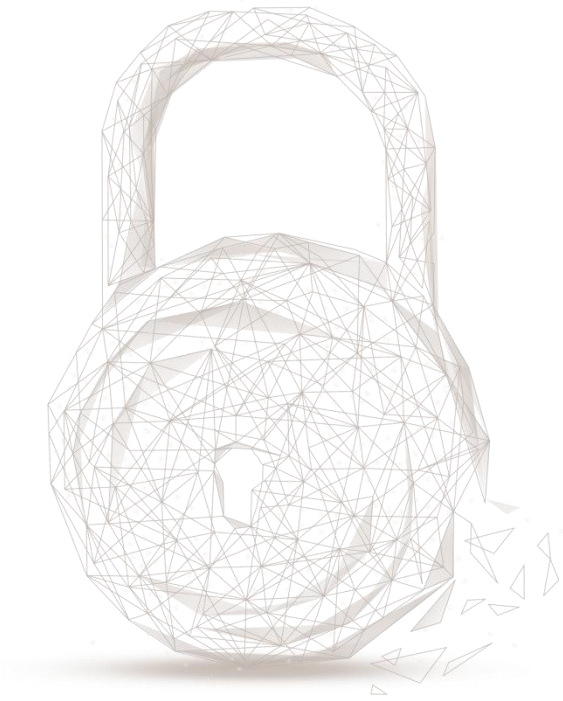
成都链安
BEOSIN

Smart contract security audit report



Audit Number：202009301805

Report Query Name: spaceswap

Smart Contract Name And Address Link：

Smart Contract Name	Smart Contract Address Link
MilkyWayToken.sol	https://etherscan.io/address/0x80c8c3dcfb854f9542567c8dac3f44d709ebc1de#code
Blender.sol	https://etherscan.io/address/0x19b911d1bedcbe6ba3efc372f4ae69710426d85b#code
ShakeToken.sol	https://etherscan.io/address/0x6006fc2a849fedaba8330ce36f5133de01f96189#code
Interstellar.sol	https://etherscan.io/address/0xb95ebbf2a9fc64e4dc4d6951a60bc4d3c8f55b9d#code
Timelock.sol	https://etherscan.io/address/0xa17809ce669594dc13b0f218cad87e445bb4d770#code

Start Date：2020.09.24

Completion Date：2020.09.30

0Audit Team: Beosin (Chengdu LianAn) Technology Co. Ltd.

Audit Categories and Results:

No.	Categories	Subitems	Results
1	Coding Conventions	Compiler Version Security	Pass
		Deprecated Items	Pass
		Redundant Code	Pass
		SafeMath Features	Pass
		require/assert Usage	Pass
		Gas Consumption	Pass
		Visibility Specifiers	Pass
		Fallback Usage	Pass
2	General Vulnerability	Integer Overflow/Underflow	Pass
		Reentrancy	Pass
		Pseudo-random Number Generator (PRNG)	Pass
		Transaction-Ordering Dependence	Pass
		DoS (Denial of Service)	Pass

		Access Control of Owner	Pass
		Low-level Function (call/delegatecall) Security	Pass
		Returned Value Security	Pass
		tx.origin Usage	Pass
		Replay Attack	Pass
		Overriding Variables	Pass
3	Business Security	Business Logics	Pass
		Business Implementations	The results of the analysis in 3.3(3)

Note: Audit results and suggestions in code comments

Audit Results Explained:

Beosin (Chengdu LianAn) Technology has used several methods including Formal Verification, Static Analysis, Typical Case Testing and Manual Review to audit three major aspects of smart contracts Blender, Interstellar, MilkyWayToken, ShakeToken and Timelock, including Coding Standards, Security, and Business Logic. **The Interstellar, MilkyWayToken, ShakeToken and Timelock contracts pass most audit items. The overall result is Pass. The smart contract is able to function properly.**

1. Coding Conventions

Check the code style that does not conform to Solidity code style.

1.1 Compiler Version Security

- Description: Check whether the code implementation of current contract contains the exposed solidity compiler bug.
- Result: Pass

1.2 Deprecated Items

- Description: Check whether the current contract has the deprecated items.
- Result: Pass

1.3 Redundant Code

- Description: Check whether the contract code has redundant codes.
- Result: Pass

1.4 SafeMath Features

- Description: Check whether the SafeMath has been used. Or prevents the integer overflow/underflow in mathematical operation.
- Result: Pass

1.5 require/assert Usage

- Description: Check the use reasonability of 'require' and 'assert' in the contract.
- Result: Pass

1.6 Gas Consumption

- Description: Check whether the gas consumption exceeds the block gas limitation.
- Result: Pass

1.7 Visibility Specifiers

- Description: Check whether the visibility conforms to design requirement.
- Result: Pass

1.8 Fallback Usage

- Description: Check whether the Fallback function has been used correctly in the current contract.
- Result: Pass

2. General Vulnerability

Check whether the general vulnerabilities exist in the contract.

2.1 Integer Overflow/Underflow

- Description: Check whether there is an integer overflow/underflow in the contract and the calculation result is abnormal.
- Result: Pass

2.2 Reentrancy

- Description: An issue when code can call back into your contract and change state, such as withdrawing ETH.
- Result: Pass

2.3 Pseudo-random Number Generator (PRNG)

- Description: Whether the results of random numbers can be predicted.
- Result: Pass

2.4 Transaction-Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Pass

2.5 DoS (Denial of Service)

- Description: Whether exist DoS attack in the contract which is vulnerable because of unexpected reason.
- Result: Pass

2.6 Access Control of Owner

- Description: Whether the owner has excessive permissions, such as malicious issue, modifying the balance of others.
- Result: Pass

2.7 Low-level Function (call/delegatecall) Security

- Description: Check whether the usage of low-level functions like call/delegatecall have vulnerabilities.
- Result: Pass

2.8 Returned Value Security

- Description: Check whether the function checks the return value and responds to it accordingly.
- Result: Pass

2.9 tx.origin Usage

- Description: Check the use secure risk of 'tx.origin' in the contract.
- Result: Pass

2.10 Replay Attack

- Description: Check the weather the implement possibility of Replay Attack exists in the contract.
- Result: Pass

2.11 Overriding Variables

- Description: Check whether the variables have been overridden and lead to wrong code execution.
- Result: Pass

3. Business Security

Check whether the business is secure.

3.1 Business analysis of Contract MilkyWayToken and ShakeToken

(1) Basic Token Information

Token name	MilkyWay Token by SpaceSwap v2
Token symbol	MILK2
decimals	18
totalSupply	Initial supply is 0 (Mintable , without maximum token total supply)
Token type	ERC20

Table 1 Basic Token Information of MILK2

Token name	SHAKE token by SpaceSwap v2
Token symbol	SHAKE
decimals	18
totalSupply	Initial supply is 0 (Mintable , maximum token total supply is 10000)
Token type	ERC20

Table 2 Basic Token Information of SHAKE

(2) ERC20 Token Standard Functions

- Description: The MilkyWayToken and ShakeToken Contracts both implement a Token which conforms to the ERC20 Standards.It should be noted that the user can directly call the approve function to set the approval value for the specified address, but in order to avoid multiple authorizations, it is recommended to use the increaseAllowance and decreaseAllowance functions when modifying the approval value, instead of using the approve function directly.
- Related functions: *name, symbol, decimals, totalSupply, balanceOf, allowance, transfer, transferFrom, approve, increaseAllowance, decreaseAllowance. burn*
- Result: Pass

(3) mint function and mint authority management

- Description: As shown in Figure 1 and 2 below, the user or contract with mint or governance permission can call *mint* function to mint tokens to the specified address. The maximum token total supply of MILK2 is unlimited and the maximum token total supply of SHAKE is 10000. The contract owner can set the minter of SHAKE and the governance contract address of MILK2.

```
826  /// @notice Creates `_amount` token to `_to`. Must only be called by the  Governance Contracts
827  function mint(address _to, uint256 _amount) public onlyGovernanceContracts virtual returns (bool) {
828      _mint(_to, _amount);
829      _moveDelegates(address(0), _delegates[_to], _amount);
830      return true;
831  }
```

Figure 1 mint Function Source Code (MilkyWayToken.sol)

```
615  /**
616  * @dev Function to mint tokens
617  * @param to The address that will receive the minted tokens.
618  * @param value The amount of tokens to mint.
619  * @return A boolean that indicates if the operation was successful.
620  */
621  function mint(address to, uint256 value) public onlyMinter returns (bool) {
622      require(totalSupply().add(value) <= MAX_TOTAL_SUPPLY, "Can`t mint more than MAX_TOTOAL_SUPPLY");
623      _mint(to, value);
624      totalMinted = totalMinted.add(value);
625      return true;
626  }
```

Figure 2 mint Function Source Code (ShakeToken.sol)

- Related functions: *mint, updatePool, balanceOf, getTotalReward*
- Result: Pass

(4) delegate and delegateBySig function of Contract MilkyWayToken

- Description: As shown in Figure 3 and 4 below, the contract implements the delegate and delegateBySig functions to delegate. The user can call those functions to delegate. The function delegate updates the delegate information by calling internal functions *_delegate*, *_moveDelegates* and *_writeCheckpoint*. There is the problem of using the same funds to repeatedly swipe votes. After an account vote to the delegatee, the current token balance of the delegator is recorded as the number of votes. After the vote, the account could transfer its own tokens to another account and vote to the same delegatee again. There is the case that the same fund is used for multiple valid votes and the current total number of votes exceeds the total token supply. It is recommended that users decide the token amount used in votes and lock the corresponding number of tokens. After communicating with the project party, they stated that this does not affect their normal operations.

```
893  function delegate(address delegatee) external {
894      return _delegate(msg.sender, delegatee);
895  }
```

Figure 3 delegate Function Source Code (MilkyWayToken.sol)

```
906  function delegateBySig(
907      address delegatee,
908      uint nonce,
909      uint expiry,
910      uint8 v,
911      bytes32 r,
912      bytes32 s
913  )
914  external
915  {
916      bytes32 domainSeparator = keccak256(
917          abi.encode(
918              DOMAIN_TYPEHASH,
919              keccak256(bytes(name())),
920              getChainId(),
921              address(this)
922          )
923      );
924
925      bytes32 structHash = keccak256(
926          abi.encode(
927              DELEGATION_TYPEHASH,
928              delegatee,
929              nonce,
930              expiry
931          )
932      );
933
934      bytes32 digest = keccak256(
935          abi.encodePacked(
936              "\x19\x01",
937              domainSeparator,
938              structHash
939          )
940      );
941
942      address signatory = ecrecover(digest, v, r, s);
943      require(signatory != address(0), "MILKYWAY::delegateBySig: invalid signature");
944      require(nonce == nonces[signatory]++, "MILKYWAY::delegateBySig: invalid nonce");
945      require(now <= expiry, "MILKYWAY::delegateBySig: signature expired");
946      return _delegate(signatory, delegatee);
947  }
```

Figure 4 delegateBySig Function Source Code (MilkyWayToken.sol)

- Related functions: *_delegate*, *_moveDelegates, _writeCheckpoint, safe32*
- Result: Pass

(5) reclaimToken function of Contract ShakeToken

- Description: As shown in Figure 5 below, the contract implements the reclaimToken functions to claim token. Minter can claim any tokens that transfered to this contract address.

```
643     function reclaimToken(ERC20 token) external onlyMinter {
644         require(address(token) != address(0));
645         uint256 balance = token.balanceOf(address(this));
646         token.transfer(msg.sender, balance);
647     }
```

Figure 5 reclaimToken Function Source Code

- Related functions: None
- Result: Pass

3.2 Business analysis of Contract Blender

(1) getOneShake Function

- Description: As shown in Figure 6 below, the contract implements the *getOneShake* function to get SHAKE tokens. Users can use a certain amount of MILK2 tokens to exchange for a SHAKE token by calling the function *getOneShake*. The MILK2 tokens paid by the user will be directly destroyed, and the Blender contract will mint a SHAKE token for the user by calling the mint of the SHAKE token contract. Every time the function *getOneShake* is called, the current SHAKE price increases by 10*10**18 MILK2 tokens.

```
286     function getOneShake() external {
287         require(block.number >= START_FROM_BLOCK, "Please wait for start block");
288         require(block.number < END_AT_BLOCK, "Sorry, it's too late");
289
290         IERC20 milk2Token = IERC20(MILK_ADDRESS);
291
292         require(milk2Token.balanceOf(msg.sender) >= currShakePrice, "There is no enough MILK2");
293         require(milk2Token.burn(msg.sender, currShakePrice), "Can't burn your MILK2");
294
295         IERC20 shakeToken = IERC20(SHAKE_ADDRESS);
296         currShakePrice  = currShakePrice.add(SHAKE_PRICE_STEP);
297         shakeToken.mint(msg.sender, 1*10**18);
298
299     }
```

Figure 6 getOneShake Function Source Code

- Related functions: None
- Result: Pass

(2) getMilkForShake Function

- Description: As shown in Figure 7 below, the contract implements the *getMilkForShake* function to get MILK2 tokens. Users can use a certain amount of SHAKE tokens to exchange for MILK2 tokens by calling the function *getMilkForShake*. The SHAKE tokens paid by the user will be directly destroyed, and the Blender contract will mint a certain amount of MILK2 token for the user by calling the mint of the MILK2 token contract.

```
311  function getMilkForShake(uint16 _amount) external {
312      require(block.number >= START_FROM_BLOCK, "Please wait for start block");
313      require(block.number < END_AT_BLOCK, "Sorry, it's too late");
314
315      IERC20 shakeToken = IERC20(SHAKE_ADDRESS);
316
317      require(shakeToken.balanceOf(msg.sender) >= uint256(_amount)*10**18, "There is no enough SHAKE");
318      require(shakeToken.burn(msg.sender, uint256(_amount)*10**18), "Can't burn your SHAKE");
319
320      IERC20 milk2Token = IERC20(MILK_ADDRESS);
321      milk2Token.mint(msg.sender, uint256(_amount).mul(currShakePrice.sub(SHAKE_PRICE_STEP)));
322
323  }
324
325  }
```

Figure 7 getMilkForShake Function Source code

- Related functions: None
- Result: Pass

3.3 Business analysis of Contract Interstellar

(1) addPool Function

- Description: As shown in Figure 8 below, the contract implements the add function to add the Pool. The contract owner can call this function to add the Pool for the user to stake for getting the reward and store the pool-related information.

```
1478    // Add a new lp to the pool. Can only be called by the owner.
1479    // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
1480    function add(uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate) public onlyOwner {
1481        if (_withUpdate) {
1482            massUpdatePools();
1483        }
1484        uint256 lastRewardBlock = block.number > startFirstPhaseBlock ? block.number : startFirstPhaseBlock;
1485        totalAllocPoint = totalAllocPoint.add(_allocPoint);
1486        poolInfo.push(PoolInfo({
1487        lpToken: _lpToken,
1488        allocPoint: _allocPoint,
1489        lastRewardBlock: lastRewardBlock,
1490        accMilkPerShare: 0
1491        }));
1492    }
```

Figure 8 addPool Function Source Code

- Related functions: *add*
- Result: Pass

(2) set Function

- Description: As shown in Figure 9 below, contract implements *set* function to set the reward allocation point of the specified pool, the contract owner can call this function to set the reward allocation point of the specified pool. After the pool reward allocation point is modified, it will affect the value of MILK rewards when users withdraw or deposit tokens.

```
1494    // Update the given pool's MILK allocation point. Can only be called by the owner.
1495    function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner {
1496        if (_withUpdate) {
1497            massUpdatePools();
1498        }
1499        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
1500        poolInfo[_pid].allocPoint = _allocPoint;
1501    }
```

Figure 9 set Function Source Code

(3) getMultiplier Function

- Description: As shown in Figure 10 below, The contract implements the getMultiplier function to calculate the MILK reward generated from the last update block to the current block. If the current block is less than startFirstPhaseBlock, the reward for each block is 1 times the original set reward. If the current block is less than startSecondPhaseBlock, the reward for each block is 20 times the original set reward. If the current block is smaller than startThirdPhaseBlock, the reward for each block is 10 times the original set reward. If the current block is smaller than bonusEndBlock, the reward for each block is 5 times the original set reward. If the last updated block is larger than bonusEndBlock, the reward for each block is 1 times the original set reward. In other cases, the total reward is 20 times the originally set reward from the last updated block to bonusEndBlock plus 1 times the originally set reward from bonusEndBlock to the current block. But this function has logic implementation problems.For example, when the user is depositing tokens, the last updated block is just less than startSecondPhaseBlock, and the current block is greater than startSecondPhaseBlock and less than startThirdPhaseBlock, the user's reward will be calculated 10 times reward, however, ignored 20 times reward from the user deposit block to startSecondPhaseBlock(No other users deposit/withdraw and call function updatePool during the period). After communicating with the project party, they stated that this does not affect their normal operations.

```
1504    // Return reward multiplier over the given _from to _to block.
1505    function getMultiplier(uint256 _from, uint256 _to) public view returns (uint256) {
1506        if (_to <= startFirstPhaseBlock) { // 0
1507            return  _to.sub(_from); // x1
1508        }
1509        else if (_to <= startSecondPhaseBlock) { // + 10,000 blocks
1510            return _to.sub(_from).mul(BONUS_MULTIPLIER_1); // x20
1511        }
1512        else if (_to <= startThirdPhaseBlock) { // + 40,000 blocks
1513            return _to.sub(_from).mul(BONUS_MULTIPLIER_2); //x10
1514        }
1515        else if (_to <= bonusEndBlock) { // + 40,000 blocks
1516            return _to.sub(_from).mul(BONUS_MULTIPLIER_3); // x5
1517        }
1518        else if (_from >= bonusEndBlock) { // + 100,000 blocks
1519            return _to.sub(_from);
1520        }
1521        else {
1522            return bonusEndBlock.sub(_from).mul(BONUS_MULTIPLIER_1).add( // todo ????
1523                _to.sub(bonusEndBlock)
1524            );
1525        }
1526    }
```

Figure 10 getMultiplier Function Source code

- Related functions: getMultiplier
- Safety advice: It is recommended that the project party modify the MILK reward logic, because the current logic may affect the amount of rewards users will eventually receive.
- Result: Fail

(4) updatePool Function

- Description: As shown in Figure 11 and 12 below, contract implementation *updatePool* function to update pool MILK rewards and information of current block. Any user can call this function to update latest pool MILK rewards and information, and call *mint* function to mint all MILK rewards generated after last block update to this contract address.At the same time, the devAddr and distributor addresses will receive an additional 3% and 1% of the MILK reward respectively(the MilkyWayToken contract owner must be this contract address).

```
1550    // Update reward variables of the given pool to be up-to-date.
1551    function updatePool(uint256 _pid) public {
1552        PoolInfo storage pool = poolInfo[_pid];
1553        if (block.number <= pool.lastRewardBlock) {
1554            return;
1555        }
1556        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1557        if (lpSupply == 0) {
1558            pool.lastRewardBlock = block.number;
1559            return;
1560        }
1561        uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
1562        uint256 milkReward = multiplier.mul(milkPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
1563        milk.mint(devAddr, milkReward.mul(3).div(100)); // 3% developers
1564        milk.mint(distributor, milkReward.div(100)); // 1% shakeHolders
1565        milk.mint(address(this), milkReward);
1566        pool.accMilkPerShare = pool.accMilkPerShare.add(milkReward.mul(1e12).div(lpSupply));
1567        pool.lastRewardBlock = block.number;
1568    }
```

Figure 11 updatePool Function Source Code

```
822     /// @notice Creates `_amount` token to `_to`. Must only be called by the  Governance Contracts
823     function mint(address _to, uint256 _amount) public onlyGovernanceContracts virtual returns (bool) {
824         _mint(_to, _amount);
825         _moveDelegates(address(0), _delegates[_to], _amount);
826         return true;
827     }
```

Figure 12 mint Function Source Code

- Related functions: *updatePool, balanceOf, getMultiplier, mint*
- Result: Pass

(5) deposit Function

- Description: As shown in Figure 13 below, the contract implements the *deposit* function for users to stake tokens, the user pre-approves this contract address and then calls this function to deposit tokens(require the pool is exist). Update the pool information when the user is deposited, if the user has previous deposit, calculate the user's previous deposit reward and send the reward to the user address.

```
1570    // Deposit LP tokens to Interstellar for MILK allocation.
1571    function deposit(uint256 _pid, uint256 _amount) public {
1572        PoolInfo storage pool = poolInfo[_pid];
1573        UserInfo storage user = userInfo[_pid][msg.sender];
1574        updatePool(_pid);
1575        if (user.amount > 0) {
1576            uint256 pending = user.amount.mul(pool.accMilkPerShare).div(1e12).sub(user.rewardDebt);
1577            safeMilkTransfer(msg.sender, pending);
1578        }
1579        pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
1580        user.amount = user.amount.add(_amount);
1581        user.rewardDebt = user.amount.mul(pool.accMilkPerShare).div(1e12);
1582        emit Deposit(msg.sender, _pid, _amount);
1583    }
```

Figure 13 deposit Function Source Code

- Related functions: *deposit, updatePool, safeMilkTransfer, safeTransferFrom*
- Result: Pass

(6) withdraw Function

- Description: As shown in Figure 14 below, the contract implements the withdraw function for users to withdraw deposit tokens and MILK rewards, the user can call this function to withdraw the specified amount of deposit tokens and all MILK reward in the current block . Update pool information when users withdraw deposit tokens and MILK rewards, and transfer the specified deposit tokens and MILK rewards to the user address and update the user deposit information.

```
1585    // Withdraw LP tokens from Interstellar.
1586    function withdraw(uint256 _pid, uint256 _amount) public {
1587        PoolInfo storage pool = poolInfo[_pid];
1588        UserInfo storage user = userInfo[_pid][msg.sender];
1589        require(user.amount >= _amount, "withdraw: not good");
1590        updatePool(_pid);
1591        uint256 pending = user.amount.mul(pool.accMilkPerShare).div(1e12).sub(user.rewardDebt);
1592        safeMilkTransfer(msg.sender, pending);
1593        user.amount = user.amount.sub(_amount);
1594        user.rewardDebt = user.amount.mul(pool.accMilkPerShare).div(1e12);
1595        pool.lpToken.safeTransfer(address(msg.sender), _amount);
1596        emit Withdraw(msg.sender, _pid, _amount);
1597    }
```

Figure 14 withdraw Function Source Code

- Related functions: *withdraw, safeMilkTransfer, safeTransfer*
- Result: Pass

(7) emergencyWithdraw Function

- Description: As shown in Figure 15 below, the contract implements the *emergencyWithdraw* function for users to withdraw deposited tokens, the user can call this function to withdraw all deposited tokens . Update user deposit information and transfer all deposited tokens to the user address(Note: calling this function cannot get any deposit rewards).

```
1599    // Withdraw without caring about rewards. EMERGENCY ONLY.
1600    function emergencyWithdraw(uint256 _pid) public {
1601        PoolInfo storage pool = poolInfo[_pid];
1602        UserInfo storage user = userInfo[_pid][msg.sender];
1603        pool.lpToken.safeTransfer(address(msg.sender), user.amount);
1604        emit EmergencyWithdraw(msg.sender, _pid, user.amount);
1605        user.amount = 0;
1606        user.rewardDebt = 0;
1607    }
```

Figure 15 emergencyWithdraw Function Source Code

4. Conclusion

Beosin(Chengdu LianAn) conducted a detailed audit on the design and code implementation of the smart contracts Blender, Interstellar, MilkyWayToken, ShakeToken and Timelock. All problems found during the audit have been notified to the project party, and the project party believes that no repair is needed. The owner of the Interstellar contract is the Timelock contract. During the delay time when the contract owner calls the function to add/set to perform pool-related operations or adding/removing issuer permissions, users can withdraw their deposited tokens and corresponding rewards. In addition, the MilkyWayToken token contract has the problem of unlimited minting, and the ShakeToken token contract has the problem of minter minting. In the MilkyWayToken contract, the owner (0x81cfe8efdb6c7b7218ddd5f6bda3aa4cd1554fd2) can call the addAddressToGovernanceContract function to add addresses to the governmentContracts list, and the addresses in the governmentContracts list can call the mint function to mint tokens; the ShakeToken contract

deployer (0x81cfe8efdb6c7b7218ddd5f6bda3aa4cd1554fd2) is initially minter, and minter can add new minters and call mint function to mint tokens. Finally, the reward mechanism of the contract has logic implementation problems, which may affect the MILK rewards users actually get. The overall audit result of the smart contracts Blender, Interstellar, MilkyWayToken, ShakeToken and Timelock is **Pass**.

成都链安

BEOSIN

Official Website

https://lianantech.com

E-mail

vaas@lianantech.com

Twitter

https://twitter.com/Beosin_com