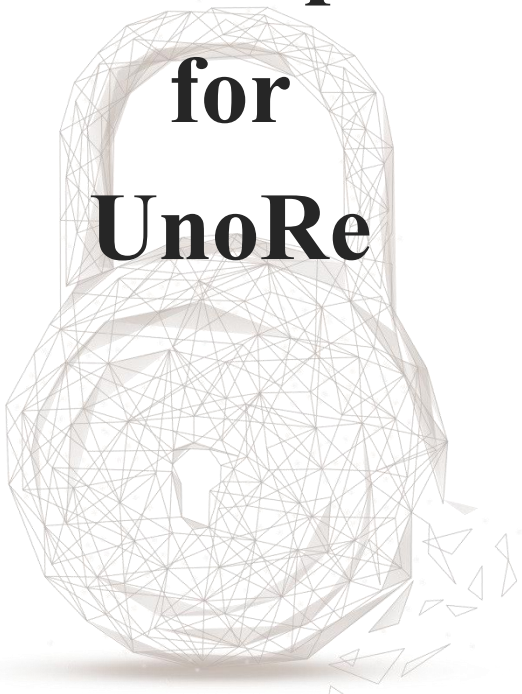
BEOSIN
Blockchain Security

Smart contract security audit report

for

UnoRe



Audit Number: 202112141130

Project Name: UnoRe

Deployment Platform: Ethereum

Audit Files Link: https://github.com/UnoRe/SSIP/

Commit Hash:

e23fdc8d2c939409a7b0d0efcfe41482c4909c76 (Initial)

1c1aebf85b8b09d393a8c6e58287a247c455ea03 (Final)

Audit Start Date: 2021.11.18

Audit Completion Date: 2021.12.14

Audit Result: Pass

Audit Team: Beosin Technology Co. Ltd.

Audit Results Explained

Beosin Technology has used several methods including Formal Verification, Static Analysis, Typical Case Testing and Manual Review to audit three major aspects of UnoRe project, including Coding Conventions, General Vulnerability and Business Security. **After auditing, the UnoRe project was found to have 20 risk items: 2 High-risks, 8 Medium-risks, 6 Low-risks, 4 Info-risks. As of the completion of the audit, all risk items have been fixed or properly handled. The overall result of the UnoRe project is Pass.** The following is the detailed audit information for this project.

Index	Risk description	Risk level	Fix results
SSRP-1	*leaveFromPeding* function implementation defects	High	Fixed
SSRP-2	Risk of duplication of rewards	High	Fixed
SSRP-3	rewardDebt update error	Medium	Fixed
SSRP-4	The *cancelWithdrawReques* function is missing permission checks	Medium	Fixed
SSRP-5	*enterInPool* function implementation defects	Medium	Fixed
SSRP-6	riskPool data coverage risk	Low	Fixed
SSRP-7	initialRewarder function implementation defects	Low	Fixed
EA-1	The *addDex* function is missing a permission check	Medium	Fixed
EA-2	Recommended fixed compiler version	Info	Fixed
PP-1	The *withdrawPremium* function has too much authority	Medium	Acknowledged
SP-1	*buyPolicy* function implementation defects	Medium	Fixed
SP-2	Unused ERC1155 tokens	Info	Acknowledged
SP-3	BuyPolicy event triggers parameter error	Info	Fixed
RW-1	*onUnoReward* function implementation defects	Medium	Fixed
RW-2	Error on *balanceOf* function call with parameters	Medium	Fixed
MT-1	The *onMigration* function is not implemented	Low	Acknowledged
RP-1	*policyClaim* function implementation defects	Low	Fixed
RP-2	The *transferFrom* and *transfer* functions are flawed in their judgement	Low	Fixed
RP-3	The *leaveFromPending* function has a flawed judgment	Low	Fixed
RP-4	Redundant codes	Info	Fixed

Table 1. Risk Statistics

Risk explained:

- Item PP-1 is not fixed and may cause the owner of the PremiumPool contract to be able to withdraw any asset within the contract (Normally only the premium paid by the user for the insurance will be sent to this contract).
- Item SP-2 is not fixed and may cause the minted ERC1155 tokens useless.
- Item MT-1 is not fixed and may cause *migrate* function calls to fail.

Risk descriptions and fix results explained

[SSRP-1 High] *leaveFromPeding* function implementation defects

- Description: The *_withdrawImplement* function has changed the lp token balance of the _to address, and the return result of 'balanceOf(_to)' may be less than the pendingAmount. The determination of the balance of the _to address should be placed before the lp tokens are burned.

```
63    function leaveFromPending(address _to) external override onlySSIP returns (uint256, uint256) {
64        uint256 cryptoBalance = IERC20(currency).balanceOf(address(this));
65        uint256 pendingAmount = uint256(withdrawRequestPerUser[_to].pendingAmount);
66        _withdrawImplement(_to);
67        require(cryptoBalance > 0, "UnoRe: zero uno balance");
68        require(balanceOf(_to) >= pendingAmount, "UnoRe: lp balance overflow");
69        uint256 pendingAmountInUno = (pendingAmount * lpPriceUno) / 1e18;
70        if (pendingAmountInUno < cryptoBalance) {
71            TransferHelper.safeTransfer(currency, _to, pendingAmountInUno);
72            emit LogLeaveFromPending(_to, pendingAmount, pendingAmountInUno);
73            return (pendingAmount, pendingAmountInUno);
74        } else {
75            TransferHelper.safeTransfer(currency, _to, cryptoBalance - MIN_LP_CAPITAL);
76            emit LogLeaveFromPending(_to, pendingAmount, cryptoBalance - MIN_LP_CAPITAL);
77            return (pendingAmount, cryptoBalance - MIN_LP_CAPITAL);
78        }
79    }
```

Figure 1 source code of *leaveFromPending* function (Unfixed)

- Fix recommendations: It is recommended to determine if the balance is sufficient before burning the lp tokens.
- Fix results: Fixed.

```
64    function leaveFromPending(address _to) external override onlySSRP returns (uint256, uint256) {
65        uint256 cryptoBalance = IERC20(currency).balanceOf(address(this));
66        uint256 pendingAmount = uint256(withdrawRequestPerUser[_to].pendingAmount);
67        require(cryptoBalance > 0, "UnoRe: zero uno balance");
68        require(balanceOf(_to) >= pendingAmount, "UnoRe: lp balance overflow");
69        _withdrawImplement(_to);
70        uint256 pendingAmountInUno = (pendingAmount * lpPriceUno) / 1e18;
71        if (pendingAmountInUno < cryptoBalance - MIN_LP_CAPITAL) {
72            TransferHelper.safeTransfer(currency, _to, pendingAmountInUno);
73            emit LogLeaveFromPending(_to, pendingAmount, pendingAmountInUno);
74            return (pendingAmount, pendingAmountInUno);
75        } else {
76            TransferHelper.safeTransfer(currency, _to, cryptoBalance - MIN_LP_CAPITAL);
77            emit LogLeaveFromPending(_to, pendingAmount, cryptoBalance - MIN_LP_CAPITAL);
78            return (pendingAmount, cryptoBalance - MIN_LP_CAPITAL);
79        }
80    }
```

Figure 2 source code of *leaveFromPending* function (Fixed)

[SSRP-2 High] Risk of duplication of rewards

- Description: After calling the *enterInPool* function in the SingleSidedInsurancePool contract to get the LP tokens, the user can transfer the LP to other addresses and then use the other addresses to call the harvest function to get a huge UNO reward.

```
309     function harvest(address _to) external override nonReentrant {
310         _harvest(_to);
311     }
312
313     function _harvest(address _to) private {
314         updatePool();
315         uint256 amount = IERC20(riskPool).balanceOf(_to);
316         uint256 accumulatedUno = (amount * uint256(poolInfo.accUnoPerShare)) / ACC_UNO_PRECISION;
317         uint256 _pendingUno = accumulatedUno - rewardDebt[_to];
318
319         // Effects
320         rewardDebt[_to] = accumulatedUno;
321
322         if (rewarder != address(0) && _pendingUno != 0) {
323             IRewarder(rewarder).onUnoReward(_to, _pendingUno);
324         }
325
326         emit Harvest(msg.sender, _to, _pendingUno);
327     }
```

Figure 3 source code of *harvest* and *_harvest* functions

- Fix recommendations: It is recommended to change it to not be able to transfer LP tokens.
- Fix results: Fixed. They want to keep LP tokens transferring in business logic. So they overridden ERC20 standard *transfer* and *transferFrom* functions of riskPool LP token and then added *lpTransfer* function in SSIP, where they adjust rewardDebt of sender and recipient. Of course, at that time, the pending reward till the transferring time of the sender will be transferred to the sender's wallet.

```
158     function transfer(address recipient, uint256 amount) external override returns (bool) {
159         require(
160             balanceOf(msg.sender) - uint256(withdrawRequestPerUser[msg.sender].pendingAmount) >= amount,
161             "ERC20: transfer amount exceeds balance or pending WR"
162         );
163         _transfer(msg.sender, recipient, amount);
164
165         ISingleSidedReinsurancePool(SSRP).lpTransfer(msg.sender, recipient, amount);
166         return true;
167     }
```

Figure 4 source code of *transfer* function

```
169     function transferFrom(
170         address sender,
171         address recipient,
172         uint256 amount
173     ) external override returns (bool) {
174         require(
175             balanceOf(sender) - uint256(withdrawRequestPerUser[sender].pendingAmount) >= amount,
176             "ERC20: transfer amount exceeds balance or pending WR"
177         );
178         _transfer(sender, recipient, amount);
179 
180         uint256 currentAllowance = _allowances[sender][msg.sender];
181         require(currentAllowance >= amount, "ERC20: transfer amount exceeds allowance");
182         _approve(sender, msg.sender, currentAllowance - amount);
183         ISingleSidedReinsurancePool(SSRP).lpTransfer(sender, recipient, amount);
184         return true;
185     }
```

Figure 5 source code of *transferFrom* function

```
323     function lpTransfer(
324         address _from,
325         address _to,
326         uint256 _amount
327     ) external override nonReentrant {
328         require(msg.sender == address(riskPool), "UnoRe: not allow others transfer");
329         updatePool();
330         uint256 amount = userInfo[_from].amount;
331         (uint256 pendingAmount, , ) = IRiskPool(riskPool).getWithdrawRequest(_from);
332         require(amount - pendingAmount >= _amount, "UnoRe: balance overflow");
333         uint256 accumulatedUno = (amount * uint256(poolInfo.accUnoPerShare)) / ACC_UNO_PRECISION;
334         uint256 _pendingUno = accumulatedUno - userInfo[_from].rewardDebt;
335         userInfo[_from].rewardDebt = accumulatedUno - ((_amount * uint256(poolInfo.accUnoPerShare)) / ACC_UNO_PRECISION);
336         if (rewarder != address(0)) {
337             uint256 rewardAmount = IRewarder(rewarder).onUnoReward(_from, _pendingUno);
338             emit Harvest(_from, _from, rewardAmount);
339         }
340         userInfo[_from].amount = amount - _amount;
341 
342         userInfo[_to].rewardDebt = userInfo[_to].rewardDebt + ((_amount * uint256(poolInfo.accUnoPerShare)) / ACC_UNO_PRECISION);
343         userInfo[_to].amount = userInfo[_to].amount + _amount;
344 
345         emit LogLpTransferInSSRP(_from, _to, _amount);
346     }
```

Figure 6 source code of *lpTransfer* function

[SSRP-3 Medium] rewardDebt update error

- Description: The *leaveFromPoolInPending* function in the SingleSidedInsurancePool contract calculates the accumulatedUnoRe using the number containing the pendingAmount, but updates the rewardDebt by subtracting the number added to the pendingAmount this time.

```
281     function leaveFromPoolInPending(address _to, uint256 _amount) external override nonReentrant {
282         require(_to == msg.sender, "UnoRe: Forbidden");
283         // Withdraw desired amount from pool
284         updatePool();
285         uint256 amount = IERC20(riskPool).balanceOf(msg.sender);
286         uint256 lpPriceUno = IRiskPool(riskPool).lpPriceUno();
287         (uint256 pendingAmount, , ) = IRiskPool(riskPool).getWithdrawRequest(msg.sender);
288         require(((amount - pendingAmount) * lpPriceUno) / 1e18 >= _amount, "UnoRe: withdraw amount overflow");
289         uint256 accumulatedUno = (amount * uint256(poolInfo.accUnoPerShare)) / ACC_UNO_PRECISION;
290         uint256 _pendingUno = accumulatedUno - rewardDebt[msg.sender];
291         rewardDebt[_to] =
292             accumulatedUno -
293             (((_amount * 1e18 * uint256(poolInfo.accUnoPerShare)) / lpPriceUno) / ACC_UNO_PRECISION);
294         IRiskPool(riskPool).leaveFromPoolInPending(msg.sender, _amount);
295         if (rewarder != address(0)) {
296             IRewarder(rewarder).onUnoReward(_to, _pendingUno, amount);
297         }
298
299         lastWithdrawTime[msg.sender] = block.timestamp;
300
301         emit Harvest(msg.sender, _to, _pendingUno);
302         emit LeftPool(msg.sender, riskPool);
303     }
```

Figure 7 source code of *leaveFromPoolInPending* function (Unfixed)

- Fix recommendations: It is recommended to use the current balance to update the rewardDebt.
- Fix results: Fixed.

```
276     function leaveFromPoolInPending(uint256 _amount) external override nonReentrant {
277         // Withdraw desired amount from pool
278         uint256 amount = IERC20(riskPool).balanceOf(msg.sender);
279         uint256 lpPriceUno = IRiskPool(riskPool).lpPriceUno();
280         (uint256 pendingAmount, , ) = IRiskPool(riskPool).getWithdrawRequest(msg.sender);
281         require(((amount - pendingAmount) * lpPriceUno) / 1e18 >= _amount, "UnoRe: withdraw amount overflow");
282         IRiskPool(riskPool).leaveFromPoolInPending(msg.sender, _amount);
283         _harvest(msg.sender);
284
285         lastWithdrawTime[msg.sender] = block.timestamp;
286         emit LeftPool(msg.sender, riskPool, _amount);
287     }
```

Figure 8 source code of *leaveFromPoolInPending* function (Fixed)

[SSRP-4 Medium] The *cancelWithdrawReques* function is missing permission checks

- Description: Any user can call the *cancelWithdrawRequest* function in the SingleSidedInsurancePool contract to cancel a withdrawal request from any address.

```
332     function cancelWithdrawRequest(address _to) external nonReentrant {
333         require(_to != address(0), "UnoRe: zero address");
334         IRiskPool(riskPool).cancelWithrawRequest(_to);
335     }
```

Figure 9 source code of *cancelWithdrawRequest* function (Unfixed)

- Fix recommendations: It is recommended to change it so that the caller can only cancel its own extraction request.
- Fix results: Fixed.

```
329     function cancelWithdrawRequest() external nonReentrant {
330         IRiskPool(riskPool).cancelWithrawRequest(msg.sender);
331     }
```

Figure 10 source code of *cancelWithdrawRequest* function (Fixed)

[SSRP-5 Medium] *enterInPool* function implementation defects

- Description: The *enterInPool* function in the SingleSidedInsurancePool contract is called with the address filled in, so that if an address approved for the contract but the *enterInPool* function has not yet been called, other addresses can participate for that address. And if the owner has authorized the contract more than the amount used in the *initialRewarder* function, the user can also call this function to pledge the UNO tokens of the owner's address to the riskPool.

```
259     function enterInPool(address _from, uint256 _amount) external override nonReentrant {
260         require(_amount != 0, "UnoRe: ZERO Value");
261         updatePool();
262         address token = IRiskPool(riskPool).currency();
263         uint256 lpPriceUno = IRiskPool(riskPool).lpPriceUno();
264         TransferHelper.safeTransferFrom(token, _from, riskPool, _amount);
265         IRiskPool(riskPool).enter(_from, _amount);
266         rewardDebt[_from] =
267             rewardDebt[_from] +
268             ((_amount * 1e18 * uint256(poolInfo.accUnoPerShare)) / lpPriceUno) /
269             ACC_UNO_PRECISION;
270         emit StakedInPool(_from, riskPool, _amount);
271     }
```

Figure 11 source code of *enterInPool* function (Unfixed)

- Fix recommendations: It is recommended that _from be changed to msg.sender.
- Fix results: Fixed.

```
269     function enterInPool(uint256 _amount) external override nonReentrant {
270         require(_amount != 0, "UnoRe: ZERO Value");
271         updatePool();
272         address token = IRiskPool(riskPool).currency();
273         uint256 lpPriceUno = IRiskPool(riskPool).lpPriceUno();
274         TransferHelper.safeTransferFrom(token, msg.sender, riskPool, _amount);
275         IRiskPool(riskPool).enter(msg.sender, _amount);
276         userInfo[msg.sender].rewardDebt =
277             userInfo[msg.sender].rewardDebt +
278             ((_amount * 1e18 * uint256(poolInfo.accUnoPerShare)) / lpPriceUno) /
279             ACC_UNO_PRECISION;
280         userInfo[msg.sender].amount = IERC20(riskPool).balanceOf(msg.sender);
281         emit StakedInPool(msg.sender, riskPool, _amount);
282     }
```

Figure 12 source code of *enterInPool* function (Fixed)

[SSRP-6 Low] riskPool data coverage risk

- Description: The *createRiskPool* function in the SingleSidedInsurancePool contract can be called multiple times, which will overwrite the previous data and may result in the loss of data in the previous riskPool.

```
195     function createRiskPool(
196         string calldata _name,
197         string calldata _symbol,
198         address _factory,
199         address _currency,
200         uint256 _rewardMultiplier
201     ) external onlyOwner {
202         riskPool = IRiskPoolFactory(_factory).newRiskPool(_name, _symbol, address(this), _currency);
203         poolInfo.lastRewardBlock = uint128(block.number);
204         poolInfo.accUnoPerShare = 0;
205         poolInfo.unoMultiplierPerBlock = _rewardMultiplier;
206         emit RiskPoolCreated(address(this), riskPool);
207     }
```

Figure 13 source code of *createRiskPool* function (Unfixed)

- Fix recommendations: It is recommended to change it to be created only once.
- Fix results: Fixed.

[SSRP-7 Low] *initialRewarder* function implementation defects

- Description: The *initialRewarder* function in the SingleSidedInsurancePool contract sends tokens to the Rewarder contract first, and then calls initialRewardBalance. unoBalance only determines if the token amount sent to the contract is greater than 0, and does not determine if the _ amount. This may cause *onUnoReward* function to fail when sending rewards.

```
18      function initialRewardBalance(uint256 _amount) external override onlySSIP {
19          uint256 unoBalance = IERC20(rewardToken).balanceOf(address(this));
20          require(unoBalance > 0, "UnoRe: zero balance");
21          rewardBalance += _amount;
22      }
```

Figure 14 source code of *initialRewarder* function

- Fix recommendations: It is recommended to make a judgement on the number of tokens actually sent to the contract.
- Fix results: Fixed. This function has been removed.

[EA-1 Medium] The *addDex* function is missing a permission check

- Description: The *addDex* function in the ExchangeTokens contract can be called by any user, which may result in incorrect addresses being added to the list, causing the contract to not be called properly.

```
40      function addDex(address _dexAddress) external nonReentrant {
41          require(_dexAddress != address(0), "UnoRe: zero address");
42          for (uint256 ii = 0; ii < dexList.length; ii++) {
43              if (_dexAddress == dexList[ii]) {
44                  return;
45              }
46          }
47          dexList.push(_dexAddress);
48      }
```

Figure 15 source code of *addDex* function

- Fix recommendations: It is recommended to add restrictions on calls to the *addDex* function.
- Fix results: Fixed. The *addDex* function has been removed and replaced by using the TwapOraclePriceFeed contract to get the price.

[EA-2 Info] Recommended fixed compiler version

- Description: The compiler version is not fixed in ExchangeTokens contracts, and there may be potential security risks due to compiler version updates.

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.0;
3
4   import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
5   import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
6   import "./interfaces/IUniswapFactory.sol";
7   import "./interfaces/IUniswapRouter02.sol";
8   import "./interfaces/IExchangeTokens.sol";
9
10  contract ExchangeTokens is IExchangeTokens, ReentrancyGuard {
```

Figure 16 ExchangeTokens contract partial code screenshot

- Fix recommendations: It is recommended to fix the compiler version.
- Fix results: Fixed.

[PP-1 Medium] The *withdrawPremium* function has too much authority

- Description: The *withdrawPremium* function in the PremiumPool contract can withdraw the user's purchase insurance premium.

```
56  function withdrawPremium(
57      address _currency,
58      address _to,
59      uint256 _amount
60  ) external override onlyOwner {
61      require(_to != address(0), "UnoRe: zero address");
62      require(_amount > 0, "UnoRe: zero amount");
63      if (_currency == address(0)) {
64          require(address(this).balance >= _amount, "UnoRe: Insufficient Premium");
65          TransferHelper.safeTransferETH(_to, _amount);
66      } else {
67          require(IERC20(_currency).balanceOf(address(this)) >= _amount, "UnoRe: Insufficient Premium");
68          TransferHelper.safeTransfer(_currency, _to, _amount);
69      }
70      emit PremiumWithdraw(_currency, _to, _amount);
71  }
```

Figure 17 source code of *withdrawPremium* function

- Fix recommendations: It is not recommended to use EOA as owner.
- Fix results: Acknowledged. Project response: The user's purchase insurance premium will be collected in the premium pool through the buyPolicy function of the SalesPolicy contract. But once collected, only the admin can handle the premium. Currently they will be using an EOA here but the goal is to move ownership once their DAO voting contracts are ready. Second, they only use our admin accounts from very secluded, separate dedicated machine with very tight security policies.

[SP-1 Medium] *buyPolicy* function design flaws

- Description: The buyPolicy function in the SalesPolicy contract allows the user to sign the incoming data themselves. This may result in the user being able to arbitrarily enter _coverageAmount, _coverageDuration and _policyPriceInUSDT.

```
77      function buyPolicy(
78          uint256 _coverageAmount,
79          uint256 _coverageDuration,
80          uint256 _policyPriceInUSDT,
81          bytes32 r,
82          bytes32 s,
83          uint8 v
84      ) external payable nonReentrant {
85          (string memory protocolName, , ) = ISingleSidedInsurancePool(SSIP).getProtocolData(protocolIdx);
86          address _signer = getSender(protocolName, _policyPriceInUSDT, _coverageDuration, _coverageAmount, r, s, v);
87          require(_signer != address(0) && _signer == msg.sender, "UnoRe: invalid signer");
88
89          uint256 lastIdx = policyIdx.current();
90
91          uint256 policyPriceInUNO = IExchangeAgent(exchangeAgent).getTokenAmountForUSDT(UNORE_TOKEN, _policyPriceInUSDT);
92
93          getPolicyPerUser[msg.sender] = Policy({
94              coverageAmount: _coverageAmount,
95              coverageDuration: uint64(_coverageDuration),
96              coverStartAt: uint64(block.timestamp),
97              coverStartNumberAt: uint64(block.number),
98              policyPriceInUNO: policyPriceInUNO,
99              policyPriceInUSDT: _policyPriceInUSDT,
100             policyIdx: uint64(lastIdx)
101         });
102
103         TransferHelper.safeTransferFrom(UNORE_TOKEN, msg.sender, premiumPool, policyPriceInUNO);
104         _mint(msg.sender, lastIdx, _policyPriceInUSDT, "");
105
106         totalUtilizedAmount += _coverageAmount;
107
108         policyIdx.increment();
109         emit BuyPolicy(protocolIdx, lastIdx, msg.sender, _coverageAmount, policyPriceInUNO, _policyPriceInUSDT);
110     }
```

Figure 18 source code of *buyPolicy* function (Unfixed)

- Fix recommendations: It is recommended to amend the signature confirmation to the project party's address.
- Fix results: Fixed. They added signer address determined by admin for signature confirmation. And also they added signature deadline validation part in *buyPolicy* function. Current max signature deadline duration is set to 1 week.

```
82  function buyPolicy(
83      uint256 _coverageAmount,
84      uint256 _coverageDuration,
85      uint256 _policyPriceInUSDT,
86      uint256 _signedTime,
87      bytes32 r,
88      bytes32 s,
89      uint8 v
90  ) external payable nonReentrant {
91      (string memory protocolName, , ) = ISingleSidedReinsurancePool(SSRP).getProtocolData(protocolIdx);
92      address _signer = getSender(protocolName, _policyPriceInUSDT, _coverageDuration, _coverageAmount, _signedTime, r, s, v);
93      require(_signer != address(0) && _signer == signer, "UnoRe: invalid signer");
94      require(_signedTime <= block.timestamp && block.timestamp - _signedTime < maxDeadline, "UnoRe: signature expired");
95
96      uint256 lastIdx = policyIdx.current();
97
98      uint256 policyPriceInUNO = IExchangeAgent(exchangeAgent).getTokenAmountForUSDT(UNORE_TOKEN, _policyPriceInUSDT);
99
100     getPolicyPerUser[msg.sender] = Policy({
101         coverageAmount: _coverageAmount,
102         coverageDuration: uint64(_coverageDuration),
103         coverStartAt: uint64(block.timestamp),
104         coverStartNumberAt: uint64(block.number),
105         policyPriceInUNO: policyPriceInUNO,
106         policyPriceInUSDT: _policyPriceInUSDT,
107         policyIdx: uint64(lastIdx)
108     });
109
110     TransferHelper.safeTransferFrom(UNORE_TOKEN, msg.sender, premiumPool, policyPriceInUNO);
111     _mint(msg.sender, lastIdx, _policyPriceInUSDT, "");
112
113     totalUtilizedAmount += _coverageAmount;
114
115     policyIdx.increment();
116     emit BuyPolicy(protocolIdx, lastIdx, msg.sender, _coverageAmount, policyPriceInUNO, _policyPriceInUSDT);
117 }
```

Figure 19 source code of *buyPolicy* function (Fixed)

[SP-2 Info] Unused ERC1155 tokens

- Description: The ERC1155 token is minted inside the function, but it is not used elsewhere.

```
109
110     TransferHelper.safeTransferFrom(UNORE_TOKEN, msg.sender, premiumPool, policyPriceInUNO);
111     _mint(msg.sender, lastIdx, _policyPriceInUSDT, "");
112
113     totalUtilizedAmount += _coverageAmount;
114
```

Figure 20 partial source code of *buyPolicy* function

- Fix recommendations: It is recommended to confirm whether the business logic is met.
- Fix results: Acknowledged. Project response: The ERC1155 token is not used for now but they are planning to expand their SalesPolicy contract (Out of Scope for now) in a various ways.

[SP-3 Info] BuyPolicy event triggers parameter error

- Description: The BuyPolicy event trigger in the SalesPolicy contract has an incorrect parameter.

```
40      event BuyPolicy(
41          uint256 indexed _protocolIdx,
42          uint256 indexed _policyIdx,
43          address _owner,
44          uint256 _coverageAmount,
45          uint256 _policyPriceInUSDT,
46          uint256 _policyPriceInUNO
47      );
```

Figure 21 BuyPolicy event statement (Unfixed)

```
emit BuyPolicy(protocolIdx, lastIdx, msg.sender, _coverageAmount, policyPriceInUNO, policyPriceInUNO);
```

Figure 22 BuyPolicy event trigger (Unfixed)

- Fix recommendations: It is recommended to change to the correct parameters.
- Fix results: Fixed.

```
42
43      event BuyPolicy(
44          uint256 indexed _protocolIdx,
45          uint256 indexed _policyIdx,
46          address _owner,
47          uint256 _coverageAmount,
48          uint256 _policyPriceInUNO,
49          uint256 _policyPriceInUSDT
50      );
```

Figure 23 BuyPolicy event statement (Fixed)

```
emit BuyPolicy(protocolIdx, lastIdx, msg.sender, _coverageAmount, policyPriceInUNO, _policyPriceInUSDT);
```

Figure 24 BuyPolicy event trigger (Fixed)

[RW-1 Medium] *onUnoReward* function implementation defects

- Description: The *onUnoReward* function in Rewarder contract throws an exception when the token reward is 0, which may cause the singleSidedInsurancePool contract migrate, harvest, *leaveFromPending* and *leaveFromPoolInPending* functions to not be called properly.

```
24      function onUnoReward(address to, uint256 unoAmount) external override onlySSIP {
25          require(rewardBalance > 0, "UnoRe: zero rewarder balance");
26          if (unoAmount > rewardBalance) {
27              TransferHelper.safeTransfer(rewardToken, to, rewardBalance);
28              rewardBalance = 0;
29          } else {
30              TransferHelper.safeTransfer(rewardToken, to, unoAmount);
31              rewardBalance -= unoAmount;
32          }
33      }
```

Figure 25 source code of *onUnoReward* function (Unfixed)

- Fix recommendations: It is recommended that no exceptions are thrown.
- Fix results: Fixed.

```
17    function onUnoReward(address to, uint256 unoAmount) external override onlySSRP returns (uint256) {
18        uint256 rewardBalance = IERC20(rewardToken).balanceOf(address(this));
19        if (unoAmount > rewardBalance) {
20            TransferHelper.safeTransfer(rewardToken, to, rewardBalance);
21            rewardBalance = 0;
22            return rewardBalance;
23        } else {
24            TransferHelper.safeTransfer(rewardToken, to, unoAmount);
25            rewardBalance -= unoAmount;
26            return unoAmount;
27        }
28    }
```

Figure 26 source code of *onUnoReward* function (Fixed)

[RW-2 Medium] Error on *balanceOf* function call with parameters

- Description: The *onUnoReward* function in the Rewarder contract is using the wrong address to get the balance of the rewardToken tokens for this contract.

```
17    function onUnoReward(address to, uint256 unoAmount) external override onlySSRP returns (uint256) {
18        uint256 rewardBalance = IERC20(rewardToken).balanceOf(to);
19        if (unoAmount > rewardBalance) {
20            TransferHelper.safeTransfer(rewardToken, to, rewardBalance);
21            rewardBalance = 0;
22            return rewardBalance;
23        } else {
24            TransferHelper.safeTransfer(rewardToken, to, unoAmount);
25            rewardBalance -= unoAmount;
26            return unoAmount;
27        }
28    }
```

Figure 27 source code of *onUnoReward* function (Unfixed)

- Fix recommendations: It is recommended 'to' change the to address to the address of this contract.
- Fix results: Fixed.

```
17    function onUnoReward(address to, uint256 unoAmount) external override onlySSRP returns (uint256) {
18        uint256 rewardBalance = IERC20(rewardToken).balanceOf(address(this));
19        if (unoAmount > rewardBalance) {
20            TransferHelper.safeTransfer(rewardToken, to, rewardBalance);
21            rewardBalance = 0;
22            return rewardBalance;
23        } else {
24            TransferHelper.safeTransfer(rewardToken, to, unoAmount);
25            rewardBalance -= unoAmount;
26            return unoAmount;
27        }
28    }
```

Figure 28 source code of *onUnoReward* function (Fixed)

[MT-1 Low] The *onMigration* function is not implemented

- Description: The internal implementation of the *onMigration* function is not visible and may pose a security risk.

```
216     function migrate(bool _isWithdraw) external nonReentrant {
217         require(migrateTo != address(0), "UnoRe: zero address");
218         updatePool();
219         uint256 amount = IERC20(riskPool).balanceOf(msg.sender);
220         if (amount > 0) {
221             uint256 accumulatedUno = (amount * uint256(poolInfo.accUnoPerShare)) / ACC_UNO_PRECISION;
222             uint256 _pendingUno = accumulatedUno - rewardDebt[msg.sender];
223             rewardDebt[msg.sender] = 0;
224             if (rewarder != address(0)) {
225                 IRewarder(rewarder).onUnoReward(msg.sender, _pendingUno);
226                 emit Harvest(msg.sender, msg.sender, _pendingUno);
227             }
228         }
229         bool isLocked = block.timestamp - lastWithdrawTime[msg.sender] > LOCK_TIME;
230         IRiskPool(riskPool).migrateLP(msg.sender, migrateTo, _isWithdraw, isLocked);
231         IMigration(migrateTo).onMigration(msg.sender, amount, "");
232     }
```

Figure 29 source code of *migrate* function

```
6   contract Migration is IMigration {
7       constructor() {}
8
9       function onMigration(
10          address who_,
11          uint256 amount_,
12          bytes memory data_
13      ) external virtual override {}
14  }
15
```

Figure 30 source code of Migration contract

- Fix recommendations: It is recommended to confirm if this is required.
- Fix results: Acknowledged. Project response: It will be used to migrate to the new LP pool in the future but is not used for now. It is their future migration contract's simple template and it will be not deployed now.

[RP-1 Low] *policyClaim* function implementation defects

- Description: If cryptoBalance is just greater than _amount, it will result in the number of LPs reserved in the RiskPool contract being less than 100.

```
88      function policyClaim(address _to, uint256 _amount) external override onlySSIP {
89          uint256 cryptoBalance = IERC20(currency).balanceOf(address(this));
90          require(cryptoBalance > 0 && totalSupply() > 0, "UnoRe: zero uno and lp balance");
91          if (cryptoBalance > _amount) {
92              TransferHelper.safeTransfer(currency, _to, _amount);
93              emit LogPolicyClaim(_to, _amount);
94          } else {
95              TransferHelper.safeTransfer(currency, _to, cryptoBalance - MIN_LP_CAPITAL);
96              emit LogPolicyClaim(_to, cryptoBalance - MIN_LP_CAPITAL);
97          }
98          cryptoBalance = IERC20(currency).balanceOf(address(this));
99          lpPriceUno = (cryptoBalance * 1e18) / totalSupply(); // UNO value per lp
100     }
```

Figure 31 source code of *policyClaim* function (Unfixed)

- Fix recommendations: It is recommended to delete redundant codes.
- Fix results: Fixed.

```
89      function policyClaim(address _to, uint256 _amount) external override onlySSRP returns (uint256 realClaimAmount) {
90          uint256 cryptoBalance = IERC20(currency).balanceOf(address(this));
91          require(totalSupply() > 0, "UnoRe: zero lp balance");
92          require(cryptoBalance > MIN_LP_CAPITAL, "UnoRe: minimum UNO capital underflow");
93          if (cryptoBalance - MIN_LP_CAPITAL > _amount) {
94              TransferHelper.safeTransfer(currency, _to, _amount);
95              realClaimAmount = _amount;
96              emit LogPolicyClaim(_to, _amount);
97          } else {
98              TransferHelper.safeTransfer(currency, _to, cryptoBalance - MIN_LP_CAPITAL);
99              realClaimAmount = cryptoBalance - MIN_LP_CAPITAL;
100             emit LogPolicyClaim(_to, cryptoBalance - MIN_LP_CAPITAL);
101         }
102         cryptoBalance = IERC20(currency).balanceOf(address(this));
103         lpPriceUno = (cryptoBalance * 1e18) / totalSupply(); // UNO value per lp
104     }
```

Figure 32 source code of *policyClaim* function (Fixed)

[RP-2 Low] The *transferFrom* and *transfer* functions are flawed in their judgement

- Description: The *transfer* and *transferFrom* functions in the RiskPool contract do not include the case where the balance minus the pendingAmount is equal to the amount, which may result in the part of the LP that is not in the pending state not being transferred.

```
159     function transfer(address recipient, uint256 amount) external override returns (bool) {
160         require(
161             balanceOf(msg.sender) - uint256(withdrawRequestPerUser[msg.sender].pendingAmount) > amount,
162             "ERC20: transfer amount exceeds balance or pending WR"
163         );
164         _transfer(msg.sender, recipient, amount);
165 
166         ISingleSidedReinsurancePool(SSRP).lpTransfer(msg.sender, recipient, amount);
167         return true;
168     }
169 
170     function transferFrom(
171         address sender,
172         address recipient,
173         uint256 amount
174     ) external override returns (bool) {
175         require(
176             balanceOf(sender) - uint256(withdrawRequestPerUser[sender].pendingAmount) > amount,
177             "ERC20: transfer amount exceeds balance or pending WR"
178         );
179         _transfer(sender, recipient, amount);
180 
181         uint256 currentAllowance = _allowances[sender][msg.sender];
182         require(currentAllowance >= amount, "ERC20: transfer amount exceeds allowance");
183         _approve(sender, msg.sender, currentAllowance - amount);
184         ISingleSidedReinsurancePool(SSRP).lpTransfer(sender, recipient, amount);
185         return true;
186     }
```

Figure 33 source code of *transfer* and *transferFrom* functions (Unfixed)

- Fix recommendations: It is recommended to change this to greater than or equal to.
- Fix results: Fixed.

```
158     function transfer(address recipient, uint256 amount) external override returns (bool) {
159         require(
160             balanceOf(msg.sender) - uint256(withdrawRequestPerUser[msg.sender].pendingAmount) >= amount,
161             "ERC20: transfer amount exceeds balance or pending WR"
162         );
163         _transfer(msg.sender, recipient, amount);
164 
165         ISingleSidedReinsurancePool(SSRP).lpTransfer(msg.sender, recipient, amount);
166         return true;
167     }
168 
169     function transferFrom(
170         address sender,
171         address recipient,
172         uint256 amount
173     ) external override returns (bool) {
174         require(
175             balanceOf(sender) - uint256(withdrawRequestPerUser[sender].pendingAmount) >= amount,
176             "ERC20: transfer amount exceeds balance or pending WR"
177         );
178         _transfer(sender, recipient, amount);
179 
180         uint256 currentAllowance = _allowances[sender][msg.sender];
181         require(currentAllowance >= amount, "ERC20: transfer amount exceeds allowance");
182         _approve(sender, msg.sender, currentAllowance - amount);
183         ISingleSidedReinsurancePool(SSRP).lpTransfer(sender, recipient, amount);
184         return true;
185     }
```

Figure 34 source code of *transfer* and *transferFrom* functions (Fixed)

[RP-3 Low] The *leaveFromPending* function has a flawed judgment

- Description: The business logic is that 100 currency tokens are kept in the RiskPool contract at all times. But the last user to withdraw from the RiskPool contract is not the currency tokens minus the amount retained.

```
64  function leaveFromPending(address _to) external override onlySSRP returns (uint256, uint256) {
65      uint256 cryptoBalance = IERC20(currency).balanceOf(address(this));
66      uint256 pendingAmount = uint256(withdrawRequestPerUser[_to].pendingAmount);
67      require(cryptoBalance > 0, "UnoRe: zero uno balance");
68      require(balanceOf(_to) >= pendingAmount, "UnoRe: lp balance overflow");
69      _withdrawImplement(_to);
70      uint256 pendingAmountInUno = (pendingAmount * lpPriceUno) / 1e18;
71      if (pendingAmountInUno < cryptoBalance) {
72          TransferHelper.safeTransfer(currency, _to, pendingAmountInUno);
73          emit LogLeaveFromPending(_to, pendingAmount, pendingAmountInUno);
74          return (pendingAmount, pendingAmountInUno);
75      } else {
76          TransferHelper.safeTransfer(currency, _to, cryptoBalance - MIN_LP_CAPITAL);
77          emit LogLeaveFromPending(_to, pendingAmount, cryptoBalance - MIN_LP_CAPITAL);
78          return (pendingAmount, cryptoBalance - MIN_LP_CAPITAL);
79      }
80  }
```

Figure 35 source code of *leaveFromPending* function (Unfixed)

- Fix recommendations: It is recommended to confirm that the code meets the business logic.
- Fix results: Fixed.

```
64  function leaveFromPending(address _to) external override onlySSRP returns (uint256, uint256) {
65      uint256 cryptoBalance = IERC20(currency).balanceOf(address(this));
66      uint256 pendingAmount = uint256(withdrawRequestPerUser[_to].pendingAmount);
67      require(cryptoBalance > 0, "UnoRe: zero uno balance");
68      require(balanceOf(_to) >= pendingAmount, "UnoRe: lp balance overflow");
69      _withdrawImplement(_to);
70      uint256 pendingAmountInUno = (pendingAmount * lpPriceUno) / 1e18;
71      if (pendingAmountInUno < cryptoBalance - MIN_LP_CAPITAL) {
72          TransferHelper.safeTransfer(currency, _to, pendingAmountInUno);
73          emit LogLeaveFromPending(_to, pendingAmount, pendingAmountInUno);
74          return (pendingAmount, pendingAmountInUno);
75      } else {
76          TransferHelper.safeTransfer(currency, _to, cryptoBalance - MIN_LP_CAPITAL);
77          emit LogLeaveFromPending(_to, pendingAmount, cryptoBalance - MIN_LP_CAPITAL);
78          return (pendingAmount, cryptoBalance - MIN_LP_CAPITAL);
79      }
80  }
```

Figure 36 source code of *leaveFromPending* function (Fixed)

[RP-4 Info] Redundant codes

- Description: There are some redundant codes in the contract.
- Fix recommendations: It is recommended to delete redundant codes.
- Fix results: Fixed.

Other audit items explained

1. Notes to users

The user's purchase insurance premium will be collected in the premium pool through the *buyPolicy* function of the SalesPolicy contract. The owner of a PremiumPool contract can call the *withdrawPremium* function to withdraw any number of arbitrary tokens from the contract (Normally only the premium paid by the user for the insurance will be sent to this contract).

Some of the functions in the migrate function of the SingleSidedInsurancePool contract (the *onMigration* function) are not yet implemented and may be potentially risky.

If the balance of tokens in the RiskPool is insufficient, the compensation received by the user may be less than expected.

The insurance claim calling authority is currently held by EOA and the project promises to transfer it to the DAO contract at a later stage.

There is a 10-day waiting period after the user enters the RiskPool if he/she needs to withdraw, after which he/she will be able to collect.

If the Rewarder contract has too few or zero reward tokens, users may not be able to access their normal rewards.

Appendix 1 Vulnerability Severity Level

Vulnerability Level	Description	Example
Critical	Vulnerabilities that lead to the complete destruction of the project and cannot be recovered. It is strongly recommended to fix.	Malicious tampering of core contract privileges and theft of contract assets.
High	Vulnerabilities that lead to major abnormalities in the operation of the contract due to contract operation errors. It is strongly recommended to fix.	Unstandardized docking of the USDT interface, causing the user's assets to be unable to withdraw.
Medium	Vulnerabilities that cause the contract operation result to be inconsistent with the design but will not harm the core business. It is recommended to fix.	The rewards that users received do not match expectations.
Low	Vulnerabilities that have no impact on the operation of the contract, but there are potential security risks, which may affect other functions. The project party needs to confirm and determine whether the fix is needed according to the business scenario as appropriate.	Inaccurate annual interest rate data queries.
Info	There is no impact on the normal operation of the contract, but improvements are still recommended to comply with widely accepted common project specifications.	It is needed to trigger corresponding events after modifying the core configuration.

Appendix 2 Description of Audit Categories

No.	Categories	Subitems
1	Coding Conventions	Compiler Version Security
		Deprecated Items
		Redundant Code
		require/assert Usage
		Gas Consumption
2	General Vulnerability	Integer Overflow/Underflow
		Reentrancy
		Pseudo-random Number Generator (PRNG)
		Transaction-Ordering Dependence
		DoS (Denial of Service)
		Function Call Permissions
		call/delegatecall Security
		Returned Value Security
		tx.origin Usage
		Replay Attack
		Overriding Variables
3	Business Security	Business Logics
		Business Implementations

1. Coding Conventions

1.1. Compiler Version Security

The old version of the compiler may cause various known security issues. Developers are advised to specify the contract code to use the latest compiler version and eliminate the compiler alerts.

1.2. Deprecated Items

The Solidity smart contract development language is in rapid iteration. Some keywords have been deprecated by newer versions of the compiler, such as throw, years, etc. To eliminate the potential pitfalls they may cause, contract developers should not use the keywords that have been deprecated by the current compiler version.

1.3. Redundant Code

Redundant code in smart contracts can reduce code readability and may require more gas consumption for contract deployment. It is recommended to eliminate redundant code.

1.4. SafeMath Features

Check whether the functions within the SafeMath library are correctly used in the contract to perform mathematical operations, or perform other overflow prevention checks.

1.5. require/assert Usage

Solidity uses state recovery exceptions to handle errors. This mechanism will undo all changes made to the state in the current call (and all its subcalls) and flag the errors to the caller. The functions assert and require can be used to check conditions and throw exceptions when the conditions are not met. The assert function can only be used to test for internal errors and check non-variables. The require function is used to confirm the validity of conditions, such as whether the input variables or contract state variables meet the conditions, or to verify the return value of external contract calls.

1.6. Gas Consumption

The smart contract virtual machine needs gas to execute the contract code. When the gas is insufficient, the code execution will throw an out of gas exception and cancel all state changes. Contract developers are required to control the gas consumption of the code to avoid function execution failures due to insufficient gas.

1.7. Visibility Specifiers

Check whether the visibility conforms to design requirement.

1.8. Fallback Usage

Check whether the Fallback function has been used correctly in the current contract.

2. General Vulnerability

2.1. Integer overflow

Integer overflow is a security problem in many languages, and they are especially dangerous in smart contracts. Solidity can handle up to 256-bit numbers (2**256-1). If the maximum number is increased by 1, it will overflow to 0. Similarly, when the number is a uint type, 0 minus 1 will underflow to get the maximum number value. Overflow conditions can lead to incorrect results, especially if its possible results are not

expected, which may affect the reliability and safety of the program. For the compiler version after Solidity 0.8.0, smart contracts will perform overflow checking on mathematical operations by default. In the previous compiler versions, developers need to add their own overflow checking code, and SafeMath library is recommended to use.

2.2. Reentrancy

The reentrancy vulnerability is the most typical Ethereum smart contract vulnerability, which has caused the DAO to be attacked. The risk of reentry attack exists when there is an error in the logical order of calling the call.value() function to send assets.

2.3 Pseudo-random Number Generator (PRNG)

Random numbers may be used in smart contracts. In solidity, it is common to use block information as a random factor to generate, but such use is insecure. Block information can be controlled by miners or obtained by attackers during transactions, and such random numbers are to some extent predictable or collidable.

2.4. Transaction-Ordering Dependence

In the process of transaction packing and execution, when faced with transactions of the same difficulty, miners tend to choose the one with higher gas cost to be packed first, so users can specify a higher gas cost to have their transactions packed and executed first.

2.5. DoS(Denial of Service)

DoS, or Denial of Service, can prevent the target from providing normal services. Due to the immutability of smart contracts, this type of attack can make it impossible to ever restore the contract to its normal working state. There are various reasons for the denial of service of a smart contract, including malicious revert when acting as the recipient of a transaction, gas exhaustion caused by code design flaws, etc.

2.6. Function Call Permissions

If smart contracts have high-privilege functions, such as coin minting, self-destruction, change owner, etc., permission restrictions on function calls are required to avoid security problems caused by permission leakage.

2.7. call/delegatecall Security

Solidity provides the call/delegatecall function for function calls, which can cause call injection vulnerability if not used properly. For example, the parameters of the call, if controllable, can control this contract to perform unauthorized operations or call dangerous functions of other contracts.

2.8. Returned Value Security

In Solidity, there are transfer(), send(), call.value() and other methods. The transaction will be rolled back if the transfer fails, while send and call.value will return false if the transfer fails. If the return is not correctly

judged, the unanticipated logic may be executed. In addition, in the implementation of the transfer/transferFrom function of the token contract, it is also necessary to avoid the transfer failure and return false, so as not to create fake recharge loopholes.

2.9. tx.origin Usage

The tx.origin represents the address of the initial creator of the transaction. If tx.origin is used for permission judgment, errors may occur; in addition, if the contract needs to determine whether the caller is the contract address, then tx.origin should be used instead of extcodesize.

2.10. Replay Attack

A replay attack means that if two contracts use the same code implementation, and the identity authentication is in the transmission of parameters, the transaction information can be replayed to the other contract to execute the transaction when the user executes a transaction to one contract.

2.11. Overriding Variables

There are complex variable types in Solidity, such as structures, dynamic arrays, etc. When using a lower version of the compiler, improperly assigning values to it may result in overwriting the values of existing state variables, causing logical exceptions during contract execution.

Appendix 3 Disclaimer

This report is made in response to the project code. No description, expression or wording in this report shall be construed as an endorsement, affirmation or confirmation of the project. This audit is only applied to the type of auditing specified in this report and the scope of given in the results table. Other unknown security vulnerabilities are beyond auditing responsibility. Beosin Technology only issues this report based on the attacks or vulnerabilities that already existed or occurred before the issuance of this report. For the emergence of new attacks or vulnerabilities that exist or occur in the future, Beosin Technology lacks the capability to judge its possible impact on the security status of smart contracts, thus taking no responsibility for them. The security audit analysis and other contents of this report are based solely on the documents and materials that the contract provider has provided to Beosin Technology before the issuance of this report, and the contract provider warrants that there are no missing, tampered, deleted; if the documents and materials provided by the contract provider are missing, tampered, deleted, concealed or reflected in a situation that is inconsistent with the actual situation, or if the documents and materials provided are changed after the issuance of this report, Beosin Technology assumes no responsibility for the resulting loss or adverse effects. The audit report issued by Beosin Technology is based on the documents and materials provided by the contract provider, and relies on the technology currently possessed by Beosin. Due to the technical limitations of any organization, this report conducted by Beosin still has the possibility that the entire risk cannot be completely detected. Beosin disclaims any liability for the resulting losses.

The final interpretation of this statement belongs to Beosin Technology.

Appendix 4 About Beosin

BEOSIN is a leading global blockchain security company dedicated to the construction of blockchain security ecology, with team members coming from professors, post-docs, PhDs from renowned universities and elites from head Internet enterprises who have been engaged in information security industry for many years. BEOSIN has established in-depth cooperation with more than 100 global blockchain head enterprises; and has provided security audit and defense deployment services for more than 1,000 smart contracts, more than 50 blockchain platforms and landing application systems, and nearly 100 digital financial enterprises worldwide. Relying on technical advantages, BEOSIN has applied for nearly 50 software invention patents and copyrights.

BEOSIN

Blockchain Security

Official Website

https://beosin.com

Twitter

https://twitter.com/Beosin_com