BEOSIN
Blockchain Security

Uno-Re

Smart Contract Security Audit

No. 202401310928

Jan 31st, 2024

SECURING BLOCKCHAIN ECOSYSTEM

WWW.BEOSIN.COM

Contents

Summary of Audit Results

After auditing, 2 Critical-risk,3 High-risk,4 Medium-risk,2 Low-risk and 4 Info items were identified in the Uno-Re project. Specific audit details will be presented in the Findings section. Users should pay attention to the following aspects when interacting with this project:

Critical	Fixed: 2	Acknowledged: 0
High	Fixed: 3	Acknowledged: 0
Medium	Fixed: 4	Acknowledged: 0
Low	Fixed: 2	Acknowledged: 0
Info	Fixed: 4	Acknowledged: 0

- **Risk Description:**

1. In the SSRP section, the insurance claim process tends to be centralized, with the claimAssessor calling the `policyClaim` function to process insurance claims.

2. There is an `emergencyWithdraw` function in the insurance pool. If activated before the insurance settlement is complete, it may have a certain impact on the financial security of the insurance pool.

- **Project Description:**

Business overview

The Uno-Re project consists of two parts: SSIP-SSRP and unore-uno-dao. The SSIP-SSRP part involves the logic for insurance sales and the insurance pool, while the unore-uno-dao part involves the logic for rewarding VeUno tokens.

When purchasing the corresponding insurance, the Policyholder needs to first request and obtain relevant signatures from the corresponding Signer. This is a prerequisite for acquiring insurance policies and is necessary for the purchase to occur when meeting the corresponding MLR. When the Policyholder applies for a claim, there are two types: SSIP pool claims and SSRP pool claims. For SSIP claims, users need to apply for insurance compensation in the corresponding SSIP pool. The compensation is then approved based on the contract's strategy through either an oracle or governance (GOV) judgment. The SSRP pool, relatively centralized, processes claims by calling the `policyClaim` function through an account with CLAIM_ACCESSOR permissions. Seventy percent of the fees associated with purchasing policies are allocated to reward the SSIP pool, 20% is utilized to appreciate UNO tokens, and the remaining 10% is designated for rewarding the SSRP pool.

The insurance pools are divided into two types: SSIP pools and SSRP pools. The SSIP pool leans towards decentralization, while the SSRP pool leans towards centralization. In the SSIP pool, depositors can stake their corresponding collateral tokens in the SSIP pool as part of the overall insurance fund, earning rewards based on the fees Policyholders pay when purchasing insurance. Depositors need to initiate a pending request and comply with specific MCR and SCR restrictions when withdrawing collateral. The withdrawal operation is executed only after a certain pending period to ensure the safety of the insurance fund. In the SSRP pool, the operations for depositing and withdrawing are similar, but they do not require compliance with specific MCR and SCR requirements.

The reward distribution in unore-uno-dao occurs in multiple cycles. Each time a reward pool is added, the reward cycle resets, and the unreleased rewards are added to the existing rewards. Rewards are distributed based on the user's holdings of VeUno tokens and the duration of holding during each cycle.

1 Overview

1.1 Project Overview

Project Name	Uno-Re
Project Language	Solidity
Platform	Ethereum
Github	https://github.com/Uno-Re/SSIP-SSRP-contracts/tree/main https://github.com/Uno-Re/unore-uno-dao/tree/master
Audit Scope	SSIP-SSRP-contracts: ./contracts/EIP712MetaTransaction.sol ./contracts/factories/SyntheticSSRPFactory.sol ./contracts/factories/SalesPolicyFactory.sol ./contracts/factories/RiskPoolFactory.sol ./contracts/factories/RewarderFactory.sol ./contracts/factories/SyntheticSSIPFactory.sol ./contracts/RiskPoolERC20.sol ./contracts/SingleSidedReinsurancePool.sol ./contracts/libraries/EIP712Base.sol ./contracts/libraries/TransferHelper.sol ./contracts/libraries/MultiSigWallet.sol ./contracts/ExchangeAgent.sol ./contracts/SingleSidedInsurancePool.sol ./contracts/CapitalAgent.sol ./contracts/uma/EscalationManager.sol ./contracts/RiskPool.sol ./contracts/Rewarder.sol ./contracts/SalesPolicy.sol ./contracts/PremiumPool.sol unore-uno-dao: ./access/Owned.sol ./libraries/TransferHelper.sol ./SmartWalletChecker.sol ./apps/VeUnoDaoYieldDistributor.sol ./misc/Helpers.sol ./Ownership.sol

commit	./automation/Resolver.sol 6cbc5261303b35c6d05883ce0064353fde02216e 92cbdac57f6716043b90e382215a53d2883a558f 3711e2aea1dbf6ed1ae41b7776c14ca3c53aaae0 13f68f855877ae652557c2bb11b2948e4ace42a0 f18b31ef90cf6de1881327b64d6f49bba0de1d42

1.2 Audit Overview

Audit work duration: Dec 18, 2023 – Jan 31, 2024

Audit team: Beosin Security Team

1.3 Audit Method

The audit methods are as follows:

1. Formal Verification

Formal verification is a technique that uses property-based approaches for testing and verification. Property specifications define a set of rules using Beosin's library of security expert rules. These rules call into the contracts under analysis and make various assertions about their behavior. The rules of the specification play a crucial role in the analysis. If the rule is violated, a concrete test case is provided to demonstrate the violation.

2. Manual Review

Using manual auditing methods, the code is read line by line to identify potential security issues. This ensures that the contract's execution logic aligns with the client's specifications and intentions, thereby safeguarding the accuracy of the contract's business logic.

The manual audit is divided into three groups to cover the entire auditing process:

The Basic Testing Group is primarily responsible for interpreting the project's code and conducting comprehensive functional testing.

The Simulated Attack Group is responsible for analyzing the audited project based on the collected historical audit vulnerability database and security incident attack models. They identify potential attack vectors and collaborate with the Basic Testing Group to conduct simulated attack tests.

The Expert Analysis Group is responsible for analyzing the overall project design, interactions with third parties, and security risks in the on-chain operational environment. They also conduct a review of the entire audit findings.

3. Static Analysis

Static analysis is a method of examining code during compilation or static analysis to detect issues. Beosin-VaaS can detect more than 100 common smart contract vulnerabilities through static analysis, such as reentrancy and block parameter dependency. It allows early and efficient discovery of problems to improve code quality and security.

2 Findings

Index	Risk description	Severity level	Status
Uno-Re-01	Migration Data Calculation Error	Critical	Fixed
Uno-Re-02	Staking Locking Risk	Critical	Fixed
Uno-Re-03	Tx.origin Restriction Issue	High	Fixed
Uno-Re-04	Function Call Failed	High	Fixed
Uno-Re-05	Withdrawable Stake Before Initiating Compensation	High	Fixed
Uno-Re-06	TotalCapital Are Inaccurate	Medium	Fixed
Uno-Re-07	Chainlink DOS Attack	Medium	Fixed
Uno-Re-08	Missing Approval	Medium	Fixed
Uno-Re-09	Reward Claiming Duplication	Medium	Fixed
Uno-Re-10	Signature Reuse Risk	Low	Fixed
Uno-Re-11	Expired Verification Is Invalid	Low	Fixed
Uno-Re-12	Redundant Increment	Info	Fixed
Uno-Re-13	Meaningless Data Comparison	Info	Fixed
Uno-Re-14	Redundant code	Info	Fixed
Uno-Re-15	Missing Event Trigger	Info	Fixed

Finding Details:

[Uno-Re-01] Migration Data Calculation Error

Severity Level	**Critical**
Type	Business Security
Lines	SingleSidedInsurancePool.sol #L294-304 SingleSidedReinsurancePool.sol #L203-213 RiskPool.sol #133-157

Description

In the SingleSidedInsurancePool contract, the `migrate` function, when calculating the migration quantity, processes the tokens in the pending state through leave and sends the completed pending tokens to the `_to` address. However, subsequently, these tokens sent to the `_to` address are also included in the calculation of `migratedAmount`. This results in an incorrect migration quantity obtained by the `onMigration` function, potentially allowing attackers to exploit the migration vulnerability to harvest staked tokens. Similar issues exist in the SingleSidedReinsurancePool contract.

RiskPool:

```
    function migrateLP(address _to, address _migrateTo, bool _isUnLocked) external override onlySSRP returns (uint256) {
        require(_migrateTo != address(0), "UnoRe: zero address");
        uint256 migratedAmount;
        uint256 cryptoBalance;
        if (_isUnLocked && withdrawRequestPerUser[_to].pendingAmount > 0) {
            uint256 pendingAmountInUno = (uint256(withdrawRequestPerUser[_to].pendingAmount) * lpPriceUno) / 1e18;
            cryptoBalance = currency != address(0) ? IERC20(currency).balanceOf(address(this)) : address(this).balance;
            if (pendingAmountInUno < cryptoBalance - MIN_LP_CAPITAL) {
                if (currency != address(0)) {
                    TransferHelper.safeTransfer(currency, _to, pendingAmountInUno);
                } else {
                    TransferHelper.safeTransferETH(_to,
```

```
pendingAmountInUno);
                }
                migratedAmount += pendingAmountInUno;
                _withdrawImplement(_to);
            } else {
                if (currency != address(0)) {
                    TransferHelper.safeTransfer(currency, _to,
cryptoBalance - MIN_LP_CAPITAL);
                } else {
                    TransferHelper.safeTransferETH(_to, cryptoBalance
- MIN_LP_CAPITAL);
                }
                migratedAmount += cryptoBalance - MIN_LP_CAPITAL;
                _withdrawImplementIrregular(_to, ((cryptoBalance -
MIN_LP_CAPITAL) * 1e18) / lpPriceUno);
            }
        } else {
            if (withdrawRequestPerUser[_to].pendingAmount > 0) {
                _cancelWithdrawRequest(_to);
            }
        }
```

SingleSidedInsurancePool:

```
    function migrate() external nonReentrant isAlive {
        require(migrateTo != address(0), "UnoRe: zero address");
        _harvest(msg.sender);
        bool isUnLocked = block.timestamp -
userInfo[msg.sender].lastWithdrawTime > lockTime;
        uint256 migratedAmount =
IRiskPool(riskPool).migrateLP(msg.sender, migrateTo, isUnLocked);
        ICapitalAgent(capitalAgent).SSIPPolicyCaim(migratedAmount, 0,
false);
        IMigration(migrateTo).onMigration(msg.sender, migratedAmount,
"");
        userInfo[msg.sender].amount = 0;
        userInfo[msg.sender].rewardDebt = 0;
        emit LogMigrate(msg.sender, migrateTo, migratedAmount);
    }
```

Recommendation	It is recommended to use the actually deducted staking quantity for calculating the migration quantity.

Status

Fixed. The project team modified the relevant code to use the return value as the actual migration quantity.

```
    function migrateLP(address _to, address _migrateTo, bool _isUnLocked) external override onlySSRP returns (uint256) {
        require(_migrateTo != address(0), "UnoRe: zero address");
        uint256 migratedAmount;
        uint256 cryptoBalance;
        if (_isUnLocked && withdrawRequestPerUser[_to].pendingAmount > 0) {
            uint256 pendingAmountInUno = (uint256(withdrawRequestPerUser[_to].pendingAmount) * lpPriceUno) / 1e18;
            cryptoBalance = currency != address(0) ? IERC20(currency).balanceOf(address(this)) : address(this).balance;
            if (pendingAmountInUno < cryptoBalance - MIN_LP_CAPITAL) {
                if (currency != address(0)) {
                    TransferHelper.safeTransfer(currency, _to, pendingAmountInUno);
                } else {
                    TransferHelper.safeTransferETH(_to, pendingAmountInUno);
                }
                _withdrawImplement(_to);
            } else {
                if (currency != address(0)) {
                    TransferHelper.safeTransfer(currency, _to, cryptoBalance - MIN_LP_CAPITAL);
                } else {
                    TransferHelper.safeTransferETH(_to, cryptoBalance - MIN_LP_CAPITAL);
                }
                _withdrawImplementIrregular(_to, ((cryptoBalance - MIN_LP_CAPITAL) * 1e18) / lpPriceUno);
            }
        } else {
            if (withdrawRequestPerUser[_to].pendingAmount > 0) {
                _cancelWithdrawRequest(_to);
            }
        }
```

[Uno-Re-02] Staking Locking Risk

Severity Level	**Critical**
Type	Business Security
Lines	CapitalAgent.sol #L237-256
Description	In the CapitalAgent contract, the inaccurate valuation of `totalCapital` could potentially lead to a situation where users withdrawing tokens at a peak may result in subsequent users being unable to withdraw, causing tokens to be locked. For example, if User A deposits 10 ETH when ETH is at 1000U, and User B also deposits 10 ETH, resulting in a `totalCapital` of 20000U. If User A withdraws 10 ETH when ETH is at 2000U, `totalCapital` would become 0, causing User B's deposited ETH to be entirely locked.

```
    function _updatePoolCapital(address _pool, uint256 _amount, bool
isAdd) private {
        address currency = poolInfo[_pool].currency;
        uint256 stakingAmountInUSDC;
        if (currency == USDC_TOKEN) {
            stakingAmountInUSDC = _amount;
        } else {
            stakingAmountInUSDC = currency != address(0)
                ?
IExchangeAgent(exchangeAgent).getNeededTokenAmount(currency,
USDC_TOKEN, _amount)
                :
IExchangeAgent(exchangeAgent).getTokenAmountForETH(USDC_TOKEN,
_amount);}
        if (!isAdd) {
            require(poolInfo[_pool].totalCapital >=
stakingAmountInUSDC, "UnoRe: pool capital overflow");
        }
        poolInfo[_pool].totalCapital = isAdd
            ? poolInfo[_pool].totalCapital + stakingAmountInUSDC
            : poolInfo[_pool].totalCapital - stakingAmountInUSDC;
        totalCapitalStaked = isAdd ? totalCapitalStaked +
stakingAmountInUSDC : totalCapitalStaked - stakingAmountInUSDC;
        emit LogUpdatePoolCapital(_pool, poolInfo[_pool].totalCapital,
totalCapitalStaked);
```

	`` } ```
Recommendation	It is recommended to record `totalCapital` for each pool in the form of staked tokens and calculate MCR and SCR based on the real-time token value.
Status	**Fixed.** The project team modified the related algorithms of `totalCapital` and `totalCapitalStaked` to calculate the USDC value of the corresponding Pool in real time.

```
    function _updatePoolCapital(address _pool, uint256 _amount, bool
isAdd) private {
        if (!isAdd) {
            require(poolInfo[_pool].totalCapital >= _amount, "UnoRe:
pool capital overflow");
        }
        address currency = poolInfo[_pool].currency;
        poolInfo[_pool].totalCapital = isAdd ?
poolInfo[_pool].totalCapital + _amount : poolInfo[_pool].totalCapital
- _amount;
        totalCapitalStakedByCurrency[currency] = isAdd ?
totalCapitalStakedByCurrency[currency] + _amount :
totalCapitalStakedByCurrency[currency] - _amount;
        emit LogUpdatePoolCapital(_pool, poolInfo[_pool].totalCapital,
totalCapitalStakedByCurrency[currency]);
    }
```

[Uno-Re-03] Tx.origin Restriction Issue

Severity Level	High
Type	Business Security
Lines	Rewarder.sol #L64
Description	In the Rewarder contract, the `onReward` function restricts the to address to be only `tx.origin`. This limitation can prevent the use of the `rollOverReward` function in multiple contracts, as well as the `transferFrom` function within the riskPool contract. This limitation will cause direct calls involving these functions to fail.

```
    function onReward(address _to, uint256 _amount) external payable
override onlyPOOL whenNotPaused returns (uint256) {
        require(tx.origin == _to, "UnoRe: must be message sender");
        ISSIP ssip = ISSIP(pool);
        ISSIP.UserInfo memory userInfos = ssip.userInfo(_to);
        ISSIP.PoolInfo memory poolInfos = ssip.poolInfo();
        uint256 accumulatedUno = (userInfos.amount *
uint256(poolInfos.accUnoPerShare)) / ACC_UNO_PRECISION;
        address riskPool = ssip.riskPool();
        if (ssip.userInfo(riskPool).rewardDebt != accumulatedUno) {
            require(userInfos.rewardDebt == accumulatedUno, "UnoRe:
updated rewarddebt incorrectly");
        }
        require(accumulatedUno > _amount, "UnoRe: invalid reward
amount");
```

Recommendation	It is recommended to implement special handling for the riskPool contract.
Status	**Fixed.** The project team added a new `onRewardForRollOver` function and carried out special treatment for rollover.

```
    function onRewardForRollOver(
        address _to,
        uint256 _amount,
        uint256 _accumulatedAmount
    ) external payable onlyPOOL whenNotPaused returns (uint256) {
        ISSIP ssip = ISSIP(pool);
        ISSIP.PoolInfo memory poolInfos = ssip.poolInfo();
        uint256 accumulatedUno = (_accumulatedAmount *
uint256(poolInfos.accUnoPerShare)) / ACC_UNO_PRECISION;
```

```
        require(accumulatedUno > _amount, "UnoRe: invalid reward
amount");
        if (currency == address(0)) {
            require(address(this).balance >= _amount, "UnoRe:
insufficient reward balance");
            TransferHelper.safeTransferETH(_to, _amount);
            return _amount;
        } else {
            require(IERC20(currency).balanceOf(address(this)) >=
_amount, "UnoRe: insufficient reward balance");
            TransferHelper.safeTransfer(currency, _to, _amount);
            return _amount;
        }
    }
```

[Uno-Re-04] Function Call Failed

Severity Level	High
Type	Business Security
Lines	Rewarder.sol #68-75
Description	In the Rewarder contract, the onReward function requires accumulatedUno to be greater than the _amount being claimed. However, since the rollOverReward function acts as a proxy for reinvestment, the accumulatedUno value is 0. As a result, the reinvestment operation cannot be executed correctly.

```
    function onReward(address _to, uint256 _amount) external payable
override onlyPOOL whenNotPaused returns (uint256) {
        require(tx.origin == _to, "UnoRe: must be message sender");
        ISSIP ssip = ISSIP(pool);
        ISSIP.UserInfo memory userInfos = ssip.userInfo(_to);
        ISSIP.PoolInfo memory poolInfos = ssip.poolInfo();
        uint256 accumulatedUno = (userInfos.amount *
uint256(poolInfos.accUnoPerShare)) / ACC_UNO_PRECISION;
        address riskPool = ssip.riskPool();
        if (ssip.userInfo(riskPool).rewardDebt != accumulatedUno) {
            require(userInfos.rewardDebt == accumulatedUno, "UnoRe:
updated rewarddebt incorrectly");
        }
        require(accumulatedUno > _amount, "UnoRe: invalid reward
amount");
```

Recommendation	It is recommended to implement special handling for the riskPool contract.
Status	**Fixed.** The project team added a new onRewardForRollOver function and carried out special treatment for rollover.

```
    function onRewardForRollOver(
        address _to,
        uint256 _amount,
        uint256 _accumulatedAmount
    ) external payable onlyPOOL whenNotPaused returns (uint256) {
        ISSIP ssip = ISSIP(pool);
        ISSIP.PoolInfo memory poolInfos = ssip.poolInfo();
        uint256 accumulatedUno = (_accumulatedAmount *
uint256(poolInfos.accUnoPerShare)) / ACC_UNO_PRECISION;
        require(accumulatedUno > _amount, "UnoRe: invalid reward
```

```
amount");
        if (currency == address(0)) {
            require(address(this).balance >= _amount, "UnoRe:
insufficient reward balance");
            TransferHelper.safeTransferETH(_to, _amount);
            return _amount;
        } else {
            require(IERC20(currency).balanceOf(address(this)) >=
_amount, "UnoRe: insufficient reward balance");
            TransferHelper.safeTransfer(currency, _to, _amount);
            return _amount;
        }
    }
```

[Uno-Re-05] Withdrawable Stake Before Initiating Compensation

Severity Level	High
Type	Business Security
Lines	SingleSidedInsurancePool.sol #340-371 SingleSidedReinsurancePool.sol #L249-277
Description	In the SingleSidedInsurancePool contract, when calculating the user's pending rewards using `_updateReward`, the rewards from the pending state are not deducted. This allows staked assets in the pending state to continue accruing rewards. If a staker does not withdraw their stake after the pending period expires, they can continue to receive staking rewards indefinitely. However, when the community identifies a large insurance policy requiring a claims voting process, stakers with pending stakes can immediately withdraw them without waiting for the lockTime.

```
    function leaveFromPending() external override isStartTime
whenNotPaused nonReentrant {
        require(block.timestamp -
userInfo[msg.sender].lastWithdrawTime >= lockTime, "UnoRe: Locked
time");
        _harvest(msg.sender);
        uint256 amount = userInfo[msg.sender].amount;
        (uint256 pendingAmount, , ) =
IRiskPool(riskPool).getWithdrawRequest(msg.sender);
        uint256 accumulatedUno = (amount *
uint256(poolInfo.accUnoPerShare)) / ACC_UNO_PRECISION;
        userInfo[msg.sender].rewardDebt =
            accumulatedUno -
            ((pendingAmount * uint256(poolInfo.accUnoPerShare)) /
ACC_UNO_PRECISION);
        (uint256 withdrawAmount, uint256 withdrawAmountInUNO) =
IRiskPool(riskPool).leaveFromPending(msg.sender);
        userInfo[msg.sender].amount = amount - withdrawAmount;
        ICapitalAgent(capitalAgent).SSIPWithdraw(withdrawAmountInUNO)
;
        emit LogLeaveFromPendingSSIP(msg.sender, riskPool,
withdrawAmount, withdrawAmountInUNO);
    }
```

Recommendation	It is recommended to add a BOT interface to the `leaveFromPending` function to assist accounts with long-unclaimed pending staked tokens in withdrawing the corresponding staked tokens.
Status	**Fixed.** The project team has modified the corresponding code, and rewards in the pending status will not be calculated. This calculation method can help mitigate arbitrage to some extent when withdrawing pending stake.

```
    function _updateReward(address _to) internal returns (uint256) {
        uint256 requestTime;
        (, requestTime, ) =
IRiskPool(riskPool).getWithdrawRequest(_to);
        if (requestTime > 0) {
            return 0;
        }
```

[Uno-Re-06] TotalCapital Are Inaccurate

Severity Level	Medium
Type	Business Security
Lines	CapitalAgent.sol #L237-256
Description	In the CapitalAgent contract, `totalCapital` and `totalCapitalStaked` only record the token value at the time of staking and cannot be updated in real-time. This could result in a mismatch between the actual staked value and the recorded staked value.

```
    function _updatePoolCapital(address _pool, uint256 _amount, bool isAdd) private {
        address currency = poolInfo[_pool].currency;
        uint256 stakingAmountInUSDC;
        if (currency == USDC_TOKEN) {
            stakingAmountInUSDC = _amount;
        } else {
            stakingAmountInUSDC = currency != address(0)
                ? IExchangeAgent(exchangeAgent).getNeededTokenAmount(currency, USDC_TOKEN, _amount)
                : IExchangeAgent(exchangeAgent).getTokenAmountForETH(USDC_TOKEN, _amount);
        }
        if (!isAdd) {
            require(poolInfo[_pool].totalCapital >= stakingAmountInUSDC, "UnoRe: pool capital overflow");
        }
        poolInfo[_pool].totalCapital = isAdd
            ? poolInfo[_pool].totalCapital + stakingAmountInUSDC
            : poolInfo[_pool].totalCapital - stakingAmountInUSDC;
        totalCapitalStaked = isAdd ? totalCapitalStaked + stakingAmountInUSDC : totalCapitalStaked - stakingAmountInUSDC;
        emit LogUpdatePoolCapital(_pool, poolInfo[_pool].totalCapital, totalCapitalStaked);
    }
```

Recommendation	It is recommended to calculate the staked token value in real-time to determine MCR and SCR.

Status	**Fixed.** The project team modified the related algorithms of `totalCapital` and `totalCapitalStaked` to calculate the USDC value of the corresponding Pool in real time.

```
    function _updatePoolCapital(address _pool, uint256 _amount, bool 
isAdd) private {
        if (!isAdd) {
            require(poolInfo[_pool].totalCapital >= _amount, "UnoRe: 
pool capital overflow");
        }
        address currency = poolInfo[_pool].currency;
        poolInfo[_pool].totalCapital = isAdd ? 
poolInfo[_pool].totalCapital + _amount : poolInfo[_pool].totalCapital 
- _amount;
        totalCapitalStakedByCurrency[currency] = isAdd ? 
totalCapitalStakedByCurrency[currency] + _amount : 
totalCapitalStakedByCurrency[currency] - _amount;
        emit LogUpdatePoolCapital(_pool, poolInfo[_pool].totalCapital, 
totalCapitalStakedByCurrency[currency]);
    }
```

[Uno-Re-07] Chainlink DOS Attack

Severity Level	Medium
Type	General Vulnerability
Lines	SingleSideInsurancePool.sol #L473-488
Description	In the SingleSideInsurancePool contract, the `requestPayout` function lacks a call lock for `_policyId`, allowing unlimited payout requests for the same `_policyId`. As `assertTruth` requires the contract to pay a bond, an attacker could perform a DOS attack by repeatedly calling the `requestPayout` function, consuming the contract's bond tokens.

```
oo.getMinimumBond(address(defaultCurrency));
            assertionId = oo.assertTruth(
                abi.encodePacked(
                    "Insurance contract is claiming that insurance event
",
                    " had occurred as of ",
                    ClaimData.toUtf8BytesUint(block.timestamp),
                    "."
                ),
                _to,
                address(this),
                escalationManager,
                uint64(assertionliveTime),
                defaultCurrency,
                bond,
                defaultIdentifier,
                bytes32(0) // No domain.
            );
```

Recommendation	1.Implement a lock for the same `_policyId`, and if subsequent requests are denied, let the administrator lift the lock. 2.Users should bear this portion of the bond.
Status	**Fixed.** The project party added the isRequestInit variable to lock the policy that initiated the claim request, and unlocked it when the claim failed.

```
isRequestInit[_policyId] = true;
```

[Uno-Re-08] Missing Approval

Severity Level	**Medium**
Type	Business Security
Lines	SingleSideInsurancePool.sol #L473-488
Description	In the SingleSidedInsurancePool contract, when invoking the `assertTruth` function of the oracle, a `safeTransferFrom` is performed. However, due to the absence of approval for the oracle within the SingleSidedInsurancePool contract, the delegated transfer will directly result in a failed call.

```
oo.getMinimumBond(address(defaultCurrency));
            assertionId = oo.assertTruth(
                abi.encodePacked(
                    "Insurance contract is claiming that insurance event
",
                    " had occurred as of ",
                    ClaimData.toUtf8BytesUint(block.timestamp),
                    "."
                ),
                _to,
                address(this),
                escalationManager,
                uint64(assertionliveTime),
                defaultCurrency,
                bond,
                defaultIdentifier,
                bytes32(0) // No domain.
            );
```

Recommendation	It is recommended to add approval for the oracle.
Status	**Fixed.** The project team adds corresponding authorization operations.

```
defaultCurrency.approve(address(optimisticOracle), bond);
```

[Uno-Re-09] Reward Claiming Duplication

Severity Level	**Medium**
Type	Business Security
Lines	SingleSidedInsurancePool.sol #357-371 SingleSidedReinsurancePool.sol #L265-277
Description	In the SingleSidedInsurancePool contract, the `leaveFromPending` function deducts existing user staking based on the pending status. When a user withdraws all staking, the returned `withdrawAmount` may not necessarily be equal to the staked amount due to the `MIN_LP_CAPITAL` limit within the riskPool contract. However, in the calculation of `rewardDebt`, the `accumulatedUno` is computed using the staked amount, causing the `rewardDebt` to be zeroed. When `rewardDebt` is zeroed but user staking is not, it results in an additional portion of rewards being obtained out of thin air, allowing unlimited claiming from the reward pool. Similar issues are present in the SingleSidedReinsurancePool contract.

```
        uint256 accumulatedUno = (amount *
uint256(poolInfo.accUnoPerShare)) / ACC_UNO_PRECISION;
        userInfo[msg.sender].rewardDebt =
            accumulatedUno -
            ((pendingAmount * uint256(poolInfo.accUnoPerShare)) /
ACC_UNO_PRECISION);
        (uint256 withdrawAmount, uint256 withdrawAmountInUNO) =
IRiskPool(riskPool).leaveFromPending(msg.sender);
        userInfo[msg.sender].amount = amount - withdrawAmount;
```

Recommendation	It is recommended to use the corresponding `withdrawAmount` for updating `rewardDebt` to address this issue.
Status	**Fixed.** The project team updated the debt using the actual withdrawal amount.

```
        (uint256 withdrawAmount, uint256 withdrawAmountInUNO) =
IRiskPool(riskPool).leaveFromPending(msg.sender, _amount);
        ICapitalAgent(capitalAgent).SSIPWithdraw(withdrawAmountInUNO)
;

        uint256 accumulatedUno = (amount *
uint256(poolInfo.accUnoPerShare)) / ACC_UNO_PRECISION;
        userInfo[msg.sender].rewardDebt =
            accumulatedUno -
```

```
            ((withdrawAmount * uint256(poolInfo.accUnoPerShare)) /
ACC_UNO_PRECISION);
        userInfo[msg.sender].amount = amount - withdrawAmount;
```

[Uno-Re-10] Signature Reuse Risk

Severity Level	Low
Type	Business Security
Lines	SalesPolicy.sol #L122-134
Description	In the SalesPolicy contract, the `buyPolicy` function requires signature verification through the signer, but there is no prevention mechanism for signature reuse. Consequently, the same signature can be used to purchase the same policy multiple times.

```
        address _signer = getSender(
            _policyPriceInUSDC,
            _protocols,
            _coverageDuration,
            _coverageAmount,
            _signedTime,
            _premiumCurrency,
            r,
            s,
            v
        );
        require(_signer != address(0) && _signer == signer, "UnoRe: invalid signer");
        require(_signedTime <= block.timestamp && block.timestamp - _signedTime < maxDeadline, "UnoRe: signature expired");
```

Recommendation	1.The signature should contain `msg.sender` to prevent other users from using the signature. 2.Use mapping to record the used hash, and before calling the function, check whether the signature is used
Status	**Fixed.** The project party added replay verification in the getSender function.

```
        require(usedHash[msgHash] == address(0), "Already used hash");
        usedHash[msgHash] = sender;
```

[Uno-Re-11] Expired Verification Is Invalid

Severity Level	Low
Type	Business Security
Lines	SingleSideInsurancePool.sol #L461-466
Description	In the SingleSideInsurancePool contract, the `requestPayout` function fetches data status for `_policyId` only from the salesPolicy contract without updating it beforehand. This could lead to inaccurate information about whether the policy has expired.

```
    function requestPayout(uint256 _policyId, uint256 _amount, address
_to) public isAlive returns (bytes32 assertionId) {
        (address salesPolicy, , ) =
ICapitalAgent(capitalAgent).getPolicyInfo();
        require(IERC721(salesPolicy).ownerOf(_policyId) == msg.sender,
"UnoRe: not owner of policy id");
        (uint256 _coverageAmount, , , bool _exist, bool _expired) =
ISalesPolicy(salesPolicy).getPolicyData(_policyId);
        require(_amount <= _coverageAmount, "UnoRe: amount exceeds
coverage amount");
        require(_exist && !_expired, "UnoRe: policy expired or not
exist");
```

Recommendation	It is recommended to call `updatePolicyStatus` before fetching PolicyData.
Status	**Fixed.** The project team added an update before the expire check.

```
    function initRequest(uint256 _policyId, uint256 _amount, address
_to) public whenNotPaused returns (bytes32 assertionId) {
        (address salesPolicy, , ) =
ICapitalAgent(capitalAgent).getPolicyInfo();
        ICapitalAgent(capitalAgent).updatePolicyStatus(_policyId);
        (uint256 _coverageAmount, , , bool _exist, bool _expired) =
ISalesPolicy(salesPolicy).getPolicyData(_policyId);
```

[Uno-Re-12] Redundant Increment

Severity Level	Info
Type	Coding Conventions
Lines	ClaimProcessor.sol #L34-42
Description	In the ClaimProcessor contract, the `lastIndex` in the `requestPolicyId` function increments twice instead of once each time it grows. This leads to a significant number of empty array members in assertions.

```
    function requestPolicyId(uint256 _policyId) external
onlyRole(SSIP_ROLE) {
        uint256 _lastIndex = ++lastIndex;
        Claim memory _claim = assertion[_lastIndex];
        _claim.ssip = msg.sender;
        _claim.policyId = _policyId;
        assertion[_lastIndex] = _claim;
        lastIndex++;
        emit PolicyRequested(msg.sender, _lastIndex, _policyId);
    }
```

Recommendation	It is recommended to remove `++lastIndex` or `lastIndex++`.
Status	**Fixed.** This contract has been deprecated.

[Uno-Re-13] Meaningless Data Comparison

Severity Level	Info
Type	Coding Conventions
Lines	Rewarder.sol #63-74
Description	In the Rewarder contract, the `onReward` function compares the `rewardDebt` of the riskPool contract with the user's data. However, since the riskPool contract does not have `userInfo` data, this comparison doesn't have any practical significance.

```
    function onReward(address _to, uint256 _amount) external payable
override onlyPOOL whenNotPaused returns (uint256) {
        require(tx.origin == _to, "UnoRe: must be message sender");
        ISSIP ssip = ISSIP(pool);
        ISSIP.UserInfo memory userInfos = ssip.userInfo(_to);
        ISSIP.PoolInfo memory poolInfos = ssip.poolInfo();
        uint256 accumulatedUno = (userInfos.amount *
uint256(poolInfos.accUnoPerShare)) / ACC_UNO_PRECISION;
        address riskPool = ssip.riskPool();
        if (ssip.userInfo(riskPool).rewardDebt != accumulatedUno) {
            require(userInfos.rewardDebt == accumulatedUno, "UnoRe:
updated rewarddebt incorrectly");
        }
```

Recommendation	It is recommended to review the corresponding logic to identify any issues.
Status	**Fixed.** The project team deleted the corresponding meaningless data.

[Uno-Re-14] Redundant Code

Severity Level	Info
Type	Coding Conventions
Lines	CapitalAgent.sol #L80
Description	In the CapitalAgent contract, `UNO_TOKEN` is only initialized but not utilized.

```
    function initialize(
        address _exchangeAgent,
        address _UNO_TOKEN,
        address _USDC_TOKEN,
        address _multiSigWallet,
        address _operator
    ) external initializer {
        require(_exchangeAgent != address(0), "UnoRe: zero 
exchangeAgent address");
        require(_UNO_TOKEN != address(0), "UnoRe: zero UNO address");
        require(_USDC_TOKEN != address(0), "UnoRe: zero USDC address");
        require(_multiSigWallet != address(0), "UnoRe: zero 
multisigwallet address");
        exchangeAgent = _exchangeAgent;
        UNO_TOKEN = _UNO_TOKEN;
        USDC_TOKEN = _USDC_TOKEN;
        operator = _operator;
        __ReentrancyGuard_init();
        __Ownable_init(_multiSigWallet);
    }
```

Recommendation	It is recommended to remove redundant code.
Status	**Fixed.** The project team deleted the corresponding redundant code.

[Uno-Re-15] Missing Event Trigger

Severity Level	Info
Type	Coding Conventions
Lines	SalesPolicyFactory.sol#L108-135
Description	In the SalesPolicyFactory contract, there are several functions that are called by the owner to modify critical contract variables, but these functions do not trigger any events. This is not considered a good practice and can hinder the ability to obtain contract information.

```
    function setExchangeAgentInPolicy(address _exchangeAgent) external onlyOwner {
        require(_exchangeAgent != address(0), "UnoRe: zero address");
        ISalesPolicy(salesPolicy).setExchangeAgent(_exchangeAgent);
    }
    function setBuyPolicyMaxDeadlineInPolicy(uint256 _maxDeadline) external onlyOwner {
        require(_maxDeadline > 0, "UnoRe: zero max deadline");
        ISalesPolicy(salesPolicy).setBuyPolicyMaxDeadline(_maxDeadline);
    }
    function setPremiumPoolInPolicy(address _premiumPool) external onlyOwner {
        require(_premiumPool != address(0), "UnoRe: zero address");
        ISalesPolicy(salesPolicy).setPremiumPool(_premiumPool);
    }
    function setSignerInPolicy(address _signer) external onlyOwner {
        require(_signer != address(0), "UnoRe: zero address");
        ISalesPolicy(salesPolicy).setSigner(_signer);
    }
    function setCapitalAgentInPolicy(address _capitalAgent) external onlyOwner {
        require(_capitalAgent != address(0), "UnoRe: zero address");
        ISalesPolicy(salesPolicy).setCapitalAgent(_capitalAgent);
    }
    function setProtocolURIInPolicy(string memory _uri) external onlyOwner {
        ISalesPolicy(salesPolicy).setProtocolURI(_uri);
    }
```

Recommendation	It is recommended to emit events when modifying critical variables is a recommended practice as it provides a standardized way to capture and communicate important changes within the contract. Events enable transparency and allow external systems and users to easily track and react to these modifications.
Status	**Fixed.** The project team added the corresponding event.

3 Appendix

3.1 Vulnerability Assessment Metrics and Status in Smart Contracts

3.1.1 Metrics

In order to objectively assess the severity level of vulnerabilities in blockchain systems, this report provides detailed assessment metrics for security vulnerabilities in smart contracts with reference to CVSS 3.1 (Common Vulnerability Scoring System Ver 3.1).

According to the severity level of vulnerability, the vulnerabilities are classified into four levels: "critical", "high", "medium" and "low". It mainly relies on the degree of impact and likelihood of exploitation of the vulnerability, supplemented by other comprehensive factors to determine of the severity level.

Impact / Likelihood	Severe	High	Medium	Low
Probable	Critical	High	Medium	Low
Possible	High	Medium	Medium	Low
Unlikely	Medium	Medium	Low	Info
Rare	Low	Low	Info	Info

3.1.2 Degree of impact

- **Severe**

Severe impact generally refers to the vulnerability can have a serious impact on the confidentiality, integrity, availability of smart contracts or their economic model, which can cause substantial economic losses to the contract business system, large-scale data disruption, loss of authority management, failure of key functions, loss of credibility, or indirectly affect the operation of other smart contracts associated with it and cause substantial losses, as well as other severe and mostly irreversible harm.

- **High**

High impact generally refers to the vulnerability can have a relatively serious impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a greater economic loss, local functional unavailability, loss of credibility and other impact to the contract business system.

- **Medium**

Medium impact generally refers to the vulnerability can have a relatively minor impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a small amount of economic loss to the contract business system, individual business unavailability and other impact.

- **Low**

Low impact generally refers to the vulnerability can have a minor impact on the smart contract, which can pose certain security threat to the contract business system and needs to be improved.

3.1.4 Likelihood of Exploitation

- **Probable**

Probable likelihood generally means that the cost required to exploit the vulnerability is low, with no special exploitation threshold, and the vulnerability can be triggered consistently.

- **Possible**

Possible likelihood generally means that exploiting such vulnerability requires a certain cost, or there are certain conditions for exploitation, and the vulnerability is not easily and consistently triggered.

- **Unlikely**

Unlikely likelihood generally means that the vulnerability requires a high cost, or the exploitation conditions are very demanding and the vulnerability is highly difficult to trigger.

- **Rare**

Rare likelihood generally means that the vulnerability requires an extremely high cost or the conditions for exploitation are extremely difficult to achieve.

3.1.5 Fix Results Status

Status	Description
Fixed	The project party fully fixes a vulnerability.
Partially Fixed	The project party did not fully fix the issue, but only mitigated the issue.
Acknowledged	The project party confirms and chooses to ignore the issue.

3.2 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
		Third-party Protocol Interface Consistency
3	Business Security	Business Logics
		Business Implementations
		Manipulable Token Price
		Centralized Asset Control
		Asset Tradability
		Arbitrage Attack

Beosin classified the security issues of smart contracts into three categories: Coding Conventions, General Vulnerability, Business Security. Their specific definitions are as follows:

- **Coding Conventions**

Audit whether smart contracts follow recommended language security coding practices. For example, smart contracts developed in Solidity language should fix the compiler version and do not use deprecated keywords.

- **General Vulnerability**

General Vulnerability include some common vulnerabilities that may appear in smart contract projects. These vulnerabilities are mainly related to the characteristics of the smart contract itself, such as integer overflow/underflow and denial of service attacks.

- **Business Security**

Business security is mainly related to some issues related to the business realized by each project, and has a relatively strong pertinence. For example, whether the lock-up plan in the code match the white paper, or the flash loan attack caused by the incorrect setting of the price acquisition oracle.

*Note that the project may suffer stake losses due to the integrated third-party protocol. This is not something Beosin can control. Business security requires the participation of the project party. The project party and users need to stay vigilant at all times.

3.3 Disclaimer

The Audit Report issued by Beosin is related to the services agreed in the relevant service agreement. The Project Party or the Served Party (hereinafter referred to as the "Served Party") can only be used within the conditions and scope agreed in the service agreement. Other third parties shall not transmit, disclose, quote, rely on or tamper with the Audit Report issued for any purpose.

The Audit Report issued by Beosin is made solely for the code, and any description, expression or wording contained therein shall not be interpreted as affirmation or confirmation of the project, nor shall any warranty or guarantee be given as to the absolute flawlessness of the code analyzed, the code team, the business model or legal compliance.

The Audit Report issued by Beosin is only based on the code provided by the Served Party and the technology currently available to Beosin. However, due to the technical limitations of any organization, and in the event that the code provided by the Served Party is missing information, tampered with, deleted, hidden or subsequently altered, the audit report may still fail to fully enumerate all the risks.

The Audit Report issued by Beosin in no way provides investment advice on any project, nor should it be utilized as investment suggestions of any type. This report represents an extensive evaluation process designed to help our customers improve code quality while mitigating the high risks in blockchain.

3.4 About Beosin

Beosin is the first institution in the world specializing in the construction of blockchain security ecosystem. The core team members are all professors, postdocs, PhDs, and Internet elites from world-renowned academic institutions. Beosin has more than 20 years of research in formal verification technology, trusted computing, mobile security and kernel security, with overseas experience in studying and collaborating in project research at well-known universities. Through the security audit and defense deployment of more than 2,000 smart contracts, over 50 public blockchains and wallets, and nearly 100 exchanges worldwide, Beosin has accumulated rich experience in security attack and defense of the blockchain field, and has developed several security products specifically for blockchain.

BEOSIN
Blockchain Security

Official Website
https://www.beosin.com

Telegram
https://t.me/beosin

Twitter
https://twitter.com/Beosin_com

Email
service@beosin.com