UnoRe

AUDIT REPORT

Auditors : Dimitar Dimitrov & Chrisdior

Disclaimer

This report containing confidential information can be used internally by the Customer, or it can be disclosed publicly after all the vulnerabilities are fixed - upon the decision of the Customer.

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts.

Document Changelog:

19th December 2023	**Audit Kickoff and Scoping**

Severity classification

Severity	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic, and reputation damage of a successful attack
Likelihood - the chance that a particular vulnerability gets discovered and exploited
Severity - the overall criticality of the risk

Security Review Summary

Previously Reviewed commit hash - 2ca5fb57b2b1e23d3f7a8d049a65ce28f2ec0eee
Latest Reviewed commit hash - db7fed08abf40dabfd8598b936ac77de521ee9ab

Audit Scope

The following smart contracts were in scope of the audit:

Precommit hash: 2ca5fb57b2b1e23d3f7a8d049a65ce28f2ec0eee

- SingleSidedInsurancePool
- SingleSidedReinsurancePool
- RiskPoolFactory
- RiskPool
- RewarderFactory
- Rewarder
- CapitalAgent
- SalesPolicyFactory
- SalesPolicy
- PremiumPool
- ExchangeAgent
- EscalationManager
- PayoutRequest
- MultiSigWallet

The following number of issues were found, categorized by their severity:

- Critical: 1 issue
- High: 1 issue

- Medium: 2 issues
- Low: 6 issues
- Informational: 6 issues

Summary Table of Our Findings :

ID	TITLE	Severity	Status
[C-01]	Pausing mechanism will leave users' funds stuck	Critical	Fixed
[H-01]	Deadline check is not effective	High	Fixed
[M-01]	Insufficient input validation	Medium	Fixed
[M-02]	Unsafe downcasting can lead to errors	Medium	Fixed
[L-01]	Use Ownable2StepUpgradeable instead of OwnableUpgradeable contract	Low	Fixed
[L-02]	Inconsistent 0 address checks	Low	Fixed
[L-03]	Missing array length check	Low	Fixed
[L-04]	Mismatch between NatSpec and the actual code	Low	Fixed
[L-05]	Dangerous role setting	Low	Fixed
[L-06]	Direct usage of ecrecover allows signature malleability	Low	Fixed
[I-01]	Wrong event emitted	Informational	Fixed

[I-02]	Unused Code	Informational	Fixed
[I-03]	NatSpec docs are incomplete	Informational	Fixed
[I-04]	Typos	Informational	Fixed
[I-05]	Redundant code	Informational	Fixed
[I-06]	Variables can be turned immutable	Informational	Fixed

Initial Report Detailed Findings

[C-01] Pausing mechanism will leave users' funds stuck

Impact: High because users will lose all of their funds

Likelihood: High because there is no way to retrieve their funds when the contract is paused

Description

All of the contracts inherit the Pausable library which allows to pause the project. This is a helpful feature in different cases (e.g. emergency) as it allows the admin to pause specific functionalities. However, the way it is implemented is problematic and will lead to users losing funds.

For example, the SingleSidedInsurancePool can be paused and the following functions have the whenNotPaused modifier:

- leaveFromPoolInPending
- leaveFromPending
- lpTransfer
- harvest

However, the enterInPool doesn't have it:

```
function enterInPool(uint256 _amount) external override isStartTime isAlive
nonReentrant {
```

This means that, if the protocol is paused, users will be able to deposit funds into the protocol, but there is absolutely no way to get back their funds. This is also true for the SSRP and the PremiumPool. This will result in users losing 100% of their funds.

This opens up another attack vector, where the protocol owner can decide if the users are able to withdraw/claim any funds from it. There is also the possibility that an admin pauses the contracts and renounces ownership, which will leave the funds stuck in the contract forever.

Recommendations

Add the whenNotPaused modifier to deposit functionalities and consider removing it from withdraw functions as users should be able to withdraw their funds anytime.

[H-01] Deadline check is not effective

Impact: High, because the transaction might be left hanging in the mempool and be executed way later than the user wanted at a possibly worse price

Likelihood: Medium, because there is a great chance that the user won't adjust the gas price to be lucrative for the validators to include its transaction fast

The deadline parameter in swapExactTokensForTokensSupportingFeeOnTransferTokens, swapExactETHForTokensSupportingFeeOnTransferTokens, and swapExactTokensForETHSupportingFeeOnTransferTokens() which are called in the convert methods inside ExchangeAgent.sol is hardcoded to block.timestamp.

Example in _convertTokenForETH:

```
function _convertTokenForETH(
       address _dexAddress,
       address _token,
       uint256 _convertAmount,
       uint256 _desiredAmount
   ) private returns (uint256) {
       ...
       if (IUniswapFactory(_factory).getPair(_token, WETH) != address(0)) {
           address[] memory path = new address[](2);
           path[0] = _token;
           path[1] = WETH;
           _dexRouter.swapExactTokensForETHSupportingFeeOnTransferTokens(
               _convertAmount,
               _desiredAmount,
               path,
               msg.sender,
               block.timestamp //@audit deadline
           );
       }
       ...
   }
```

The swapExactTokensForETHSupportingFeeOnTransferTokens in UniswapV2Router02 contract:

```

function swapExactTokensForETHSupportingFeeOnTransferTokens(
        ...
        uint deadline
    )
        external
        virtual
        override
        ensure(deadline)
    {

```

The deadline parameter enforces a time limit by which the transaction must be executed otherwise it will revert.

Let's take a look at the ensure modifier that is present in the functions you are calling in UniswapV2Router02 contract:

```
modifier ensure(uint deadline) {
        require(deadline >= block.timestamp, 'UniswapV2Router: EXPIRED');
        _;
    }
```

```

Now when the deadline is hardcoded as block.timestamp, the transaction will not revert because the require statement will always be fulfilled by block.timestamp == block.timestamp.

If a user chooses a transaction fee that is too low for miners to be interested in including the transaction in a block, the transaction stays pending in the mempool for extended periods, which could be hours, days, weeks, or even longer.

This could lead to users getting a worse price because a validator can just hold onto the transaction.

## Recommendations

Protocols should let users who interact with AMMs set expiration deadlines. Without this, there's a risk of a serious loss of funds for anyone starting a swap.

Use a user-supplied deadline instead of block.timestamp.

# [M-01] Insufficient input validation

Impact:
Medium, because a protocol can be broken and the code could give a false calculations

Likelihood:
Medium, as it can be gamed but it needs compromised / malicious owner

Description

The _rewardMultiplier param in createRiskPool() is not constrained in any way.

Another instances where an upper constrain is missing are:

- the setter functions in CapitalAgent.sol that involve uint256 as a param
- the setter functions in SingleSidedInsurancePool.sol that involve uint256 as a param
- the setter functions in SingleSidedReinsurancePool.sol that involve uint256 as a param
- the param in setMinLPCapital()
- the param in setBuyPolicyMaxDeadline()
- the param in policyClaim()
- the param in setBuyPolicyMaxDeadlineInPolicy()

## Recommendation

Set reasonable lower and upper constrains for these params.
```

[M-02] Unsafe downcasting can lead to errors

Impact: High because important data can be lost

Likelihood: Low because it will happen when very large amounts are used

Description

There are instances where uint256 is downcasted to a much smaller uint128. For example, the uint256 unoReward is then multiplied and downcasted to uint128. Let's take a look at SSIP::pendingUno:

```

  function pendingUno(address _to) external view returns (uint256 pending) {
        uint256 tokenSupply = IERC20(riskPool).totalSupply();
        ...
            uint256 unoReward = blocks * poolInfo.unoMultiplierPerBlock;
            accUnoPerShare = accUnoPerShare + uint128((unoReward *
ACC_UNO_PRECISION) / tokenSupply); //@audit wrong downcasting
        }
        uint256 userBalance = userInfo[_to].amount;
        pending = (userBalance * uint256(accUnoPerShare)) / ACC_UNO_PRECISION
- userInfo[_to].rewardDebt;
    }

```

This is problematic because if the calculation is bigger than uint128, then only the least significant 128 bits will be used. This can lead to loss of data. The downcasting in this example is unnecessary as the accUnoPerShare then is casted to uint256 within the pending calculation. The same applies to the updatePool function where we can see unsafe downcasting again:

```
  function updatePool() public override {
        ...
                uint256 unoReward = blocks * poolInfo.unoMultiplierPerBlock;
                poolInfo.accUnoPerShare = poolInfo.accUnoPerShare +
uint128(((unoReward * ACC_UNO_PRECISION) / tokenSupply));//@audit unsafe
downcasting
            }
```

```
               ...
       }
```

Recommendations

Be consistent with uints or use the SafeCast library to avoid losing any data and undesirable scenarios.

[L-01] Use Ownable2StepUpgradeable instead of OwnableUpgradeable contract

contract: CapitalAgent.sol

transferOwnership function is used to change Ownership from OwnableUpgradeable.sol.

There is another Openzeppelin Ownable contract (Ownable2StepUpgradeable.sol). This helps prevent accidental transfers of ownership and provides an additional layer of security:

Ownable2StepUpgradeable.sol

Also in SalesPolicyFactory.sol we can see the following import:

```
      "@openzeppelin/contracts/access/Ownable.sol"
```

which is better to be changed with Ownable2Step:

Ownable2Step.sol

[L-02] Inconsistent 0 address checks

All address params in the PremiumPool.sol constructor have a check for address 0 except _governance.

```
constructor(address _exchangeAgent, address _unoToken, address _usdcToken,
address _multiSigWallet, address _governance) {
        require(_exchangeAgent != address(0), "UnoRe: zero exchangeAgent address");
        require(_unoToken != address(0), "UnoRe: zero UNO address");
        require(_usdcToken != address(0), "UnoRe: zero USDC address");
        require(_multiSigWallet != address(0), "UnoRe: zero multisigwallet address");

```

Add address 0 check for _governance as well.

[L-03] Missing array length check

In buyPolicy() we have 4 array params which are all compared if they are equal to each other except the _assets array.
Validate that all the arguments have the same length so you do not get unexpected errors if they don't.

```
function buyPolicy(
        address[] memory _assets, //@audit this param is not checked against the others
        address[] memory _protocols,
        uint256[] memory _coverageAmount,
        uint256[] memory _coverageDuration,
        uint256 _policyPriceInUSDC,
        uint256 _signedTime,
        address _premiumCurrency,
        bytes32 r,
        bytes32 s,
        uint8 v
    ) external payable whenNotPaused nonReentrant {
        uint256 len = _protocols.length;
        require(len > 0, "UnoRe: no policy");
        require(len == _coverageAmount.length, "UnoRe: no match protocolIds with coverageAmount");
        require(len == _coverageDuration.length, "UnoRe: no match protocolIds with coverageDuration");

```

[L-04] Mismatch between NatSpec and the actual code

The RiskPoolERC20 contract defines empty constructor with no parameters, which is not necessary and only hinders code readability. According to the Solidity docs on constructors: “If there is no constructor, the contract will assume the default constructor, which is equivalent to constructor() public {}“.

Although there are comments above the empty defined constructor:

```

/**
     * @dev Sets the values for {name} and {symbol}.
     *
     * The default value of {decimals} is 18. To select a different value for
     * {decimals} you should overload it.
     *
     * All two of these values are immutable: they can only be set once
during
     * construction.
     */
    constructor() {}

```

Either define a constructor that the comments describe or delete the constructor declaration and the comments above it.

[L-05] Dangerous role setting

In SingleSidedReinsurancePool.sol we can observe the function setRole which sets a particular role for the given user. The problem here is that every person with a particular role can call this function and give the same role to everybody which can be problematic in some scenarios.

A better idea would be for one person, such as the admin, to have the ability to call that function and set all roles. This would be much more secure.

[L-06] Direct usage of ecrecover allows signature malleability

The getSender function of SalesPolicy.sol calls the Solidity ecrecover function directly to verify the given signatures. However, the ecrecover EVM opcode allows malleable (non-unique) signatures and thus is susceptible to replay attacks.

SWC-117: Signature Malleability

Use the recover function from OpenZeppelin's ECDSA library for signature verification.

[I-01] Wrong event emitted

In killPool() which is in PremiumPool.sol we can observe an event emission of PoolAlived() which is not the proper event for the function.
Create and emit and event that is suitable for this function such as PoolKilled().

The same is present in SingleSidedInsurancePool.sol .

[I-02] Unused code

The following events are not emitted anywhere. Either emit them in the proper functions or delete them:

- event LogForceSetUserRewardDebt()
- event PolicyApproved()
- event PolicyRejected()
- event InsuranceIssued()
- event RoleAccepted()

[I-03] NatSpec docs are incomplete

Some external methods are missing certain components from the NatSpec documentation such as @param and some methods are missing it completely. NatSpec documentation is essential for better understanding of the code by developers and auditors and is strongly recommended. Please refer to the NatSpec format and follow the guidelines outlined there.

[I-04] Typos

- defiend -> defined
- requied -> required
- cancelWithrawRequest() -> cancelWithdrawRequest

[I-05] Redundant code

A lot of functions inside CapitalAgent.sol that don't make any external calls and are protected by the onlyRole(ADMIN_ROLE) modifier also have the nonReentrant modifier which is redundant and just wastes gas. Such functions are:

- setSalesPolicyFactory
- setOperator
- addPoolWhiteList
- removePoolWhiteList
- removePool
- setPolicy
- setPolicyByAdmin
- removePolicy

There are redundant zero address checks in some of the remove functions like removePool and removePoolWhiteList. The check that the pool exists is enough as there is a zero address check in the add functions.

A redundant zero address check in setPolicy - checking that the salesPolicyFactory == msg.sender is enough.

The collectPremiumInETH function inside PremiumPool.sol also has a redundant nonReentrant modifier.

[I-06] Variables can be turned immutable

The UNO_TOKEN and the USDC_TOKEN addresses inside CapitalAgent can be made immutable as they will not be changed and this will save gas.