Smart Contract Security Audit Report

The SlowMist Security Team received the Zild Finance team's application for smart contract security audit of the Zild Financeon Sep. 23, 2020. The following are the details and results of this smart contract security audit:

Token Name:

Zild

Contract Address:

Zlid Finacce token: 0x006699d34AA3013605d468d2755A2Fe59A16B12B

Deposit: 0xf366005F71f63CdEC1Be2A2f2aE9A5bb807e0dA4

Furnace: 0x45b9B8F4770d0410D3d12214d1e474c7fBBa4B07

Minter: 0x10905cF4A5b11e7aFA064971B60701C71A04D7F3

Contract Link:

Zlid Finance Token:

https://etherscan.io/address/0x006699d34AA3013605d468d2755A2Fe59A16B12B#code

Deposit:

https://etherscan.io/address/0xf366005F71f63CdEC1Be2A2f2aE9A5bb807e0dA4#code

Furnace:

https://etherscan.io/address/0x45b9B8F4770d0410D3d12214d1e474c7fBBa4B07#code

Minter:

https://etherscan.io/address/0x10905cF4A5b11e7aFA064971B60701C71A04D7F3#code

The audit items and results :

(Other unknown security vulnerabilities are not included in the audit responsibility scope)

No.	Audit Items	Audit Subclass	Audit Subclass Result
1	Overflow Audit	-	Passed
2	Race Conditions Audit	-	Passed
3	Authority Control Audit	Permission vulnerability audit	Passed
		Excessive auditing authority	Limited administrative rights exist

4	Safety Design Audit	Zeppelin module safe use	Passed
		Compiler version security	Passed
		Hard-coded address security	Passed
		Fallback function safe use	Passed
		Show coding security	Passed
		Function return value security	Passed
		Call function security	Passed
5	Denial of Service Audit	-	Passed
6	Gas Optimization Audit	-	Passed
7	Design Logic Audit	-	Passed
8	“False Deposit” vulnerability Audit	-	Passed
9	Malicious Event Log Audit	-	Passed
10	Scoping and Declarations Audit	-	Passed
11	Replay Attack Audit	ECDSA's Signature Replay Audit	Passed
12	Uninitialized Storage Pointers Audit	-	Passed
13	Arithmetic Accuracy Deviation Audit	-	Passed

Audit Result : Passed（Limited administrative rights exist）

Audit Number : 0X002009280001

Audit Date : Sep. 28, 2020

Audit Team : SlowMist Security Team

Summary: This is a token contract that contains the tokenVault section, during audit, we found following issues:

1. In the Deposit contract, The admin can set the depositBlock, but has the effective time, it's the size of the current block plus the old depositBlock.

Potential risk: Admin can affect the withdrawal time of users' staked assets when depositBlock changed. The user needs to be aware of parameter change events

2. In the Minter contract, admin and owner can set the minHandlingFee and handlingFeeRate, but there is an effective time, which is the size of the current block plus changeFeeWaitTime. The Admin and Owner can set the collector address any way they want, but only once.

Potential risk: admin and owner can influence the handling fee charged for each mint when minHandlingFee and handlingFee changed. The user needs to be aware of parameter change events.

3. In the ZildFinance contract, the owner can set any minter and transfer token to minter, but only once. The owner can set up any furnace and transfer token to the furnace, but only once.

Potential risks: Furnace and Minter can be set at will and then transfer tokens of corresponding tokens， but only once. This part of funds does not belong to user funds. According to the project design, the corresponding shares will be transferred to the Furnace.sol contract and the Minter.sol contract. Care must be taken to identify whether the relevant tokens are transferred correctly.

4. ZildFinance Contract allows the Owner to freeze and unfreeze any account.

Potential risk: Owner can freeze or unfreeze any address and the frozen account cannot be transferred.

The source code:

Deposit.sol:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.6.12;

//SlowMist// The contract does not have the Overflow and the Race Conditions issue

//SlowMist// OpenZeppelin's SafeMath security Module is used, which is a recommend approach

library SafeMath {

    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

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

        return c;
    }

    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
        // benefit is lost if 'b' is also tested.
        // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
        if (a == 0) {
            return 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }
    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(a, b, "SafeMath: division by zero");
    }

    function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

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

contract Ownable {
    address public owner;
    address public newowner;
    address public admin;
    address public dev;

    constructor() public {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    modifier onlyNewOwner {
        require(msg.sender == newowner);
        _;
    }
    function transferOwnership(address _newOwner) public onlyOwner {
        newowner = _newOwner;
    }

    function takeOwnership() public onlyNewOwner {
        owner = newowner;
    }

    function setAdmin(address _admin) public onlyOwner {
        admin = _admin;
    }

    function setDev(address _dev) public onlyOwner {
        dev = _dev;
    }

    modifier onlyAdmin {
        require(msg.sender == admin || msg.sender == owner);
        _;
    }

    modifier onlyDev {
        require(msg.sender == dev || msg.sender == admin || msg.sender == owner);
        _;
    }
}

abstract contract ContractConn{
    function transfer(address _to, uint _value) virtual public;
    function transferFrom(address _from, address _to, uint _value) virtual public;
    function balanceOf(address who) virtual public view returns (uint);
}


contract Deposit is Ownable{

    using SafeMath for uint256;

    struct DepositInfo {
        uint256 id;
        address depositor;
        string coinType;
        uint256 amount;
        uint256 depositTime;
        uint256 depositBlock;
        uint256 ExpireBlock;
    }

    ContractConn public usdt;
    ContractConn public zild;


    uint256 public depositBlock = 78000;
    uint256 public depositBlockChange;
    uint256 public changeDepositTime;
    bool     public needChangeTime = false;

    mapping(address => DepositInfo[]) public eth_deposit;
    mapping(address => DepositInfo[]) public usdt_deposit;
    mapping(address => DepositInfo[]) public zild_deposit;

    mapping(address => uint256) public user_ethdeposit_amount;
    mapping(address => uint256) public user_usdtdeposit_amount;
    mapping(address => uint256) public user_zilddeposit_amount;

    uint256 public ethTotalDeposit;
    uint256 public usdtTotalDeposit;
    uint256 public zildTotalDeposit;

    event SetDepositBlock(uint256 dblock,address indexed who,uint256 time);
    event EffectDepositBlock(uint256 dblock,address indexed who,uint256 time);
    event DepositETH(address indexed from,uint256 depid,uint256 damount,uint256 bblock,uint256 eblock,uint256 time);
    event DepositUSDT(address indexed from,uint256 depid,uint256 damount,uint256 bblock,uint256 eblock,uint256 time);
    event DepositZILD(address indexed from,uint256 depid,uint256 damount,uint256 bblock,uint256 eblock,uint256 time);
    event WithdrawETH(address indexed to,uint256 damount,uint256 time);
    event WithdrawUSDT(address indexed to,uint256 damount,uint256 time);
    event WithdrawZILD(address indexed to,uint256 damount,uint256 time);

    constructor(address _usdt,address _zild) public {
        usdt = ContractConn(_usdt);
        zild = ContractConn(_zild);
    }
```

```
 //SlowMist// Admin can set the depositBlock but has an effective time, it is the current block plus

the block size of the old depositBlock
    function setdepositblock(uint256 _block) public onlyAdmin {
        require(_block > 0,"Desposit: New deposit time must be greater than 0");
        depositBlockChange = _block;
        changeDepositTime = block.number;
        needChangeTime = true;
        emit SetDepositBlock(_block,msg.sender,now);
    }

    function effectblockchange() public onlyAdmin {
        require(needChangeTime,"Deposit: No new deposit time are set");
        uint256 currentTime = block.number;
        uint256 effectTime = changeDepositTime.add(depositBlock);
        if (currentTime < effectTime) return;
        depositBlock = depositBlockChange;
        needChangeTime = false;
        emit SetDepositBlock(depositBlockChange,msg.sender,now);
    }

    function DepositETHCount(address _user)   view public returns(uint256) {
        require(msg.sender == _user || msg.sender == owner, "Deposit: Only check your own deposit records");
        return eth_deposit[_user].length;
    }

    function DepositUSDTCount(address _user)   view public returns(uint256) {
        require(msg.sender == _user || msg.sender == owner, "Deposit: Only check your own deposit records");
        return usdt_deposit[_user].length;
    }

    function DepositZILDCount(address _user)   view public returns(uint256) {
        require(msg.sender == _user || msg.sender == owner, "Deposit: Only check your own deposit records");
        return zild_deposit[_user].length;
    }

    function DepositETHAmount(address _user)   view public returns(uint256) {
        require(msg.sender == _user || msg.sender == owner, "Deposit: Only check your own deposit records");
        return user_ethdeposit_amount[_user];
    }
```

```
function DepositUSDTAmount(address _user)   view public returns(uint256) {
    require(msg.sender == _user || msg.sender == owner, "Deposit: Only check your own deposit records");
    return user_usdtdeposit_amount[_user];
}

function DepositZILDAmount(address _user)   view public returns(uint256) {
    require(msg.sender == _user || msg.sender == owner, "Deposit: Only check your own deposit records");
    return user_zilddeposit_amount[_user];
}

function depositETH() public payable returns(uint256){
    uint256 length = eth_deposit[msg.sender].length;
    uint256 deposit_id;
    eth_deposit[msg.sender].push(
        DepositInfo({
            id: length,
            depositor: msg.sender,
            coinType: "eth",
            amount: msg.value,
            depositTime: now,
            depositBlock: block.number,
            ExpireBlock: block.number.add(depositBlock)
        })
    );

    //SlowMist// Redundant code

    deposit_id = eth_deposit[msg.sender].length;
    user_ethdeposit_amount[msg.sender] = user_ethdeposit_amount[msg.sender].add(msg.value);
    ethTotalDeposit = ethTotalDeposit.add(msg.value);
    emit DepositETH(msg.sender,length,msg.value,block.number,block.number.add(depositBlock),now);
    return length;
}

function depositUSDT(uint256 _amount) public returns(uint256){
    usdt.transferFrom(address(msg.sender), address(this), _amount);
    uint256 length = usdt_deposit[msg.sender].length;
    usdt_deposit[msg.sender].push(
        DepositInfo({
            id: length,
            depositor: msg.sender,
            coinType: "usdt",
            amount: _amount,
                depositTime: now,
                depositBlock: block.number,
                ExpireBlock: block.number.add(depositBlock)
            })
        );
        user_usdtdeposit_amount[msg.sender] = user_usdtdeposit_amount[msg.sender].add(_amount);
        usdtTotalDeposit = usdtTotalDeposit.add(_amount);
        emit DepositUSDT(msg.sender,length,_amount,block.number,block.number.add(depositBlock),now);
        return length;
    }

    function depositZILD(uint256 _amount) public returns(uint256){
        zild.transferFrom(address(msg.sender), address(this), _amount);
        uint256 length = zild_deposit[msg.sender].length;
        zild_deposit[msg.sender].push(
            DepositInfo({
                id: length,
                depositor: msg.sender,
                coinType: "zild",
                amount: _amount,
                depositTime: now,
                depositBlock: block.number,
                ExpireBlock: block.number.add(depositBlock)
            })
        );
        user_zilddeposit_amount[msg.sender] = user_zilddeposit_amount[msg.sender].add(_amount);
        zildTotalDeposit = zildTotalDeposit.add(_amount);
        emit DepositZILD(msg.sender,length,_amount,block.number,block.number.add(depositBlock),now);
        return length;
    }

    function withdrawEth(uint256 _deposit_id) public returns(bool){
        require(block.number > eth_deposit[msg.sender][_deposit_id].ExpireBlock, "The withdrawal block has not arrived!");
        require(eth_deposit[msg.sender][_deposit_id].amount > 0, "There is no deposit available!");
        msg.sender.transfer(eth_deposit[msg.sender][_deposit_id].amount);
        user_ethdeposit_amount[msg.sender] =
user_ethdeposit_amount[msg.sender].sub(eth_deposit[msg.sender][_deposit_id].amount);
        ethTotalDeposit = ethTotalDeposit.sub(eth_deposit[msg.sender][_deposit_id].amount);
        eth_deposit[msg.sender][_deposit_id].amount =  0;
        emit WithdrawETH(msg.sender,eth_deposit[msg.sender][_deposit_id].amount,now);
        return true;
    }
    function withdrawUSDT(uint256 _deposit_id) public returns(bool){
        require(block.number > usdt_deposit[msg.sender][_deposit_id].ExpireBlock, "The withdrawal block has not 
arrived!");
        require(usdt_deposit[msg.sender][_deposit_id].amount > 0, "There is no deposit available!");
        usdt.transfer(msg.sender, usdt_deposit[msg.sender][_deposit_id].amount);
        user_usdtdeposit_amount[msg.sender] = 
user_usdtdeposit_amount[msg.sender].sub(usdt_deposit[msg.sender][_deposit_id].amount);
        usdtTotalDeposit = usdtTotalDeposit.sub(usdt_deposit[msg.sender][_deposit_id].amount);
        usdt_deposit[msg.sender][_deposit_id].amount =  0;
        emit WithdrawUSDT(msg.sender,usdt_deposit[msg.sender][_deposit_id].amount,now);
        return true;
    }

    function withdrawZILD(uint256 _deposit_id) public returns(bool){
        require(block.number > zild_deposit[msg.sender][_deposit_id].ExpireBlock, "The withdrawal block has not arrived!");
        require(zild_deposit[msg.sender][_deposit_id].amount > 0, "There is no deposit available!");
        zild.transfer(msg.sender,zild_deposit[msg.sender][_deposit_id].amount);
        user_zilddeposit_amount[msg.sender] = 
user_zilddeposit_amount[msg.sender].sub(zild_deposit[msg.sender][_deposit_id].amount);
        zildTotalDeposit = zildTotalDeposit.sub(zild_deposit[msg.sender][_deposit_id].amount);
        zild_deposit[msg.sender][_deposit_id].amount =  0;
        emit WithdrawZILD(msg.sender,zild_deposit[msg.sender][_deposit_id].amount,now);
        return true;
    }
}
```

Furnace.sol

```
// SPDX-License-Identifier: MIT
pragma solidity 0.6.12;
```

//SlowMist// The contract does not have the Overflow and the Race Conditions issue

```
abstract contract Context {
    function _msgSender() internal view virtual returns (address payable) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes memory) {
        this; // silence state mutability warning without generating bytecode - see 
https://github.com/ethereum/solidity/issues/2691
        return msg.data;
    }
```

```
}

contract Ownable is Context {
    address private _owner;

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    /**
     * @dev Initializes the contract setting the deployer as the initial owner.
     */
    constructor () internal {
        address msgSender = _msgSender();
        _owner = msgSender;
        emit OwnershipTransferred(address(0), msgSender);
    }

    /**
     * @dev Returns the address of the current owner.
     */
    function owner() public view returns (address) {
        return _owner;
    }

    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        require(_owner == _msgSender(), "Ownable: caller is not the owner");
        _;
    }

    function renounceOwnership() public virtual onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }
```

//SlowMist// It is recommended to add a function to confirm the transfer when the permission is transferred. Only when the newOwner calls the function to confirm the transfer will the Owner actually be transferred

```
    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
}
```

//SlowMist// OpenZeppelin's SafeMath security Module is used, which is a recommend approach

```
library SafeMath {

    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

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

        return c;
    }

    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
        // benefit is lost if 'b' is also tested.
        // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
        if (a == 0) {
            return 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }

    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(a, b, "SafeMath: division by zero");
    }

    function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

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

abstract contract ContractConn{

    function burn(uint256 _value) virtual public returns(bool);
}

contract Furnace is Ownable {

    using SafeMath for uint256;
    uint256 public tokenBurned = 0;

    event BurnToken(uint256 amount,address indexed who,uint256 time);

    ContractConn public token;
    constructor(address _token) public {
        token = ContractConn(_token);
    }

    //SlowMist// Owner can burn tokens from the contract
    function combustion(uint256 amount) public onlyOwner returns(bool){
        require(amount > 0, "combustion:  amount error");
        token.burn(amount);
        tokenBurned = tokenBurned.add(amount);
        emit BurnToken(amount,msg.sender,now);
        return true;
    }

}
```

Minter.sol

```
// SPDX-License-Identifier: MIT

//SlowMist// The contract does not have the Overflow and the Race Conditions issue

pragma solidity 0.6.12;

abstract contract Context {
    function _msgSender() internal view virtual returns (address payable) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes memory) {
        this; // silence state mutability warning without generating bytecode - see
https://github.com/ethereum/solidity/issues/2691
        return msg.data;
    }
}

contract Ownable is Context {
    address private _owner;
    address public admin;
    address public dev;

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    /**
     * @dev Initializes the contract setting the deployer as the initial owner.
     */
    constructor () internal {
        address msgSender = _msgSender();
        _owner = msgSender;
        emit OwnershipTransferred(address(0), msgSender);
    }
    /**
     * @dev Returns the address of the current owner.
     */
    function owner() public view returns (address) {
        return _owner;
    }

    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        require(_owner == _msgSender(), "Ownable: caller is not the owner");
        _;
    }

    function renounceOwnership() public virtual onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }
```

//SlowMist// It is recommended to add a function to confirm the transfer when the permission is transferred. Only when the newOwner calls the function to confirm the transfer will the Owner actually be transferred

```
    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }

    function setAdmin(address _admin) public onlyOwner {
        admin = _admin;
    }

    function setDev(address _dev) public onlyOwner {
        dev = _dev;
    }

    modifier onlyAdmin {
        require(msg.sender == admin || msg.sender == _owner);
        _;
    }

    modifier onlyDev {
        require(msg.sender == dev || msg.sender == admin || msg.sender == _owner);
        _;
    }
}
```

//SlowMist// OpenZeppelin's SafeMath security Module is used, which is a recommend approach

```
library SafeMath {

    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

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

        return c;
    }

    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
        // benefit is lost if 'b' is also tested.
        // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
        if (a == 0) {
            return 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }
    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(a, b, "SafeMath: division by zero");
    }

    function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

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


abstract contract ContractConn{

    function transfer(address _to, uint256 _value) virtual public;
    function balanceOf(address who) virtual public view returns (uint256);
}

contract Minter is Ownable {

    using SafeMath for uint256;
    uint256 public userMinted = 0;
    uint256 public minHandlingFee = 5;
    uint256 public minHandlingFeeNew;
    uint256 public handlingFeeRate = 4;
    uint256 public handlingFeeRateNew;
    uint256 public changeFeeWaitTime = 12000;
    uint256 public changeFeeTime;
    bool    public needChangeFee = false;
    uint256 public handlingFeeCollect;
    uint256 public minedAmount;
    address public collector = address(0);

    ContractConn public zild;

    mapping(address => uint256) public user_minter_amount;
    mapping(address => uint256) public user_minter_netincome;
    mapping(address => uint256) public user_minter_fee;

    event MinterRevenue(uint256 total,address indexed who,uint256 amount, uint256 handlingfee,uint256 netincome,uint256
userTotal);
    event SetCollector(address indexed collector,uint256 time);
    event CollectHandlingFee(uint256 amount,uint256 handlingFeeCollect,uint256 now);
    event SetHandlingFee(uint256 fee,uint256 rate,address indexed who,uint256 time);
    event EffectHandlingFee(uint256 fee,uint256 rate,address indexed who,uint256 time);

    constructor(address _token) public {
        zild = ContractConn(_token);
    }

    function generate(uint256 amount) public onlyOwner returns(bool){
        require(amount > 0, "minter： generate amount error");
        require(amount <= zild.balanceOf(address(this)), "minter： insufficient balance generates more mines");
        minedAmount = minedAmount.add(amount);
        return true;
    }

    function minter(address _to, uint256 amount) public onlyDev returns(bool){
        require(amount > minHandlingFee.mul(10 ** 18).div(100), "minter： withdrawal amount must be greater than the minimum
handling fee");
        require(amount <= minedAmount,"minter： Not so many mined token");
        uint256 handlingfee = amount.mul(handlingFeeRate).div(1000);
        if (handlingfee < minHandlingFee.mul(10 ** 18).div(100)) handlingfee = minHandlingFee.mul(10 ** 18).div(100);
        zild.transfer(_to,amount.sub(handlingfee));
        minedAmount = minedAmount.sub(amount);
        userMinted = userMinted.add(amount);
        user_minter_amount[_to] = user_minter_amount[_to].add(amount);
        user_minter_netincome[_to] = user_minter_netincome[_to].add(amount.sub(handlingfee));
        user_minter_fee[_to] = user_minter_fee[_to].add(handlingfee);
        handlingFeeCollect = handlingFeeCollect.add(handlingfee);
        emit MinterRevenue(userMinted,_to,amount,handlingfee,amount.sub(handlingfee),user_minter_amount[_to]);
        return true;
    }

    //SlowMist// The admin and owner can set the collector address any way they want, but only
once
    function setCollector(address _collector) public onlyAdmin {
        require(_collector != address(0), "Minter: collector is the zero address");
        collector = _collector;
        emit SetCollector(_collector,now);
    }

    function collectHandlingFee(uint256 amount) public onlyAdmin returns(bool){
        require(amount > 0, "minter： collect amount error");
        require(amount <= handlingFeeCollect, "minter： withdrawal amount exceeds collector balance");
        zild.transfer(collector,amount);
        handlingFeeCollect = handlingFeeCollect.sub(amount);
        emit CollectHandlingFee(amount,handlingFeeCollect,now);
        return true;
    }
    //SlowMist// admin and owner can set any rate, but there is an effective time, it is
changeFeeWaitTime size plus the current block
    function setHandlingFee(uint256 _fee,uint256 _rate) public onlyAdmin {
        require(_fee > 0 || _rate > 0,"Minter: New handling fee rate must be greater than 0");
        minHandlingFeeNew = _fee;
        handlingFeeRateNew = _rate;
        changeFeeTime = block.number;
        needChangeFee = true;
        emit SetHandlingFee(_fee,_rate,msg.sender,now);
    }
    function effectblockchange() public onlyAdmin {
        require(needChangeFee,"Minter: No new handling fee rate are set");
        uint256 currentTime = block.number;
        uint256 effectTime = changeFeeTime.add(changeFeeWaitTime);
        if (currentTime < effectTime) return;
        minHandlingFee = minHandlingFeeNew;
        handlingFeeRate = handlingFeeRateNew;
        needChangeFee = false;
        emit EffectHandlingFee(minHandlingFee,handlingFeeRate,msg.sender,now);
    }
}
```

IERC20.sol

```
pragma solidity 0.5.4;

interface IERC20 {

    function balanceOf(address account) external view returns (uint256);
    function transfer(address recipient, uint256 amount) external returns (bool);
    function allowance(address owner, address spender) external view returns (uint256);
    function approve(address spender, uint256 amount) external returns (bool);
    function transferFrom(address sender, address recipient, uint256 amount) external returns (bool);
    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);
}
```

SafeMath.sol

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue

pragma solidity ^0.5.0;

/**
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises an
 * error, which is the standard behavior in high level programming languages.
 * `SafeMath` restores this intuition by reverting the transaction when an
 * operation overflows.
 *
 * Using this library instead of the unchecked operations eliminates an entire
 * class of bugs, so it's recommended to use it always.
 */
//SlowMist// OpenZeppelin's SafeMath security Module is used, which is a recommend approach
library SafeMath {
    /**
     * @dev Returns the addition of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `+` operator.
     *
     * Requirements:
     * - Addition cannot overflow.
     */
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting on
     * overflow (when the result is negative).
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     * - Subtraction cannot overflow.
     */
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
     * overflow (when the result is negative).
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     * - Subtraction cannot overflow.
     *
     * _Available since v2.4.0._
     */
    function sub(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b <= a, errorMessage);
        uint256 c = a - b;

        return c;
    }
```

```
/**
 * @dev Returns the multiplication of two unsigned integers, reverting on
 * overflow.
 *
 * Counterpart to Solidity's `*` operator.
 *
 * Requirements:
 * - Multiplication cannot overflow.
 */
function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
    // benefit is lost if 'b' is also tested.
    // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
    if (a == 0) {
        return 0;
    }

    uint256 c = a * b;
    require(c / a == b, "SafeMath: multiplication overflow");

    return c;
}

/**
 * @dev Returns the integer division of two unsigned integers. Reverts on
 * division by zero. The result is rounded towards zero.
 *
 * Counterpart to Solidity's `/` operator. Note: this function uses a
 * `revert` opcode (which leaves remaining gas untouched) while Solidity
 * uses an invalid opcode to revert (consuming all remaining gas).
 *
 * Requirements:
 * - The divisor cannot be zero.
 */
function div(uint256 a, uint256 b) internal pure returns (uint256) {
    return div(a, b, "SafeMath: division by zero");
}

/**
 * @dev Returns the integer division of two unsigned integers. Reverts with custom message on
 * division by zero. The result is rounded towards zero.
 *
```

```
     * Counterpart to Solidity's `/` operator. Note: this function uses a
     * `revert` opcode (which leaves remaining gas untouched) while Solidity
     * uses an invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     * - The divisor cannot be zero.
     *
     * _Available since v2.4.0._
     */
    function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        // Solidity only automatically asserts when dividing by 0
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts with custom message when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
```

```
     * - The divisor cannot be zero.
     *
     * _Available since v2.4.0._
     */
    function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }
}
```

ZildFinance.sol

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity 0.5.4;

import 'SafeMath.sol';
import 'Ownable.sol';
import 'IERC20.sol';

contract ZildFinanceCoin is Ownable, IERC20 {

    using SafeMath for uint256;

    string public constant name = 'Zild Finance Coin';
    string public constant symbol = 'Zild';
    uint8 public constant decimals = 18;
    uint256 public totalSupply = 9980 * 10000 * 10 ** uint256(decimals);
    uint256 public allowBurn = 2100 * 10000 * 10 ** uint256(decimals);
    uint256 public tokenDestroyed;

    uint256 public constant FounderAllocation = 1497 * 10000 * 10 ** uint256(decimals);
    uint256 public constant FounderLockupAmount = 998 * 10000 * 10 ** uint256(decimals);
    uint256 public constant FounderLockupCliff = 365 days;
    uint256 public constant FounderReleaseInterval = 30 days;
    uint256 public constant FounderReleaseAmount = 20.7916 * 10000 * 10 ** uint256(decimals);
    uint256 public constant MarketingAllocation = 349 * 10000 * 10 ** uint256(decimals);
    uint256 public constant FurnaceAllocation = 150 * 10000 * 10 ** uint256(decimals);

    address public founder = address(0);
    uint256 public founderLockupStartTime = 0;
    uint256 public founderReleasedAmount = 0;
```

```
mapping (address => uint256) private _balances;
mapping (address => mapping (address => uint256)) private _allowances;
mapping (address => bool) public frozenAccount;

event Transfer(address indexed from, address indexed to, uint256 value);
event Approval(address indexed from, address indexed to, uint256 value);
event ChangeFounder(address indexed previousFounder, address indexed newFounder);
event SetMinter(address indexed minter);
event SetMarketing(address indexed marketing);
event SetFurnace(address indexed furnace);
event Burn(address indexed _from, uint256 _tokenDestroyed, uint256 _timestamp);
event FrozenFunds(address target, bool frozen);

constructor(address _founder, address _marketing) public {
    require(_founder != address(0), "ZildFinanceCoin: founder is the zero address");
    require(_marketing != address(0), "ZildFinanceCoin: operator is the zero address");
    founder = _founder;
    founderLockupStartTime = block.timestamp;
    _balances[address(this)] = totalSupply;
    _transfer(address(this), _marketing, MarketingAllocation);
}
```

//SlowMist// founder release

```
function release() public {
    uint256 currentTime = block.timestamp;
    uint256 cliffTime = founderLockupStartTime.add(FounderLockupCliff);
    if (currentTime < cliffTime) return;
    if (founderReleasedAmount >= FounderLockupAmount) return;
    uint256 month = currentTime.sub(cliffTime).div(FounderReleaseInterval);
    uint256 releaseAmount = month.mul(FounderReleaseAmount);
    if (releaseAmount > FounderLockupAmount) releaseAmount = FounderLockupAmount;
    if (releaseAmount <= founderReleasedAmount) return;
    uint256 amount = releaseAmount.sub(founderReleasedAmount);
    founderReleasedAmount = releaseAmount;
    _transfer(address(this), founder, amount);
}

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

function transfer(address to, uint256 amount) public returns (bool) {
        require(to != address(0), "ERC20: tranfer to the zero address");
        require(!frozenAccount[msg.sender]);
        require(!frozenAccount[to]);
        _transfer(msg.sender, to, amount);
        return true; //SlowMist// The return value conforms to the EIP20 specification
    }

    function burn(uint256 _value) public returns (bool){
        _burn(msg.sender, _value);
        return true;
    }

    function _burn(address _who, uint256 _burntAmount) internal {
        require (tokenDestroyed.add(_burntAmount) <= allowBurn, "ZildFinanceCoin: exceeded the maximum allowable
burning amount" );
        require(_balances[msg.sender] >= _burntAmount && _burntAmount > 0);
        _transfer(address(_who), address(0), _burntAmount);
        totalSupply = totalSupply.sub(_burntAmount);
        tokenDestroyed = tokenDestroyed.add(_burntAmount);
        emit Burn(_who, _burntAmount, block.timestamp);
    }


    function allowance(address from, address to) public view returns (uint256) {
        return _allowances[from][to];
    }

    function approve(address to, uint256 amount) public returns (bool) {
        _approve(msg.sender, to, amount);
        return true; //SlowMist// The return value conforms to the EIP20 specification
    }

    function transferFrom(address from, address to, uint256 amount) public returns (bool) {
        uint256 remaining = _allowances[from][msg.sender].sub(amount, "ERC20: transfer amount exceeds allowance");
        require(to != address(0), "ERC20: tranfer to the zero address");
        require(!frozenAccount[from]);
        require(!frozenAccount[to]);
        require(!frozenAccount[msg.sender]);
        _transfer(from, to, amount);
        _approve(from, msg.sender, remaining);
        return true; //SlowMist// The return value conforms to the EIP20 specification
    }

    function _transfer(address from, address to, uint256 amount) private {
        require(from != address(0), "ERC20: transfer from the zero address");
        _balances[from] = _balances[from].sub(amount, "ERC20: transfer amount exceeds balance");
        _balances[to] = _balances[to].add(amount);
        emit Transfer(from, to, amount);
    }

    function _approve(address from, address to, uint256 amount) private {
        require(from != address(0), "ERC20: approve from the zero address");
        require(to != address(0), "ERC20: approve to the zero address");
        _allowances[from][to] = amount;
        emit Approval(from, to, amount);
    }

    function changeFounder(address _founder) public onlyOwner {
        require(_founder != address(0), "ZildFinanceCoin: founder is the zero address");
        emit ChangeFounder(founder, _founder);
        founder = _founder;
    }
    //SlowMist// The owner can set any minter and transfer the tokens to the minter, but only once
    function setMinter(address minter) public onlyOwner {
        require(minter != address(0), "ZildFinanceCoin: minter is the zero address");
        require(_balances[minter] == 0, "ZildFinanceCoin: minter has been initialized");
        _transfer(address(this), minter, totalSupply.sub(FounderAllocation));
        emit SetMinter(minter);
    }
    //SlowMist// The owner can set any minter and transfer the tokens to the furnace, but only once
    function setFurnace(address furnace) public onlyOwner {
        require(furnace != address(0), "ZildFinanceCoin: furnace is the zero address");
        require(_balances[furnace] == 0, "ZildFinanceCoin: furnace has been initialized");
        _transfer(address(this), furnace, FurnaceAllocation);
        emit SetFurnace(furnace);
    }
    //SlowMist// Owner can freeze and unfreeze any account
    function freezeAccount(address _target, bool _bool) public onlyOwner {
        if (_target != address(0)) {
            frozenAccount[_target] = _bool;
            emit FrozenFunds(_target,_bool);
        }
    }
}
```