BEOSIN
Blockchain Security

# Uno-Re-V2

Smart Contract Security Audit

No. 202403101438

Mar 10th, 2024

SECURING BLOCKCHAIN ECOSYSTEM

WWW.BEOSIN.COM

# Contents

# Summary of Audit Results

After auditing,1 High-risk and 2 Medium-risk items were identified in the Uno-Re-V2 project. Specific audit details will be presented in the Findings section. Users should pay attention to the following aspects when interacting with this project:

| | |
|---|---|
| High | Fixed: 1 Acknowledged: 0 |
| Medium | Fixed: 2 Acknowledged: 0 |

- **Risk Description:**

1. When dealing with the third risk item, the project team added setting functions for several key parameters, which increased a certain degree of centralization risk.

- **Project Description:**

**Business overview**

This audit is an incremental audit of Uno-Re, focusing on the following main areas: the fallback mechanism for UMIP (claims dispute assessment) failure, the multi-isolation pool claim mechanism, and the data migration mechanism for the Binance Smart Chain (BSC).

The fallback mechanism refers to the ability to use the `setAssertionIdApproval` function within the EscalationManager to directly determine the success or failure of a claim when the optimistic oracle's claim assessment fails. This ensures that the claim functionality remains intact in case of issues with the mechanism or malicious disputers.

The multi-isolation pool claim mechanism allows users to select multiple collateral pools for claiming, rather than being limited to a single collateral pool. This helps alleviate situations where funds in a single collateral pool may be insufficient.

The data migration mechanism for the BSC ensures that existing users can utilize their funds to participate in the new version of the staking contract.

# 1 Overview

## 1.1 Project Overview

| | |
|---|---|
| **Project Name** | Uno-Re-V2 |
| **Project Language** | Solidity |
| **Platform** | Ethereum |
| **Github** | https://github.com/Uno-Re/SSIP-SSRP-contracts/tree/main |
| **Audit Scope** | SSIP-SSRP-contracts:<br>./contracts/SingleSidedReinsurancePool.sol<br>./contracts/SingleSidedInsurancePoolBSC.sol<br>./contracts/SingleSidedInsurancePool.sol<br>./contracts/CapitalAgent.sol<br>./contracts/uma/PayoutRequest.sol<br>./contracts/uma/EscalationManager.sol<br>./contracts/interfaces/ICapitalAgent.sol<br>./contracts/interfaces/IPayoutRequest.sol |
| **commit** | 589dcc9697622904bd913b4850e2f93c4c9d6724<br>e3ace2194c005949338bdd4dfd1015e2189a7aab |

## 1.2 Audit Overview

Audit work duration: Feb 29, 2024 – Mar 10, 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-V2-01 | Users cannot make multiple claims | High | Fixed |
| Uno-Re-V2-02 | Inconsistent precision | Medium | Fixed |
| Uno-Re-V2-03 | No post-upgrade stake processing | Medium | Fixed |

# Finding Details:

## [Uno-Re-V2-01] Users cannot make multiple claims

| | |
|---|---|
| **Severity Level** | High |
| **Type** | Business Security |
| **Lines** | PayoutRequest.sol#119-125 |
| **Description** | In this update, to address the issue of insufficient funds in the single risk pool, the project team has added a batch claiming feature. However, due to insufficiently rigorous code design, this may result in users' funds being locked. Specifically, in the `initRequest` function of the PayoutRequest contract, when a user calls `initRequest` to claim and the UMA oracle determines the claim to be valid, the optimisticOracle will call the `assertionResolvedCallback` function, setting policy.settled to true. If a user only makes one claim, there won't be an issue. However, if multiple claims are made for the same `policyId`, it will result in failure. This is because the line of code `if (policy.settled) return;` will cause the transaction to rollback, thereby preventing subsequent claims from succeeding. |
| **Recommendation** | It is recommended to delete the Policy check and policy.settled setting in the `assertionResolvedCallback` function, and check the usage of `_assertionId` to |

```
    function assertionResolvedCallback(bytes32 _assertionId, bool
_assertedTruthfully) external whenNotPaused {
        require(!isUMAFailed, "RPayout: pool failed");
        require(msg.sender == address(optimisticOracle),
"RPayout: !optimistic oracle");
        uint256 _policyId = assertedPolicies[_assertionId];
        if (_assertedTruthfully) {
            Policy storage policy = policies[_policyId];
            if (policy.settled) return;
            policy.settled = true;
            ssip.settlePayout(_policyId, policy.payoutAddress,
policy.insuranceAmount);
        } else {
            isRequestInit[_policyId] = false;
        }
    }
```

| | |
|---|---|
| | prevent multiple uses. |
| **Status** | **Fixed.** The project team removed the policy check and added a new check for assertedPolicies used. |

```
            if (_assertionApproved.exist
&& !_assertionApproved.approved) {
                return;
            }
            if (_policyData.settled) return;
            assertedPolicies[_assertionId].settled = true;
```

## [Uno-Re-V2-02] Inconsistent precision

| | |
|---|---|
| **Severity Level** | **Medium** |
| **Type** | Business Security |
| **Lines** | CapitalAgent.sol #L278-296<br>PayoutRequest .sol #73-75 |
| **Description** | In the CapitalAgent contract, when the `SSIPPolicyClaim` function is invoked, the `_withdrawAmount` parameter represents the amount of collateral tokens to be paid out from the pool by `msg.sender`, with the unit being the pool's currency. However, in the subsequent `SSIPPolicyClaim` function calls, variables like `_coverageAmount` and `_claimedAmount` are computed against `_withdrawAmount` (based on the MLR calculation, it's known that the unit of `_coverageAmount` is USDC). If users apply for payouts from different types of collateral pools, it could lead to mixing different currencies within different pools, thereby causing subsequent payout functionality to malfunction. Similar issues exist within the `initRequest` function of PayoutRequest contract.<br>CapitalAgent: |

```
    function SSIPPolicyCaim(uint256 _withdrawAmount, uint256 _policyId,
bool _isNotMigrate) external override nonReentrant {
        require(poolInfo[msg.sender].exist, "UnoRe: no exist ssip");
        _updatePoolCapital(msg.sender, _withdrawAmount, false);
        if (_isNotMigrate) {
            _SSIPPolicyClaim(_withdrawAmount, _policyId);
        }
    }
    function _SSIPPolicyClaim(uint256 _withdrawAmount, uint256
_policyId) internal {
        address _salesPolicyAddress = policyInfo.policy;
        (uint256 _coverageAmount, , , , ) =
ISalesPolicy(_salesPolicyAddress).getPolicyData(_policyId);
        uint256 _claimed =
claimedAmount[_salesPolicyAddress][_policyId];
        require(_coverageAmount >= _withdrawAmount + _claimed, "UnoRe:
coverage amount is less");
        claimedAmount[_salesPolicyAddress][_policyId] +=
_withdrawAmount;
```

```
        bool _isFinished = !(_coverageAmount > (_withdrawAmount +
_claimed));
        if (_isFinished) {
            _markToClaimPolicy(_policyId, _coverageAmount);
        }
    }
```

PayoutRequest:

```
        uint256 _claimed =
ICapitalAgent(capitalAgent).claimedAmount(salesPolicy, _policyId);
        (uint256 _coverageAmount, , , bool _exist, bool _expired) =
ISalesPolicy(salesPolicy).getPolicyData(_policyId);
        require(_amount + _claimed <= _coverageAmount, "UnoRe: amount
exceeds coverage amount");
```

**Recommendation**

It is recommended to calculate `_withdrawAmount` and `_claimedAmount` in the `SSIPPolicyClaim` function, as well as the `_amount` and `_claimed` variables in `initRequest`, based on price calculations before comparing them against `_coverageAmount`.

**Status**

**Fixed.** The project team records the tokens claimed each time in the form of USDC.

```
        uint256 _claimed =
claimedAmount[_salesPolicyAddress][_policyId];
        address _poolCurrency = poolInfo[msg.sender].currency;
        uint256 usdcTokenAmount =
IExchangeAgent(exchangeAgent).getNeededTokenAmount(_poolCurrency,
usdcToken, _withdrawAmount);
        require(_coverageAmount >= usdcTokenAmount + _claimed, "UnoRe:
coverage amount is less");
        claimedAmount[_salesPolicyAddress][_policyId] +=
usdcTokenAmount;
```

## [Uno-Re-V2-03] No post-upgrade stake processing

| | |
|---|---|
| **Severity Level** | **Medium** |
| **Type** | Business Security |
| **Lines** | SingleSidedInsurancePool.sol #L11-17 |
| **Description** | In the SingleSidedInsurancePoolBSC contract, the contract directly modifies the `amount` and `rewardDebt` in userInfo without comparing it with the corresponding current ACC and adding capital data. This will lead to the following three problems:<br>1. `userInfo[_user].rewardDebt` directly inherits the original `_rewardDebt` from V2. Since the ACC of the two pools are not consistent, the `rewardDebt` calculated for the same amount is not the same. For example, if the ACC in the V2 pool is 5000 and in the V3 pool is 1000 (which is highly likely), then for an amount of 100, the `rewardDebt` in V2 would be 500000 while in V3 it would only be 100000. Consequently, if the `rewardDebt` directly passed from V2 is used for subsequent reward calculations, it will result in an error.<br>2. If a user stakes in V3 after it's enabled, and also has stakes in V2, calling the `setUserDetails` function will overwrite the user's original stake information in V3.<br>3. If the stake is upgraded within the SSIP pool, the corresponding capital information in the capitalAgent will not be updated accordingly. This will lead to errors in MCR and SCR information, among others.<br>`function setUserDetails(address _user, uint256 _amount, uint256 _rewardDebt) external onlyRole(ADMIN_ROLE) roleLockTimePassed(ADMIN_ROLE) { userInfo[_user].amount = _amount; userInfo[_user].rewardDebt = _rewardDebt; IRiskPool(riskPool).enter(_user, _amount); emit LogUserUpdated(address(this), _user, _amount); }` |
| **Recommendation** | It is recommended to process the remaining data after upgrading the BSC stake. |
| **Status** | **Fixed.** The project team added a setting function to each data to manually upgrade the BSC data.<br>SingleSidedInsurancePool:<br>`function setAccUnoPerShare( uint256 _accUnoPerShare,` |

```
        uint256 _lastRewardBlock
    ) external onlyRole(ADMIN_ROLE) roleLockTimePassed(ADMIN_ROLE) {
        poolInfo.accUnoPerShare = _accUnoPerShare;
        poolInfo.lastRewardBlock = _lastRewardBlock;
    }
```

CapitalAgent:

```
    function setPoolCapital(address _ssip, uint256 _capital)
external onlyRole(ADMIN_ROLE) nonReentrant {
        require(poolInfo[_ssip].exist, "UnoRe: no exit pool");
        address currency = poolInfo[_ssip].currency;
        totalCapitalStakedByCurrency[currency] += _capital;
        poolInfo[_ssip].totalCapital = _capital;
    }
```

# 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.

| Likelihood \ Impact | 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.

### 3.1.3 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.

### 3.1.4 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