed	Pass	It is detected whether there is an unverified token source account, an originating account, and whether the target account is frozen.

3. Result analysis

3.1. REENTRANCY【PASS】

The Reentrancy attack, probably the most famous Ethereum vulnerability，led to a hard fork of Ethereum.

When the low level call() function sends ether to the msg.sender address, it becomes vulnerable; if the address is a smart contract, the payment will trigger its fallback function with what's left of the transaction gas

Test results： There is no relevant call external contract call in the smart contract code after detected.

Safety advice： None。

3.2. ARITHMETIC ISSUES【PASS】

Also known as integer overflow and integer underflow. Solidity can handle up to 256 digits (2^256-1), The largest number increases by 1 will overflow to 0. Similarly, when the number is an unsigned type, 0 minus 1 will underflow to get the maximum numeric value.

Integer overflows and underflows are not a new class of vulnerability, but they are especially dangerous in smart contracts. Overflow can lead to incorrect results, especially if the probability is not expected, which may affect the reliability and security of the program.

Test results: No related vulnerabilities in smart contract code

Safety advice： None

3.3. ACCESS CONTROL【PASS】

Access Control issues are common in all programs,Also smart contracts. The famous Parity Wallet smart contract has been affected by this issue.

Test results: No related vulnerabilities in smart contract code

Safety advice： None。

3.4. UNCHECKED RETURN VALUES FOR LOW LEVEL CALLS【PASS】

Also known as or related to silent failing sends, unchecked-send. There are transfer methods such as transfer(), send(), and call.value() in Solidity and can be used to send Ether to an address. The difference is: transfer will be thrown when failed to send, and rollback; only 2300gas will be passed for call to prevent reentry attacks; send will return false if send fails; only 2300gas will be passed for call to prevent reentry attacks; If .value fails to send, it will return false; passing all available gas calls (which can be restricted by passing in the gas_value parameter) cannot effectively prevent reentry attacks.

If the return value of the send and call.value switch functions is not been checked in the code, the contract will continue to execute the following code,and it may have caused unexpected results due to Ether sending failure.

Test results: No related vulnerabilities in smart contract code

Safety advice： None。

3.5. BAD RANDOMNESS【PASS】

Smart Contract May Need to Use Random Numbers. While Solidity offers functions and variables that can access apparently hard-to-predict values just as block.number and block.timestamp. they are generally either more public than they seem or subject to miners' influence. Because these sources of randomness are to an extent predictable, malicious users can generally replicate it and attack the function relying on its unpredictablility.

Test results: No related vulnerabilities in smart contract code

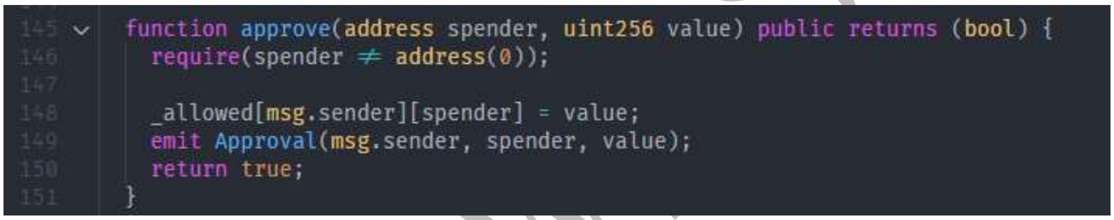
Safety advice： None。

3.6. TRANSACTION ORDERING DEPENDENCE【LOW】

Since miners always get rewarded via gas fees for running code on behalf of externally owned addresses (EOA), users can specify higher fees to have their transactions mined more quickly. Since the Ethereum blockchain is public, everyone can see the contents of others' pending transactions.

This means if a given user is revealing the solution to a puzzle or other valuable secret, a malicious user can steal the solution and copy their transaction with higher fees to preempt the original solution.

Test results ：The transactional order dependency attack risk in the approve function in the NTON token contract is detected as follows:



```
function approve(address spender, uint256 value) public returns (bool) {
  require(spender != address(0));

  _allowed[msg.sender][spender] = value;
  emit Approval(msg.sender, spender, value);
  return true;
}
```

Possible security risks are described below:

1. User A allows the number of user B transfers to be N (N > 0) by calling the approve function;

2. After a while, user A decided to change N to M (M > 0), so he called the approve function again;

3. User B quickly calls the transfer from function to transfer the number of N before the second call is processed by the miner. After user A's second call to approve is successful, user B can get the transfer amount of M again. That is, user B obtains the transfer amount of N+M by trading sequence attack.

Safety advice:

1. Front end restrictions, when user A changes the quota from N to M, it can be changed from N to 0, then from 0 to M：require((_value == 0) || (allowance[msg.sender][_spender] == 0));

2. Use the increaseapproval function and the increaseapproval function instead of the approve function

3.7. DENIAL OF SERVICE ATTACK DETECTION 【PASS】

In the ethernet world, denial of service is deadly, and smart contracts under attack of this type may never be able to return to normal. There may be a number of reasons for a denial of service in smart contracts, including malicious behavior as a recipient of transactions, gas depletion caused by artificially increased computing gas, and abuse of access control to access the private components of the intelligent contract. Take advantage of confusion and neglect, etc.

Test results: No related vulnerabilities in smart contract code.

Safety advice: None.

3.8. LOGICAL DESIGN FLAW 【PASS】

Detect the security problems related to business design in the contract code.

Test results: No related vulnerabilities in smart contract code.

Safety advice: None.

3.9. USDT FAKE DEPOSIT ISSUE 【PASS】

In the transfer function of the token contract, the balance check of the transfer initiator (msg.sender) is judged by if. When balances[msg.sender] < value, it enters the else logic part and returns false, and finally no exception is thrown. We believe that only the modest judgment of if/else is an imprecise coding method in the sensitive function scene such as transfer.

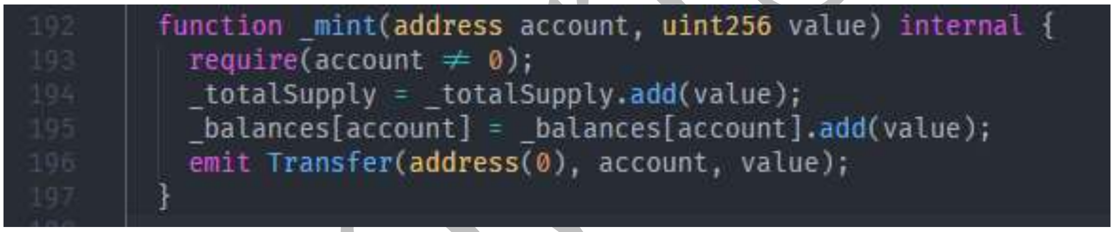
Detection results: No related vulnerabilities in smart contract code..

Safety advice: none

3.10. ADDING TOKENS 【LOW】

It is detected whether there is a function in the token contract that may increase the total amount of tokens after the total amount of tokens is initialized.

Detection results: After testing, there is a correlation function in the smart contract code, which can issue additional tokens, as shown in the figure.



```
function _mint(address account, uint256 value) internal {
  require(account ≠ 0);
  _totalSupply = _totalSupply.add(value);
  _balances[account] = _balances[account].add(value);
  emit Transfer(address(0), account, value);
}
```

Safety advice: This problem is not a security problem, but some exchanges will limit the use of the additional issue function, and the specific situation needs to be determined according to the requirements of the exchange.

3.11. FREEZING ACCOUNTS BYPASSED 【PASS】

In the token contract, when transferring the token, it is detected whether there is an unverified token source account, an originating account, and whether the target account is frozen.

Detection results: No related vulnerabilities in smart contract code.

Safety advice: none.

4. Appendix A：Contract code

The source of the code for this test:

https://etherscan.io/address/0xcfb152e5b93fc2c9906d4ff41fc8407dfa5e8851

```
/**
 *Submitted for verification at Etherscan.io on 2020-03-18
*/

pragma solidity ^0.4.24;

/**
 * @title SafeMath
 */
library SafeMath {

  /**
  * @dev Multiplies two numbers, reverts on overflow.
  */
  function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    if (a == 0) {
      return 0;
    }

    uint256 c = a * b;
    require(c / a == b);

    return c;
  }

  function div(uint256 a, uint256 b) internal pure returns (uint256) {
    require(b > 0);
    uint256 c = a / b;

    return c;
  }

  function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    require(b <= a);
    uint256 c = a - b;

    return c;
  }

  function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    require(c >= a);

    return c;
  }

  function mod(uint256 a, uint256 b) internal pure returns (uint256) {
    require(b != 0);
    return a % b;
  }
}

/**
 * @title Roles
 */
library Roles {
  struct Role {
    mapping (address => bool) bearer;
  }

  function add(Role storage role, address account) internal {
    require(account != address(0));
    role.bearer[account] = true;
  }

  function remove(Role storage role, address account) internal {
    require(account != address(0));
    role.bearer[account] = false;
  }

  function has(Role storage role, address account)
  internal
  view
  returns (bool)
  {
    require(account != address(0));
    return role.bearer[account];
  }
}


/**
 * @title ERC20 interface
 */
interface ERC20 {
  function totalSupply() external view returns (uint256);

  function balanceOf(address who) external view returns (uint256);

  function allowance(address owner, address spender)
  external view returns (uint256);

  function transfer(address to, uint256 value) external returns (bool);

  function approve(address spender, uint256 value)
  external returns (bool);

  event Transfer(
    address indexed from,
    address indexed to,
    uint256 value
  );

  event Approval(
    address indexed owner,
    address indexed spender,
    uint256 value
  );
}



/**
 * @title TokenBasic ERC20 token
 */
contract TokenBasic is ERC20 {
  using SafeMath for uint256;
  mapping (address => uint256) private _balances;
  mapping (address => mapping (address => uint256)) private _allowed;
  uint256 private _totalSupply;

  function totalSupply() public view returns (uint256) {
    return _totalSupply;
  }

  function balanceOf(address owner) public view returns (uint256) {
    return _balances[owner];
  }

  function allowance(
    address owner,
    address spender
  )
  public
  view
  returns (uint256)
  {
    return _allowed[owner][spender];
  }

  function transfer(address to, uint256 value) public returns (bool) {
    _transfer(msg.sender, to, value);
    return true;
  }

  function approve(address spender, uint256 value) public returns (bool) {
    require(spender != address(0));

    _allowed[msg.sender][spender] = value;
    emit Approval(msg.sender, spender, value);
    return true;
  }
```

```
function increaseAllowance(
  address spender,
  uint256 addedValue
)
public
returns (bool)
{
  require(spender != address(0));

  _allowed[msg.sender][spender] = (
  _allowed[msg.sender][spender].add(addedValue));
  emit Approval(msg.sender, spender, _allowed[msg.sender][spender]);
  return true;
}

function decreaseAllowance(
  address spender,
  uint256 subtractedValue
)
public
returns (bool)
{
  require(spender != address(0));

  _allowed[msg.sender][spender] = (
  _allowed[msg.sender][spender].sub(subtractedValue));
  emit Approval(msg.sender, spender, _allowed[msg.sender][spender]);
  return true;
}

function _transfer(address from, address to, uint256 value) internal {
  require(value <= _balances[from]);
  require(to != address(0));

  _balances[from] = _balances[from].sub(value);
  _balances[to] = _balances[to].add(value);
  emit Transfer(from, to, value);
}
```

```
  function _mint(address account, uint256 value) internal {
    require(account != 0);
    _totalSupply = _totalSupply.add(value);
    _balances[account] = _balances[account].add(value);
    emit Transfer(address(0), account, value);
  }

  function _burn(address account, uint256 value) internal {
    require(account != 0);
    require(value <= _balances[account]);

    _totalSupply = _totalSupply.sub(value);
    _balances[account] = _balances[account].sub(value);
    emit Transfer(account, address(0), value);
  }

  function _burnFrom(address account, uint256 value) internal {
    require(value <= _allowed[account][msg.sender]);

    _allowed[account][msg.sender] = _allowed[account][msg.sender].sub(
      value);
    _burn(account, value);
  }
}

contract MinterRole {
  using Roles for Roles.Role;

  event MinterAdded(address indexed account);
  event MinterRemoved(address indexed account);

  Roles.Role private minters;

  constructor() public {
    _addMinter(msg.sender);
  }

  modifier onlyMinter() {
    require(isMinter(msg.sender));
    _;
  }

  function isMinter(address account) public view returns (bool) {
    return minters.has(account);
  }

  function addMinter(address account) public onlyMinter {
    _addMinter(account);
  }

  function renounceMinter() public {
    _removeMinter(msg.sender);
  }

  function _addMinter(address account) internal {
    minters.add(account);
    emit MinterAdded(account);
  }

  function _removeMinter(address account) internal {
    minters.remove(account);
    emit MinterRemoved(account);
  }
}

/**
 * @title Mintable
 */
contract Mintable is TokenBasic, MinterRole {
  function mint(
    address to,
    uint256 value
  )
  public
  onlyMinter
  returns (bool)
  {
    _mint(to, value);
    return true;
  }
}

/**
 * @title Burnable Token
 */
contract Burnable is TokenBasic {

  function burn(uint256 value) public {
    _burn(msg.sender, value);
  }

  function burnFrom(address from, uint256 value) public {
    _burnFrom(from, value);
  }
}


/**
 * @title NTON
 */
contract NTON is Burnable, Mintable {

  string public constant name = "NTON";
  string public constant symbol = "NTON";
  uint8 public constant decimals = 18;
  uint256 public constant INITIAL_SUPPLY = 3500000000 * (10 ** uint256(decimals));

  constructor(address _owner) public {
    _mint(_owner, INITIAL_SUPPLY);
  }

}
```

5. Appendix B: vulnerability risk rating criteria

Smart contract vulnerability rating standard	
Vulnerability rating	Vulnerability rating description
High risk vulnerability	The loophole which can directly cause the contract or the user's fund loss, such as the value overflow loophole which can cause the value of the substitute currency to zero, the false recharge loophole that can cause the exchange to lose the substitute coin, can cause the contract account to lose the ETH or the reentry loophole of the substitute currency, and so on; It can cause the loss of ownership rights of token contract, such as: the key function access control defect or call injection leads to the key function access control bypassing, and the loophole that the token contract can not work properly. Such as: a denial-of-service vulnerability due to sending ETHs to a malicious address, and a denial-of-service vulnerability due to gas depletion.
Middle risk vulnerability	High risk vulnerabilities that need specific addresses to trigger, such as numerical overflow vulnerabilities that can be triggered by the owner of a token contract, access control defects of non-critical functions, and logical design defects that do not result in direct capital losses, etc.
Low risk vulnerability	A vulnerability that is difficult to trigger, or that will harm a limited number after triggering, such as a numerical overflow that requires a large number of ETH or tokens to trigger, and a vulnerability that the attacker cannot directly profit from after triggering a numerical overflow. Rely on risks by specifying the order of transactions triggered by a high gas.

6. Appendix C：Introduction of test tool

6.1. MANTICORE

Manticore is a symbolic execution tool for analysis of binaries and smart contracts. It discovers inputs that crash programs via memory safety violations. Manticore records an instruction-level trace of execution for each generated input and exposes programmatic access to its analysis engine via a Python API.

6.2. OYENTE

Oyente is a smart contract analysis tool that Oyente can use to detect common bugs in smart contracts, such as reentrancy, transaction ordering dependencies, and more. More conveniently, Oyente's design is modular, so this allows advanced users to implement and insert their own detection logic to check for custom attributes in their contracts.

6.3. SECURIFY.SH

Securify can verify common security issues with Ethereum smart contracts, such as transactional out-of-order and lack of input validation. It analyzes all possible execution paths of the program while fully automated. In addition, Securify has a specific language for specifying vulnerabilities. Securify can keep an eye on current security and other reliability issues.

6.4. ECHIDNA

Echidna is a Haskell library designed for fuzzing EVM code.

6.5. MAIAN

MAIAN is an automated tool for finding Ethereum smart contract vulnerabilities. Maian deals with the contract's bytecode and tries to establish a series of transactions to find and confirm errors.

6.6. ETHERSPLAY

Ethersplay is an EVM disassembler that contains related analysis tools.

6.7. IDA-EVM

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

6.8. REMIX-IDE

Remix is a browser-based compiler and IDE that allows users to build Ethereum contracts and debug transactions using the Solidity language.

6.9. KNOWNSEC PENETRATION TESTER SPECIAL TOOLKIT

Knownsec penetration tester special tool kit, developed and collected by Knownsec penetration testing engineers, includes batch automatic testing tools dedicated to testers, self-developed tools, scripts, or utility tools.