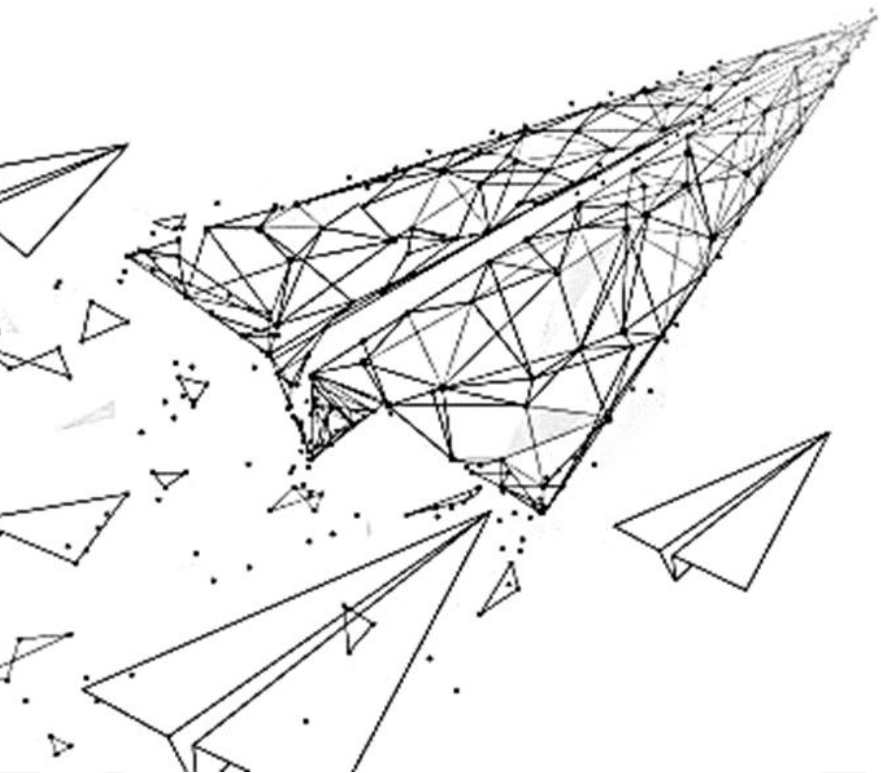
盾牌安全

2020
FOUR AUDIT REPORT



File No：DP- 20200717FOUR

Table of Contents

I、Overview

The date of issuing this report is July 17, 2020. The security and specifications of the FOUR smart contract code are audited and used as the statistical basis for the report.

In this test, a comprehensive analysis of common vulnerabilities in smart contracts (see Chapter 3) was conducted. The FOUR contract code complies with the ERC-20 specification, and no known vulnerabilities have been found; thus, the comprehensive audit of FOUR is passed.

Since this test process is carried out in a testing environment, all codes are the latest version. The test process communicates with the person in charge of the relevant interface, and conducts related test operations under the control of operational risks to avoid the production, operational code and security risks during test process.

Target information for this test: module name

Token: The 4th Pillar Token (FOUR)

Code type: Token code

Contract address：0x4730fB1463A6F1F44AEB45F6c5c422427f37F4D0

Address：

https://cn.etherscan.com/token/0x4730fB1463A6F1F44AEB45F6c5c422427f37F4D0

Code language: solidity

II、Code Vulnerability Analysis

Vulnerability level distribution

The vulnerability risk level according to statistics:High risk 0 、Medium risk 0,Low risk 0, Passed 11

Audit Results Summary

Reentry attack detection: Check if the use of call.value() function is security, **passed.**

Numerical overflow detection: Check if the add and sub functions are security to use, **passed.**

Access control defect detection: Check access control of each operation, **passed.**

Call of unverified return value: Check the transfer method to see if the return value is verified, **passed.**

Random number using detection: Check whether there is a unified content filter, **passed.**

Transaction sequence dependency detection: Check transaction sequence dependency, **passed.**

Denial of service attack detection: Check whether the code has resource abuse problems when using resources, **passed.**

Logical design defect detection: Check the security issues related to business design in the smart contract code, **passed.**

Fake recharge vulnerability detection: Check whether there is a fake recharge vulnerability in the smart contract code, **passed.**

Vulnerability detection of additional tokens: Check whether there is a function of additional tokens in the smart contract code, **passed.**

Frozen account bypass: Check whether there is a frozen account bypass problem in the smart contract code, **passed.**

III、Analysis of Code Audit Results

Reentry attack detection: [passed]

The reentry vulnerability is the most famous Ethereum smart contract vulnerability, which has led to the Ethereum fork (The DAO hack).

The call.value() function in Solidity consumes all the gas it receives when it is used to send Ether. There is a risk of reentry attacks when the call.value() function sending Ether occurs before actually reducing the balance of the sender's account,

Test result: After testing, there is no relevant call in the smart contract code.

Suggestions: None.

Numerical overflow detection: [Passed]

The arithmetic problem in smart contracts refers to integer overflow and integer underflow.

Solidity can handle up to 256 digits (2^256-1), and increasing the maximum number by 1 will overflow to 0. Similarly, when the number is of unsigned type, 0

minus 1 will underflow to get the maximum number value. Integer overflow and underflow are not a new type of vulnerability, but they are particularly dangerous in smart contracts. An overflow situation can lead to incorrect results, especially if the possibility is not expected, which may affect the reliability and security of the program.

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

Suggestions: None.

Access control test: [Passed]

Access control deficiencies are possible security risks in all programs. Smart contracts also have similar problems. The famous Parity Wallet smart contract has been affected by this problem.

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

Suggestions: None.

Return value call verification: [Passed]

This problem mostly occurs in smart contracts related to token transfer, so it is also called silent failure transmission or unchecked transmission.

In Solidity, there are transfer methods such as transfer(), send(), call.value(), etc., which can be used to send Ether to an address, the difference is that: transfer will throw when the transfer fails, and the status will be rolled back; Only 2300gas will be passed for calling to prevent reentry attacks; false will be returned when send fails to send; 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 called for Restricted by passing in the gas_value parameter), 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 cause unexpected results due to the failure of Ether transmission.

Test result: After testing, there are no related vulnerabilities in the smart contract code.

Suggestions: None.

Wrong usage of random numbers:【Passed】

Random numbers may be required in smart contracts. Although the functions and variables provided by Solidity can access obviously unpredictable values, such as block.number and block.timestamp, they are usually either more public than they seem 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.

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

Suggestions: None.

Transaction sequence depends: [Passed]

Since miners always obtain gas fees through code representing externally owned addresses (EOA), users can specify a higher fee 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.

Test result: After testing, there is no risk of transaction order dependence attack in the approval function in the smart contract.

Suggestions: None.

Denial of service attack: [Passed]

In the Ethereum world, denial of service is fatal, and a smart contract that suffers 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 when acting as the recipient of the transaction, artificially increasing the gas required for the calculation function leads to the exhaustion of gas, abuse of access control to access the private component of the smart contract, the use of confusion and negligence, etc.

Test result: After testing, there are no related vulnerabilities in the smart contract code.

Suggestions: None.

Logic design defects: [Passed]

Detect security issues related to business design in smart contract code.

Test result: After testing, there are no related vulnerabilities in the logic design of the smart contract code.

Suggestions: None.

Fake recharge vulnerability: [Passed]

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

Test result: After testing, there are no related vulnerabilities in the smart contract code.

Suggestions: None.

Vulnerability of additional tokens: [passed]

After the initialization of the total amount of tokens, check whether there is a function in the token contract that may increase the total amount of tokens.

Test result: After testing, there are no additional token issuance vulnerabilities in the smart contract code.

Suggestions: None.

Frozen account bypass: [passed]

Check whether there is any operation of unverified token source account, originating account and target account when transferring tokens in the token contract.

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

Suggestions: None.

IV、Appendix A: Contract Code

Note: For audit opinions and recommendations, see code comments //AUDIT//...

```
//AUDIT// There is no overflow or conditional competition in the contract.
/**
 *Submitted for verification at Etherscan.io on 2018-04-23
*/
pragma solidity ^0.4.17;
// File: contracts/helpers/Ownable.sol
/**
```

```
* @title Ownable
* @dev The Ownable contract has an owner address, and provides basic
authorization control
* functions, this simplifies the implementation of "user permissions".
*/
contract Ownable {
  address public owner;
  event OwnershipTransferred(address indexed previousOwner, address
indexed newOwner);
  /**
  * @dev The Constructor sets the original owner of the contract to the
  * sender account.
  */
  function Ownable() public {
    owner = msg.sender;
  }
  /**
  * @dev Throws if called by any other account other than owner.
  */
  modifier onlyOwner() {
    require(msg.sender == owner);
    _;
  }
  /**
   * @dev Allows the current owner to transfer control of the contract to a
newOwner.
```

```
    * @param newOwner The address to transfer ownership to.
    */
  function transferOwnership(address newOwner) public onlyOwner {
    require(newOwner != address(0));
    OwnershipTransferred(owner, newOwner);
    owner = newOwner;
  }
}
// File: contracts/helpers/SafeMath.sol
/**
 * @title SafeMath
 * @dev Math operations with safety checks that throw on error
 */
library SafeMath {
  /**
  * @dev Multiplies two numbers, throws on overflow.
  */
  function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    if (a == 0) {
      return 0;
    }
    uint256 c = a * b;
    assert(c / a == b);
    return c;
  }
  /**
```

```
    * @dev Integer division of two numbers, truncating the quotient.
    */
    function div(uint256 a, uint256 b) internal pure returns (uint256) {
      // assert(b > 0); // Solidity automatically throws when dividing by 0
      uint256 c = a / b;
      // assert(a == b * c + a % b); // There is no case in which this doesn't hold
      return c;
    }
    /**
    * @dev Substracts two numbers, throws on overflow (i.e. if subtrahend is
greater than minuend).
    */
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
      assert(b <= a);
      return a - b;
    }
    /**
    * @dev Adds two numbers, throws on overflow.
    */
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
      uint256 c = a + b;
      assert(c >= a);
      return c;
    }
  }
```

```
// File: contracts/token/ERC20Interface.sol
contract ERC20Interface {
    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256
value);
    function totalSupply() public view returns (uint256);
    function balanceOf(address who) public view returns (uint256);
    function transfer(address to, uint256 value) public returns (bool);
    function approve(address spender, uint256 value) public returns (bool);
    function transferFrom(address from, address to, uint256 value) public returns
(bool);
    function allowance(address owner, address spender) public view returns
(uint256);
}
// File: contracts/token/BaseToken.sol
contract BaseToken is ERC20Interface {
    using SafeMath for uint256;
    mapping(address => uint256) balances;
    mapping (address => mapping (address => uint256)) internal allowed;
    uint256 totalSupply_;
    /**
    * @dev Obtain total number of tokens in existence
    */
    function totalSupply() public view returns (uint256) {
      return totalSupply_;
    }
```

```
    /**
    * @dev Gets the balance of the specified address.
    * @param _owner The address to query the the balance of.
    * @return An uint256 representing the amount owned by the passed
address.
    */
    function balanceOf(address _owner) public view returns (uint256 balance) {
      return balances[_owner];
    }
    /**
    * @dev Transfer token for a specified address
    * @param _to The address to transfer to.
    * @param _value The amount to be transferred.
    */
    function transfer(address _to, uint256 _value) public returns (bool) {
      require(_to != address(0));
      require(_value <= balances[msg.sender]);
      // SafeMath.sub will throw if there is not enough balance
      balances[msg.sender] = balances[msg.sender].sub(_value);
      balances[_to] = balances[_to].add(_value);
      Transfer(msg.sender, _to, _value);
      return true;
  //AUDIT// The return value conforms to the ERC-20 specification.

    }
```

```
    /**
     * @dev Approve the passed address to spend the specified amount of
tokens on behalf of msg.sender.
     *
     * Beware that changing an allowance with this method brings the risk that
someone may use both the old
     * and the new allowance by unfortunate transaction ordering. One possible
solution to mitigate this
     * race condition is to first reduce the spender's allowance to 0 and set the
desired value afterwards:
     * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
     * @param _spender The address which will spend the funds.
     * @param _value The amount of tokens to be spent.
     */
    function approve(address _spender, uint256 _value) public returns (bool) {
      require(_spender != address(0));
      allowed[msg.sender][_spender] = _value;
      Approval(msg.sender, _spender, _value);
      return true;
  //AUDIT// The return value conforms to the ERC-20 specification.

    }

    /**
     * @dev Transfer tokens from one address to another
     * @param _from address The address which you want to send tokens from
     * @param _to address The address which you want to transfer to
```

```
     * @param _value uint256 the amount of tokens to be transferred
     */
    function transferFrom(address _from, address _to, uint256 _value) public
returns (bool) {
      require(_to != address(0));
      require(_value <= balances[_from]);
      require(_value <= allowed[_from][msg.sender]);
      balances[_from] = balances[_from].sub(_value);
      balances[_to] = balances[_to].add(_value);
      allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(_value);
      Transfer(_from, _to, _value);
      return true;
  //AUDIT// The return value conforms to the ERC-20 specification.
    }
    /**
     * @dev Function to check the amount of tokens that an owner allowed to a
spender.
     * @param _owner address The address which owns the funds.
     * @param _spender address The address which will spend the funds.
     * @return A uint256 specifying the amount of tokens still available for the
spender.
     */
    function allowance(address _owner, address _spender) public view returns
(uint256) {
      return allowed[_owner][_spender];
    }
```

```
    /**
     * @dev Increase the amount of tokens that an owner allowed to a spender.
     *
     * approve should be called when allowed[_spender] == 0. To increment
     * allowed value is better to use this function to avoid 2 calls (and wait until
     * the first transaction is mined)
     * From MonolithDAO Token.sol
     * @param _spender The address which will spend the funds.
     * @param _addedValue The amount of tokens to increase the allowance by.
     */
    function increaseApproval(address _spender, uint256 _addedValue) public
returns (bool) {
        allowed[msg.sender][_spender] =
allowed[msg.sender][_spender].add(_addedValue);
        Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
        return true;
    }
    /**
     * @dev Decrease the amount of tokens that an owner allowed to a spender.
     *
     * approve should be called when allowed[_spender] == 0. To decrement
     * allowed value is better to use this function to avoid 2 calls (and wait until
     * the first transaction is mined)
     * From MonolithDAO Token.sol
     * @param _spender The address which will spend the funds.
```

```
     * @param _subtractedValue The amount of tokens to decrease the
allowance by.
     */
    function decreaseApproval(address _spender, uint256 _subtractedValue)
public returns (bool) {
      uint oldValue = allowed[msg.sender][_spender];
      if (_subtractedValue > oldValue) {
        allowed[msg.sender][_spender] = 0;
      } else {
        allowed[msg.sender][_spender] = oldValue.sub(_subtractedValue);
      }
      Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
      return true;
    }
  }
  // File: contracts/token/MintableToken.sol
  /**
   * @title Mintable token
   * @dev Simple ERC20 Token example, with mintable token creation
   * @dev Issue: *
https://github.com/OpenZeppelin/zeppelin-solidity/issues/120
   * Based on code by TokenMarketNet:
https://github.com/TokenMarketNet/ico/blob/master/contracts/MintableToken.so
l
   */
  contract MintableToken is BaseToken, Ownable {
```

```
    event Mint(address indexed to, uint256 amount);
    event MintFinished();

    bool public mintingFinished = false;
    modifier canMint() {
      require(!mintingFinished);
      _;
    }
    /**
     * @dev Function to mint tokens
     * @param _to The address that will receive the minted tokens.
     * @param _amount The amount of tokens to mint.
     * @return A boolean that indicates if the operation was successful.
     */
    function mint(address _to, uint256 _amount) onlyOwner canMint public
returns (bool) {
      require(_to != address(0));
      totalSupply_ = totalSupply_.add(_amount);
      balances[_to] = balances[_to].add(_amount);
      Mint(_to, _amount);
      Transfer(address(0), _to, _amount);
      return true;
    }
    /**
     * @dev Function to stop minting new tokens.
     * @return True if the operation was successful.
```

```
     */
    function finishMinting() onlyOwner canMint public returns (bool) {
      mintingFinished = true;
      MintFinished();
      return true;
    }
  }
  // File: contracts/token/CappedToken.sol
  contract CappedToken is MintableToken {
    uint256 public cap;
    function CappedToken(uint256 _cap) public {
      require(_cap > 0);
      cap = _cap;
    }
    function mint(address _to, uint256 _amount) onlyOwner canMint public
returns (bool) {
      require(totalSupply_.add(_amount) <= cap);
      return super.mint(_to, _amount);
    }
  }
  // File: contracts/helpers/Pausable.sol
  /**
   * @title Pausable
   * @dev Base contract which allows children to implement an emergency stop
mechanism.
   */
```

```
contract Pausable is Ownable {
  event Pause();
  event Unpause();
  bool public paused = false;
  /**
   * @dev Modifier to make a function callable only when the contract is not
paused.
   */
  modifier whenNotPaused() {
    require(!paused);
    _;
  }
  /**
   * @dev Modifier to make a function callable only when the contract is
paused.
   */
  modifier whenPaused() {
    require(paused);
    _;
  }
  /**
   * @dev called by the owner to pause, triggers stopped state
   */
  function pause() onlyOwner whenNotPaused public {
    paused = true;
    Pause();
```

```
    }

    /**
     * @dev called by the owner to unpause, returns to normal state
     */
    function unpause() onlyOwner whenPaused public {
      paused = false;
      Unpause();
    }
  }
  // File: contracts/token/PausableToken.sol
  /**
   * @title Pausable token
   * @dev BaseToken modified with pausable transfers.
   **/
  contract PausableToken is BaseToken, Pausable {
    function transfer(address _to, uint256 _value) public whenNotPaused returns
(bool) {
      return super.transfer(_to, _value);
    }
    function transferFrom(address _from, address _to, uint256 _value) public
whenNotPaused returns (bool) {
      return super.transferFrom(_from, _to, _value);
    }
    function approve(address _spender, uint256 _value) public whenNotPaused
returns (bool) {
```

```
        return super.approve(_spender, _value);
    }

    function increaseApproval(address _spender, uint _addedValue) public
whenNotPaused returns (bool success) {
        return super.increaseApproval(_spender, _addedValue);
    }
    function decreaseApproval(address _spender, uint _subtractedValue) public
whenNotPaused returns (bool success) {
        return super.decreaseApproval(_spender, _subtractedValue);
    }
}
// File: contracts/token/SignedTransferToken.sol
/**
* @title SignedTransferToken
* @dev The SignedTransferToken enables collection of fees for token transfers
* in native token currency. User will provide a signature that allows the third
* party to settle the transaction in his name and collect fee for provided
* serivce.
*/
contract SignedTransferToken is BaseToken {
    event TransferPreSigned(
        address indexed from,
        address indexed to,
        address indexed settler,
        uint256 value,
```

```
    uint256 fee
);
event TransferPreSignedMany(
    address indexed from,
    address indexed settler,
    uint256 value,
    uint256 fee
);
// Mapping of already executed settlements for a given address
mapping(address => mapping(bytes32 => bool)) executedSettlements;
/**
* @dev Will settle a pre-signed transfer
*/
function transferPreSigned(address _from,
                           address _to,
                           uint256 _value,
                           uint256 _fee,
                           uint256 _nonce,
                           uint8 _v,
                           bytes32 _r,
                           bytes32 _s) public returns (bool) {
    uint256 total = _value.add(_fee);
    bytes32 calcHash = calculateHash(_from, _to, _value, _fee, _nonce);
    require(_to != address(0));
    require(isValidSignature(_from, calcHash, _v, _r, _s));
    require(balances[_from] >= total);
```

```
    require(!executedSettlements[_from][calcHash]);
    executedSettlements[_from][calcHash] = true;
    // Move tokens
    balances[_from] = balances[_from].sub(_value);
    balances[_to] = balances[_to].add(_value);
    Transfer(_from, _to, _value);
    // Move fee
    balances[_from] = balances[_from].sub(_fee);
    balances[msg.sender] = balances[msg.sender].add(_fee);
    Transfer(_from, msg.sender, _fee);
    TransferPreSigned(_from, _to, msg.sender, _value, _fee);
    return true;
}
/**
* @dev Settle multiple transactions in a single call. Please note that
* should a single one fail the full state will be reverted. Your client
* implementation should always first check for balances, correct signatures
* and any other conditions that might result in failed transaction.
*/
function transferPreSignedBulk(address[] _from,
                               address[] _to,
                               uint256[] _values,
                               uint256[] _fees,
                               uint256[] _nonces,
                               uint8[] _v,
                               bytes32[] _r,
```

```
                                    bytes32[] _s) public returns (bool) {
    // Make sure all the arrays are of the same length
    require(_from.length == _to.length &&
            _to.length ==_values.length &&
            _values.length == _fees.length &&
            _fees.length == _nonces.length &&
            _nonces.length == _v.length &&
            _v.length == _r.length &&
            _r.length == _s.length);
    for(uint i; i < _from.length; i++) {
      transferPreSigned(_from[i],
                        _to[i],
                        _values[i],
                        _fees[i],
                        _nonces[i],
                        _v[i],
                        _r[i],
                        _s[i]);
    }
    return true;
}
function transferPreSignedMany(address _from,
                               address[] _tos,
                               uint256[] _values,
                               uint256 _fee,
                               uint256 _nonce,
```

```
                                        uint8 _v,
                                        bytes32 _r,
                                        bytes32 _s) public returns (bool) {
        require(_tos.length == _values.length);
        uint256 total = getTotal(_tos, _values, _fee);
        bytes32 calcHash = calculateManyHash(_from, _tos, _values, _fee, _nonce);
        require(isValidSignature(_from, calcHash, _v, _r, _s));
        require(balances[_from] >= total);
        require(!executedSettlements[_from][calcHash]);
        executedSettlements[_from][calcHash] = true;
        // transfer to each recipient and take fee at the end
        for(uint i; i < _tos.length; i++) {
          // Move tokens
          balances[_from] = balances[_from].sub(_values[i]);
          balances[_tos[i]] = balances[_tos[i]].add(_values[i]);
          Transfer(_from, _tos[i], _values[i]);
        }
        // Move fee
        balances[_from] = balances[_from].sub(_fee);
        balances[msg.sender] = balances[msg.sender].add(_fee);
        Transfer(_from, msg.sender, _fee);
        TransferPreSignedMany(_from, msg.sender, total, _fee);
        return true;
      }
      function getTotal(address[] _tos, uint256[] _values, uint256 _fee) private view
returns (uint256)  {
```

```
        uint256 total = _fee;

        for(uint i; i < _tos.length; i++) {
          total = total.add(_values[i]); // sum of all the values + fee
          require(_tos[i] != address(0)); // check that the recipient is a valid address
        }
        return total;
      }
      /**
      * @dev Calculates transfer hash for transferPreSignedMany
      */
      function calculateManyHash(address _from, address[] _tos, uint256[] _values,
uint256 _fee, uint256 _nonce) public view returns (bytes32) {
        return keccak256(uint256(1), address(this), _from, _tos, _values, _fee,
_nonce);
      }
      /**
      * @dev Calculates transfer hash.
      */
      function calculateHash(address _from, address _to, uint256 _value, uint256
_fee, uint256 _nonce) public view returns (bytes32) {
        return keccak256(uint256(0), address(this), _from, _to, _value, _fee, _nonce);
      }
      /**
      * @dev Validates the signature
      */
```

```
        function isValidSignature(address _signer, bytes32 _hash, uint8 _v, bytes32 _r,
bytes32 _s) public pure returns (bool) {
            return _signer == ecrecover(
                    keccak256("\x19Ethereum Signed Message:\n32", _hash),
                    _v,
                    _r,
                    _s
                );
        }
        /**
        * @dev Allows you to check whether a certain transaction has been already
        * settled or not.
        */
        function isTransactionAlreadySettled(address _from, bytes32 _calcHash)
public view returns (bool) {
            return executedSettlements[_from][_calcHash];
        }
    }
    // File: contracts/token/PausableSignedTransferToken.sol
    contract PausableSignedTransferToken is SignedTransferToken,
PausableToken {
        function transferPreSigned(address _from,
                                   address _to,
                                   uint256 _value,
                                   uint256 _fee,
                                   uint256 _nonce,
```

```
                                    uint8 _v,
                                    bytes32 _r,
                                    bytes32 _s) public whenNotPaused returns (bool)
{
    return super.transferPreSigned(_from, _to, _value, _fee, _nonce, _v, _r, _s);
}
function transferPreSignedBulk(address[] _from,
                                    address[] _to,
                                    uint256[] _values,
                                    uint256[] _fees,
                                    uint256[] _nonces,
                                    uint8[] _v,
                                    bytes32[] _r,
                                    bytes32[] _s) public whenNotPaused returns
(bool) {
    return super.transferPreSignedBulk(_from, _to, _values, _fees, _nonces, _v,
_r, _s);
}
function transferPreSignedMany(address _from,
                                    address[] _tos,
                                    uint256[] _values,
                                    uint256 _fee,
                                    uint256 _nonce,
                                    uint8 _v,
                                    bytes32 _r,
```

```
                                        bytes32 _s) public whenNotPaused returns
(bool) {
            return super.transferPreSignedMany(_from, _tos, _values, _fee, _nonce, _v,
_r, _s);
        }
    }
    // File: contracts/FourToken.sol
    contract FourToken is CappedToken, PausableSignedTransferToken    {
      string public name = 'The 4th Pillar Token';
      string public symbol = 'FOUR';
      uint256 public decimals = 18;
      // Max supply of 400 million
      uint256 public maxSupply = 400000000 * 10**decimals;
      function FourToken()
        CappedToken(maxSupply) public {
          paused = true;
      }
      // @dev Recover any mistakenly sent ERC20 tokens to the Token address
      function recoverERC20Tokens(address _erc20, uint256 _amount) public
onlyOwner {
        ERC20Interface(_erc20).transfer(msg.sender, _amount);
      }
    }
```

V、Appendix B: Vulnerability Risk Rating Standards

Vulnerability rating	Vulnerability rating instructions
High-risk vulnerabilities	Vulnerabilities that can directly cause the loss of token or user funds, such as: numerical overflow vulnerabilities that can cause the value of the token to return to zero, fake recharge vulnerabilities that can cause the exchange to lose tokens, and reentrant vulnerabilities that can cause contract accounts to losses ETH or token , etc.; vulnerabilities that can cause the loss of ownership of token contracts, such as: access control defects of key functions, bypass of key function access control caused by call injection, etc.; vulnerabilities that can cause token contracts to not work properly, such as: Denial of service vulnerability caused by malicious address sending ETH, denial of service vulnerability due to gas exhaustion.
Medium-risk vulnerability	High-risk vulnerabilities that require specific addresses to trigger, such as numeric overflow vulnerabilities that can only be triggered by the token contract owner; access control defects of non-critical functions, logical design defects that cannot cause direct capital loss, etc.

Low-risk vulnerabilities	Vulnerabilities that are difficult to be triggered, vulnerabilities with limited damage after triggering, such as numerical overflow vulnerabilities that require a large amount of ETH or tokens to trigger, vulnerabilities that the attacker cannot directly profit after the numerical overflow is triggered, and the transaction sequence dependence risk triggered by specifying high gas Wait.

VI、Appendix C: Introduction to Vulnerability Testing Tools

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, a Bit of Traits of Bits visual disassembler for EVM bytecode, for visual analysis. Like binary files, Manticore provides a simple command-line interface and a Python API for analyzing EVM bytecode.

Oyente

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

securify.sh

Securify can verify the common security problems of Ethereum smart contracts, such as out-of-order transactions and lack of input verification. It analyzes all possible execution paths of the program while fully automated. In addition, Securify has a specific language for specifying vulnerabilities, which makes Securify can keep abreast of current security and other reliability issues.

Echidna

Echidna is a Haskell library designed for fuzzing EVM code.

MAIAN

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

ethersplay

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

ida-evm

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

Remix-ide

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

VII、Declaration

This report only issues the facts that have occurred or existed before the issuance. It is impossible to judge the security status of the project for the facts that occur or exist after the issuance.

The security audit analysis and other contents made in this report are only based on the documents and materials provided by the information provider as of the issuance of this report (referred to as "provided materials"). It is assumed that the information provided is not missing, tampered with, deleted or concealed. If the information provided has been missing, tampered with, deleted, concealed or reflected in a situation that is inconsistent with the actual situation, the resulting losses and adverse effects have nothing to do with this report.

This report only conducts the security audit within the contract and issues the security situation of the project, and does not involve the background and other conditions of the project.

FOUR AUDIT REPORT

Official website：https://www.dpanquan.com

Official email：service@dpanquan.com