Smart Contract Security Audit Report

UnoRe

1. Contents

2. General Information

This report contains information about the results of the security audit of the UnoRe (hereafter referred to as "Customer") smart contracts, conducted by Decurity in the period from 18/12/2023 to 29/12/2023.

2.1. Introduction

Tasks solved during the work are:

- Review the protocol design and the usage of 3rd party dependencies,
- Audit the contracts implementation,
- Develop the recommendations and suggestions to improve the security of the contracts.

2.2. Scope of Work

The audit scope included the contracts in the following repositories: https://github.com/Uno-Re/SSIP-SSRP-contracts and https://github.com/Uno-Re/unore-uno-dao. Initial review was done for the commit 2ca5fb5 and f1c9430f respectively and the re-testing was done for the commit 64a989 and 00a364 respectively.

The following contracts have been tested in SSIP-SSRP-contracts:

1. ./contracts/EIP712MetaTransaction.sol
2. ./contracts/factories/SyntheticSSRPFactory.sol
3. ./contracts/factories/SalesPolicyFactory.sol
4. ./contracts/factories/RiskPoolFactory.sol
5. ./contracts/factories/RewarderFactory.sol
6. ./contracts/factories/SyntheticSSIPFactory.sol
7. ./contracts/RiskPoolERC20.sol
8. ./contracts/SingleSidedReinsurancePool.sol
9. ./contracts/libraries/EIP712Base.sol

10. ./contracts/libraries/TransferHelper.sol
11. ./contracts/libraries/MultiSigWallet.sol
12. ./contracts/ExchangeAgent.sol
13. ./contracts/SingleSidedInsurancePool.sol
14. ./contracts/CapitalAgent.sol
15. ./contracts/uma/EscalationManager.sol
16. ./contracts/RiskPool.sol
17. ./contracts/Rewarder.sol
18. ./contracts/SalesPolicy.sol
19. ./contracts/PremiumPool.sol

The following contracts have been tested in unore-uno-dao:

1. ./access/Owned.sol
2. ./libraries/TransferHelper.sol
3. ./SmartWalletChecker.sol
4. ./apps/VeUnoDaoYieldDistributor.sol
5. ./misc/Helpers.sol
6. ./Ownership.sol
7. ./automation/Resolver.sol

2.3. Threat Model

The assessment presumes actions of an intruder who might have capabilities of any role (an external user, token owner, token service owner, a contract).

The main possible threat actors are:

- User,
- Protocol owner,
- Liquidity Token owner/contract.

The table below contains sample attacks that malicious attackers might carry out.

Table. Theoretically possible attacks

Attack	Actor
Contract code or data hijacking *Deploying a malicious contract or submitting malicious data*	Contract owner Token owner
Financial fraud *A malicious manipulation of the business logic and balances, such as a re-entrancy attack or a flash loan attack*	Anyone
Attacks on implementation *Exploiting the weaknesses in the compiler or the runtime of the smart contracts*	Anyone

2.4. Weakness Scoring

An expert evaluation scores the findings in this report, an impact of each vulnerability is calculated based on its ease of exploitation (based on the industry practice and our experience) and severity (for the considered threats).

2.5. Disclaimer

Due to the intrinsic nature of the software and vulnerabilities and the changing threat landscape, it cannot be generally guaranteed that a certain security property of a program holds.

Therefore, this report is provided “as is” and is not a guarantee that the analyzed system does not contain any other security weaknesses or vulnerabilities. Furthermore, this report is not an endorsement of the Customer’s project, nor is it an investment advice.

That being said, Decurity exercises best effort to perform their contractual obligations and follow the industry methodologies to discover as many weaknesses as possible and maximize the audit coverage using the limited resources.

3. Summary

As a result of this work, we have discovered three critical security issues which have been fixed and re-tested in the course of the work.

The other suggestions included fixing the medium, low-risk issues and some best practices (see Security Process Improvement).

The UnoRe team has given the feedback for the suggested changes and explanation for the underlying code.

3.1. Suggestions

The table below contains the discovered issues, their risk level, and their status as of December 29, 2023.

Table. Discovered weaknesses

Issue	Contract	Risk Level	Status
Staking may be abused to receive extra rewards	contracts/apps/VeUnoDaoYieldDistributor.sol	Critical	Fixed
onReward() call will revert	SSIP-SSRP-contracts/contracts/Rewarder.sol	Critical	Fixed
requestPayout() call will revert	SingleSidedInsurancePool.sol	Critical	Fixed
OwnedUpgradeable allows storage collisions	unore-uno-dao/contracts/access/OwnedUpgradeable.sol	Medium	Fixed
SmartWalletChecker does not work as expected	unore-uno-dao/contracts/SmartWalletChecker.sol	Medium	Fixed

Tokens with approval race protection are incompatible with ExchangeAgent	SSIP-SSRP-contracts/contracts/ExchangeAgent.sol	Medium	Fixed
Emergency mode of the MasterChef contracts is not supported	SSIP-SSRP-contracts/contracts/SingleSidedInsurancePool.sol, SSIP-SSRP-contracts/contracts/SingleSidedReinsurancePool.sol	Medium	Fixed
SalesPolicy cannot be paused	SSIP-SSRP-contracts/contracts/factories/SalesPolicyFactory.sol	Medium	Fixed
Policy settled flag is not updated in storage	SSIP-SSRP-contracts/contracts/governance/ClaimProcessor.sol	Low	Fixed
Event will not be emitted	contracts/PremiumPool.sol	Low	Fixed
Lack of policy existence verification	SSIP-SSRP-contracts/contracts/CapitalAgent.sol	Low	Fixed
migratedAmount inconsistency	SSIP-SSRP-contracts/SingleSidedReinsurancePool.sol, SSIP-SSRP-contracts/contracts/SingleSidedInsurancePool.sol	Low	Fixed
Implementation contracts can be initialized by an attacker	SSIP-SSRP-contracts/contracts/CapitalAgent.sol, SSIP-SSRP-contracts/contracts/SingleSidedInsurancePool.sol, SSIP-SSRP-	Low	Acknowledged

	contracts/contracts/SingleSidedReinsurancePool.sol		
Replay attacks are possible with getSender	SSIP-SSRP-contracts/contracts/SalesPolicy.sol	**Low**	Fixed
Requirements for _multiSigWallet are not consistent	SSIP-SSRP-contracts/contracts/CapitalAgent.sol, SSIP-SSRP-contracts/contracts/ExchangeAgent.sol, SSIP-SSRP-contracts/contracts/PremiumPool.sol, SSIP-SSRP-contracts/contracts/factories/SalesPolicyFactory.sol	**Low**	Fixed
Direct usage of ecrecover allows signature malleability	SSIP-SSRP-contracts/contracts/EIP712MetaTransaction.sol, SSIP-SSRP-contracts/contracts/SalesPolicy.sol	**Low**	Fixed
uint128 overflow	SSIP-SSRP-contracts/contracts/RiskPoolERC20.sol	**Info**	Fixed
Typo in CLAIM_ACCESSOR_ROLE	SSIP-SSRP-contracts/contracts/uma/EscalationManager.sol, SSIP-SSRP-contracts/contracts/SingleSidedReinsurancePool.sol	**Info**	Fixed

Duplicate roleLockTime check	SSIP-SSRP-contracts/contracts/SingleSidedReinsurancePool.sol	**Info**	Fixed
hardhat/console.sol should be removed	SSIP-SSRP-contracts/contracts/SingleSidedReinsurancePool.sol	**Info**	Fixed
Non-conformance to Solidity naming conventions	SSIP-SSRP-contracts/contracts/PremiumPool.sol	**Info**	Fixed
Redundant value checks	SSIP-SSRP-contracts/contracts/factories/SalesPolicyFactory.sol	**Info**	Fixed

4. General Recommendations

This section contains general recommendations on how to improve overall security level.

The Findings section contains technical recommendations for each discovered issue.

4.1. Security Process Improvement

The following is a brief long-term action plan to mitigate further weaknesses and bring the product security to a higher level:

- Keep the whitepaper and documentation updated to make it consistent with the implementation and the intended use cases of the system,
- Perform regular audits for all the new contracts and updates,
- Ensure the secure off-chain storage and processing of the credentials (e.g. the privileged private keys),
- Launch a public bug bounty campaign for the contracts.

5. Findings

5.1. Staking may be abused to receive extra rewards

Risk Level: Critical

Status: Fixed in the commit fc44e0.

Contracts:

- contracts/apps/VeUnoDaoYieldDistributor.sol

Description:

`VeUnoDaoYieldDistributor` contract relies at the `VotingEscrow` contract. In order to participate in yield distribution users need to create lock at `VotingEscrow`. The longer period users create their lock for the more voting power they get. Thus, resulting in an increase of `veUNO` balance. So, in case a user has locked tokens for 4 years they will get more rewards from `VeUnoDaoYieldDistributor`, because rewards directly depend on `veUNO` balance.

However, it's possible to abuse lock end time duration via `VotingEscrow`.

Let's assume a user creates a lock for 4 years. Then they call `checkpoint` at `VeUnoDaoYieldDistributor`. They have registered their lock end time and `veUNO` balance. Then an owner calls `notifyRewardAmount` and `yieldDuration` lasts for 1 week. Then the user claims their yield after 1 week.

After that, they call `increase_unlock_time` and pass `_unlock_time` as `1`. Their lock end time will now be equal to `block.timestamp`, because of calculations at line 508 in `VotingEscrow`.

Now the user can call `withdraw` and unlock all of their tokens.

As a result the user has been receiving yield with max balance as if they have locked their tokens for 4 years. However, their tokens were actually locked only for 1 week.

Remediation:

Consider checking that new `_unlock_time` > `_locked.end` in `increase_unlock_time` function in `VotingEscrow` to prevent `veUNO` balance abuse.

PoC:

```
const { ethers, upgrades } = require("hardhat");
const { BigNumber } = ethers;

describe("VeUnoDaoYieldDistributor", function() {
  const YEAR = BigNumber.from(86400 * 365);
  const WEEK = BigNumber.from(86400 * 7);

  const name = "UnoRe Token";
  const symbol = "UnoRe";
  const decimal = 18;

  const MAX_TIME = YEAR.mul("4");
  const two_to_the_256_minus_1 =
BigNumber.from("2").pow(BigNumber.from("256")).sub(BigNumber.from("1"));
  const ten_to_the_40 =
BigNumber.from("100000000000000000000000000000000000000000");

  beforeEach(async function() {
    this.signers = await ethers.getSigners();
    this.creator = this.signers[0];
    this.alice = this.signers[1];
    this.bob = this.signers[2];
    const accounts = [this.creator, this.alice, this.bob];
    this.Ownership = await ethers.getContractFactory("Ownership");
    this.Token = await ethers.getContractFactory("MockUno");
    this.VotingEscrow = await ethers.getContractFactory("VotingEscrow");
    this.VeUnoDaoYieldDistributor = await
ethers.getContractFactory("VeUnoDaoYieldDistributor");
    this.ownership = await this.Ownership.deploy();
    this.token = await this.Token.deploy(name, symbol);
    this.voting_escrow = await this.VotingEscrow.deploy(
      this.token.address, "Voting-escrowed UnoRe", "veUnoRe", "1",
this.ownership.address
    );
    this.veUnoDaoYieldDistributor = await
upgrades.deployProxy(this.VeUnoDaoYieldDistributor, [
      this.token.address,
      this.voting_escrow.address,
      this.creator.address,
      this.creator.address
    ]);
    await this.veUnoDaoYieldDistributor.deployed();
    for (i = 0; i < accounts.length; i++) {
      await this.token.connect(accounts[i]).faucet(ten_to_the_40);
      await
this.token.connect(accounts[i]).approve(this.voting_escrow.address,
two_to_the_256_minus_1);
    }
    this.rewardAmount =
```

```
BigNumber.from(1000000).mul(BigNumber.from(10).pow(18)); // one million UNO
for reward
    this.escrowedAmount =
BigNumber.from(1000).mul(BigNumber.from(10).pow(18));
  });

  it("POC", async function () {
        let aliceUNOBalance = await this.token.balanceOf(this.alice.address);
        console.log('aliceUnoBalance ==>', aliceUNOBalance.toString());
        console.log(MAX_TIME);
        console.log(this.escrowedAmount);
        await
this.voting_escrow.connect(this.alice).create_lock(this.escrowedAmount,
MAX_TIME);
        let veBalance = await
this.voting_escrow["balanceOf(address)"](this.alice.address);
        console.log("Account veToken balance ==>", veBalance.toString());
        await ethers.provider.send("evm_increaseTime", [
            WEEK.mul(1).toNumber(),
        ]);
        await ethers.provider.send("evm_mine");
        await this.veUnoDaoYieldDistributor.connect(this.alice).checkpoint();
        console.log("checkpointed");
        await this.token.approve(this.veUnoDaoYieldDistributor.address,
this.rewardAmount);
        await
this.veUnoDaoYieldDistributor.notifyRewardAmount(this.creator.address,
this.rewardAmount);
        await ethers.provider.send("evm_increaseTime", [
            WEEK.mul(1).toNumber(),
        ]);
        await ethers.provider.send("evm_mine");
        await this.veUnoDaoYieldDistributor.connect(this.alice).getYield();
        aliceUNOBalance = await this.token.balanceOf(this.alice.address);
        console.log('Alice UNO Balance ==>', aliceUNOBalance.toString());
        await this.voting_escrow.connect(this.alice).increase_unlock_time(1);
        await this.voting_escrow.connect(this.alice).withdraw();
        aliceUNOBalance = await this.token.balanceOf(this.alice.address);
        console.log('Alice UNO Balance ==>', aliceUNOBalance.toString());
  })
})
```

References:

- https://github.com/InsureDAO/dao-contracts/blob/develop/contracts/VotingEscrow.sol#L526

5.2. onReward() call will revert

Risk Level: Critical

Status: Fixed in the commit 00b502.

Contracts:

- SSIP-SSRP-contracts/contracts/Rewarder.sol

Location: Lines: 64. Function: onReward.

Description:

onReward function in the Rewarder contract implements the following check:

```
require(tx.origin == _to, "UnoRe: must be message sender");
```

This function is called by rollOverReward function in SingleSidedInsurancePool and SingleSidedReinsurancePool contracts. The call looks the following way:

```
IRewarder(rewarder).onReward(riskPool, _totalPendingUno);
```

riskPool is passed as to argument. However, riskPool != tx.origin , thus resulting in a revert.

Remediation:

Consider refactoring tx.origin check to make calls successful.

5.3. requestPayout() call will revert

Risk Level: Critical

Status: Fixed in the commits 8b163c and 3711e2.

Contracts:

- SingleSidedInsurancePool.sol

Location: Lines: 473-488. Function: requestPayout.

Description:

The SSIP.requestPayout() function calls the OptimisticOracleV3.assertTruth() function.

```
uint256 bond = 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.
);
```

The `assertTruth()` in OptimisticOracleV3 calls `currency.safeTransferFrom(msg.sender, address(this), bond)` on line 196.

In this case `msg.sender` is equal to the address of the SSIP contract. The SSIP contract does not call `defaultCurrency.approve()` to the OptimisticOracleV3 contract address with minimum bond amount.

Users will not be able to collect their insurance payment since the call to the `SSIP.reqeustPayout()` function will always revert.

It should also be noted that amount of `defaultCurrency` may not be enough for calling `oo.assertTruth()`. The `oo.assertTruth()` call in SSIP would work correctly only if funds from PremiumPool contract are sent to the SSIP contract via `PremiumPool.depositToSyntheticSSIPRewarder()`, i.e. if the oracle's collateral (bond) is debited from the funds of the policy buyers. Otherwise, the users will be able to spend the balance of the SSIP contract.

Remediation:

Call `approve()` to the OptimisticOracleV3 address inside the `requestPayout()` function.

References:

- https://github.com/UMAprotocol/protocol/blob/d71f315f671fbf9ee0cad1cbad6ce569a29455b6/packages/core/contracts/optimistic-oracle-v3/implementation/OptimisticOracleV3.sol#L196
- https://docs.uma.xyz/developers/setting-custom-bond-and-liveness-parameters

5.4. OwnedUpgradeable allows storage collisions

Risk Level: Medium

Status: Fixed in the commit 00a364.

Contracts:

- unore-uno-dao/contracts/access/OwnedUpgradeable.sol

Description:

The contract `OwnedUpgradeable` is expected to be used as an upgradeable variant of the `Owned` contract. However its storage layout does not support upgradeability.

Remediation:

To prevent storage collisions implement one of the following solutions:

1. follow ERC7201 (Namespaced Storage Layout)
2. use storage gaps

References:

- https://eips.ethereum.org/EIPS/eip-7201
- https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies#storage-collisions-between-implementation-versions
- https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable#storage-gaps

5.5. SmartWalletChecker does not work as expected

Risk Level: Medium

Status: Fixed in the commit fc44e0.

Contracts:

- unore-uno-dao/contracts/SmartWalletChecker.sol

Location: Lines: 6. Function: check.

Description:

The expectation of the check function that is used inside `assert_no_contract` is to allow access only to EOA accounts. The function gets the size of the code length of the account by relying on `extcodesize` opcode, which can be bypassed when deploying a smart contract through the smart contract's constructor call.

Remediation:

Modify the code to `require(msg.sender == tx.origin);`

References:

- https://solidity-by-example.org/hacks/contract-size/

5.6. Tokens with approval race protection are incompatible with ExchangeAgent

Risk Level: Medium

Status: Fixed in the commit 2da63a.

Contracts:

- SSIP-SSRP-contracts/contracts/ExchangeAgent.sol

Location: Function: `_convertTokenForToken, _convertTokenForETH`.

Description:

Some tokens, for example USDT and KNC have approval race protection mechanism and require the allowance to be either 0 or `uint256.max` when it is updated. The problem is that `ExchangeAgent` uses `safeApprove` from Uniswap's TransferHelper which will revert on such tokens:

```
function _convertTokenForETH(
        address _dexAddress,
        address _token,
        uint256 _convertAmount,
        uint256 _desiredAmount
    ) private returns (uint256) {
        IUniswapRouter02 _dexRouter = IUniswapRouter02(_dexAddress);
        address _factory = _dexRouter.factory();
        uint256 ethBalanceBeforeSwap = address(msg.sender).balance;
        TransferHelper.safeApprove(_token, address(_dexRouter),
_convertAmount);
```

Remediation:

Make sure to use `forceApprove` from `SafeERC20`.

References:

- https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/utils/SafeERC20.sol

5.7. Emergency mode of the MasterChef contracts is not supported

Risk Level: Medium

Status: Fixed in commit e252ba.

Contracts:

- SSIP-SSRP-contracts/contracts/SingleSidedInsurancePool.sol,
- SSIP-SSRP-contracts/contracts/SingleSidedReinsurancePool.sol

Description:

The original MasterChef contract has the emergency withdrawal mode, which allows withdrawals excluding the rewards:

```
// Withdraw without caring about rewards. EMERGENCY ONLY.
function emergencyWithdraw(uint256 _pid) public nonReentrant {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    uint256 amount = user.amount;
    user.amount = 0;
    user.rewardDebt = 0;
    user.rewardLockedUp = 0;
    user.nextHarvestUntil = 0;
    pool.lpToken.safeTransfer(address(msg.sender), amount);
    emit EmergencyWithdraw(msg.sender, _pid, amount);
}
```

In case an attacker takes over the contract, the stakers should have the ability to withdraw the funds themselves.

Remediation:

Add the emergency mode implementation.

5.8. SalesPolicy cannot be paused

Risk Level: Medium

Status: Fixed in the commit 8f58bf.

Contracts:

- SSIP-SSRP-contracts/contracts/factories/SalesPolicyFactory.sol

Description:

The `SalesPolicyFactory` contract doesn't have functions that can call the `killPool()` and `revivePool()` methods, which are used to pause and unpause the contract.

```
contracts/SalesPolicy.sol:
   96:     function killPool() external onlyFactory {
   97:         _pause();
   98:     }
   99:
  100:     function revivePool() external onlyFactory {
  101:         _unpause();
  102:     }
```

Remediation:

To fix this issue functions that can call `killPool()` and `revivePool()` methods should be added to the `SalesPolicyFactory` contract. This will allow for the pausing and unpausing of the contract when required.

5.9. Policy settled flag is not updated in storage

Risk Level: Low

Status: Fixed in the commit 92c4f5.

Contracts:

- SSIP-SSRP-contracts/contracts/governance/ClaimProcessor.sol

Location: Lines: 53. Function: `claimPolicy`.

Description:

The `claimPolicy` function changes the `settled` state of a policy in memory, but this change does not persist in the contract state.

```
contracts/governance/ClaimProcessor.sol:
  50:     function claimPolicy(uint256 _assertionId) external {
  51:         Claim memory _policy = assertion[_assertionId];
  52:         require(_policy.approved && !_policy.settled, "UnoRe: not
approved or already settled");
  53:         _policy.settled = true; // @audit state changed only in memory
  54:
ISingleSidedInsurancePool(_policy.ssip).settlePayout(_policy.policyId,
bytes32(0));
  55:
  56:         emit PolicyClaimed(msg.sender, _assertionId, _policy.ssip);
  57:     }
```

Remediation:

Modify the `claimPolicy` function to ensure that the `settled` state change is updated in the contract state, not just in memory. This can be done by directly modifying the `settled` state of the policy in the contract state.

5.10. Event will not be emitted

Risk Level: Low

Status: Fixed in the commit c21f24.

Contracts:

- contracts/PremiumPool.sol

Description:

The `LogRemoveCurrency` event won't be emitted because it goes after the return statement.

```
contracts/PremiumPool.sol:
  224:     function removeCurrency(address _currency) external
onlyRole(ADMIN_ROLE) {
  225:         require(availableCurrencies[_currency], "Not available yet");
  226:         availableCurrencies[_currency] = false;
  227:         uint256 len = availableCurrencyList.length;
  228:         address lastCurrency = availableCurrencyList[len - 1];
  229:         for (uint256 ii = 0; ii < len; ii++) {
  230:             if (_currency == availableCurrencyList[ii]) {
  231:                 availableCurrencyList[ii] = lastCurrency;
  232:                 availableCurrencyList.pop();
  233:                 destroyCurrencyAllowance(_currency, exchangeAgent);
  234:                 return;
  235:             }
```

```
  236:         }
  237:         emit LogRemoveCurrency(address(this), _currency);
  238:     }
```

Remediation:

Move the `emit LogRemoveCurrency(address(this), _currency);` line before the return statement. This ensures that the event is emitted before the function exits.

5.11. Lack of policy existence verification

Risk Level: Low

Status: Fixed in the commit 1d08a6.

Contracts:

- SSIP-SSRP-contracts/contracts/CapitalAgent.sol

Description:

There is no check that the policy is already set.

```
contracts/CapitalAgent.sol:
  154:     function setPolicy(address _policy) external override nonReentrant
{
  155:         require(salesPolicyFactory != address(0), "UnoRe: not set
factory address yet");
  156:         require(salesPolicyFactory == msg.sender, "UnoRe: only
salesPolicyFactory can call");
  157:         policyInfo = PolicyInfo({policy: _policy, utilizedAmount: 0,
exist: true});
  158:
  159:         emit LogSetPolicy(_policy);
  160:     }
```

Remediation:

Add a condition to verify if a policy is already set (`policyInfo.exist`) before allowing a new one to be established. If there is a pre-existing policy, the function should not proceed and an appropriate error message should be returned.

5.12. migratedAmount inconsistency

Risk Level: Low

Status: Fixed in the commit ebecd7.

Contracts:

- SSIP-SSRP-contracts/SingleSidedReinsurancePool.sol,
- SSIP-SSRP-contracts/contracts/SingleSidedInsurancePool.sol

Location: Function: `migrate`.

Description:

Both `SingleSidedInsurancePool` and `SingleSidedReinsurancePool` contracts have the function `migrate` which allows to migrate LP shares to a new contract. However there are inconsistencies with the `migratedAmount` variable:

1) `SingleSidedReinsurancePool`

```
uint256 amount = userInfo[msg.sender].amount;
uint256 migratedAmount = IRiskPool(riskPool).migrateLP(msg.sender, migrateTo,
isUnLocked);
IMigration(migrateTo).onMigration(msg.sender, amount, "");
emit LogMigrate(msg.sender, migrateTo, migratedAmount);
```

2) `SingleSidedInsurancePool`

```
uint256 migratedAmount = IRiskPool(riskPool).migrateLP(msg.sender, migrateTo,
isUnLocked);
ICapitalAgent(capitalAgent).SSIPPolicyCaim(migratedAmount, 0, false);
IMigration(migrateTo).onMigration(msg.sender, migratedAmount, "");
emit LogMigrate(msg.sender, migrateTo, migratedAmount);
```

Remediation:

Call `onMigration` either with `migratedAmount` received from `riskPool`'s `migrateLP` or with `amount` from `userInfo`.

5.13. Implementation contracts can be initialized by an attacker

Risk Level: Low

Status: Acknowledged: "We only have an `initialize` function and removed `constructor` from upgradable smart contracts. We are using @openzeppelin/hardhat-upgrades module to deploy upgradable smart contracts, and initialize function execute at the same time".

Contracts:

- SSIP-SSRP-contracts/contracts/CapitalAgent.sol,
- SSIP-SSRP-contracts/contracts/SingleSidedInsurancePool.sol,
- SSIP-SSRP-contracts/contracts/SingleSidedReinsurancePool.sol

Description:

The contracts `CapitalAgent`, `SingleSidedInsurancePool` and `SingleSidedReinsurancePool` are assumed to be proxy implementation contracts of the OpenZeppelin's UUPS proxy pattern. Uninitialized implementation contracts can be taken over by an attacker via a direct call to the `initialize` function of the implementation contract. If an implementation contract makes a `delegatecall`, it can be destroyed with a `selfdestruct` from an exploit contract. Although there are no delegatecalls, it is recommended to protect `initialize` from being called directly.

Remediation:

Add constructors and use OZ Initializable's `_disableInitializers()` function as the only line of code in the constructor.

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

References:

- https://medium.com/immunefi/wormhole-uninitialized-proxy-bugfix-review-90250c41a43a

5.14. Replay attacks are possible with getSender

Risk Level: Low

Status: Fixed in the commit 7f321d.

Contracts:

- SSIP-SSRP-contracts/contracts/SalesPolicy.sol

Location: Lines: `308`. Function: `getSender`.

Description:

In the function `getSender` of the `SalesPolicy` contract a signature of the following message is verified:

```
bytes32 msgHash = keccak256(
    abi.encodePacked(_policyPrice, _protocols, _coverageDuration,
_coverageAmount, _signedTime, _premiumCurrency)
);
bytes32 digest = keccak256(abi.encodePacked("\x19Ethereum Signed
Message:\n32", msgHash));
address recoveredAddress = ecrecover(digest, v, r, s);
```

The `msgHash` does not include neither `chain.id` nor any nonce which makes it possible to replay this signature.

Remediation:

Include the value of `chain.id` and a nonce as part of the signed message to prevent signature replay attacks.

References:

- https://mirror.xyz/0xbuidlerdao.eth/lOE5VN-BHI0olGOXe27F0auviIuoSlnou_9t3XRJseY
- https://swcregistry.io/docs/SWC-121/

5.15. Requirements for `_multiSigWallet` are not consistent

Risk Level: Low

Status: Fixed in the commit f67f49.

Contracts:

- SSIP-SSRP-contracts/contracts/CapitalAgent.sol,
- SSIP-SSRP-contracts/contracts/ExchangeAgent.sol,
- SSIP-SSRP-contracts/contracts/PremiumPool.sol,
- SSIP-SSRP-contracts/contracts/factories/SalesPolicyFactory.sol

Location: Function: `initialize`.

Description:

In the contract `SingleSidedInsurancePool` the argument `_multiSigWallet` in the `initialize` function is supposed to be a Gnosis Safe multisig wallet:

```
require(_multiSigWallet != address(0), "UnoRe: zero multisigwallet address");
require(IGnosisSafe(_multiSigWallet).getOwners().length > 3, "UnoRe: more than
three owners requied");
require(IGnosisSafe(_multiSigWallet).getThreshold() > 1, "UnoRe: more than one
owners requied to verify");
```

However in the following contracts there are no such restrictions for `_multiSigWallet`:

- SSIP-SSRP-contracts/contracts/CapitalAgent.sol
- SSIP-SSRP-contracts/contracts/ExchangeAgent.sol
- SSIP-SSRP-contracts/contracts/PremiumPool.sol
- SSIP-SSRP-contracts/contracts/factories/SalesPolicyFactory.sol

Remediation:

The requirements for `_multiSigWallet` should be enforced the same way across all the contracts.

5.16. Direct usage of `ecrecover` allows signature malleability

Risk Level: Low

Status: Fixed in 6ef5bc.

Contracts:

- SSIP-SSRP-contracts/contracts/EIP712MetaTransaction.sol,
- SSIP-SSRP-contracts/contracts/SalesPolicy.sol

Description:

The `verify` function of `EIP712MetaTransaction` and the `getSigner` function of `SalesPolicy` calls the Solidity `ecrecover` function directly to verify the given signature. However, the `ecrecover` EVM opcode allows for malleable (non-unique) signatures and thus is susceptible to replay attacks. Rejecting malleable signatures is considered a best practice.

Remediation:

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

References:

- https://swcregistry.io/docs/SWC-117/
- https://swcregistry.io/docs/SWC-121/

5.17. uint128 overflow

Risk Level: Info

Status: Fixed in the commit fdb449.

Contracts:

- SSIP-SSRP-contracts/contracts/RiskPoolERC20.sol

Description:

The down casting from uint256 to uint128 at line 321 can potentially cause an integer overflow. If the `_amount` added to `pendingAmount` exceeds the maximum value of uint128, it will cause an overflow, leading to incorrect values. The `totalWithdrawPending` value will contain regular `_amount`.

```
contracts/RiskPoolERC20.sol:
  312:     function _withdrawRequest(address _user, uint256 _amount, uint256
_amountInUno) internal {
  313:         require(balanceOf(_user) >= _amount, "UnoRe: balance
overflow");
  314:         if (withdrawRequestPerUser[_user].pendingAmount == 0 &&
withdrawRequestPerUser[_user].requestTime == 0) {
  315:             withdrawRequestPerUser[_user] = UserWithdrawRequestInfo({
  316:                 pendingAmount: uint128(_amount),
  317:                 requestTime: uint128(block.timestamp),
  318:                 pendingUno: _amountInUno
  319:             });
  320:         } else {
  321:             withdrawRequestPerUser[_user].pendingAmount +=
uint128(_amount); // @audit overflow possible
  322:             withdrawRequestPerUser[_user].pendingUno += _amountInUno;
  323:             withdrawRequestPerUser[_user].requestTime =
uint128(block.timestamp);
  324:         }
  325:         totalWithdrawPending += _amount; //@audit because of the
overflow we can add amount more than 1 time
  326:     }
```

Remediation:

Consider checking that `_amount` does not exceed max uint128.

References:

- https://swcregistry.io/docs/SWC-101/

5.18. Typo in CLAIM_ACCESSOR_ROLE

Risk Level: Info

Status: Fixed in the commit 6af553.

Contracts:

- SSIP-SSRP-contracts/contracts/uma/EscalationManager.sol,
- SSIP-SSRP-contracts/contracts/SingleSidedReinsurancePool.sol

Description:

`CLAIM_ACCESSOR_ROLE` has a typo.

Remediation:

Rename to `CLAIM_ASSESSOR_ROLE`.

5.19. Duplicate roleLockTime check

Risk Level: Info

Status: Fixed in the commit c9b7dc.

Contracts:

- SSIP-SSRP-contracts/contracts/SingleSidedReinsurancePool.sol

Location: Lines: 348. Function: `policyClaim`.

Description:

The function `policyClaim` in the contract `SingleSidedReinsurancePool` has a `roleLockTimePassed` modifier which checks that enough time has passed since `msg.sender` got granted `CLAIM_ACCESSOR_ROLE`. However this check is duplicated:

```
function policyClaim(address _to, uint256 _amount) external
onlyRole(CLAIM_ACCESSOR_ROLE) roleLockTimePassed(CLAIM_ACCESSOR_ROLE)
isStartTime isAlive nonReentrant {
    require(block.timestamp >= roleLockTime[CLAIM_ACCESSOR_ROLE][msg.sender],
"UnoRe: lock time not passed");
```

Remediation:

Remove duplicated check.

5.20. hardhat/console.sol should be removed

Risk Level: Info

Status: Fixed in the commit 372d81.

Contracts:

- SSIP-SSRP-contracts/contracts/SingleSidedReinsurancePool.sol

Location: Lines: 20.

Description:

The contract imports the file `hardhat/console.sol` which is used only for development.

Remediation:

Remove it to maintain clean and efficient code.

5.21. Non-conformance to Solidity naming conventions

Risk Level: Info

Status: Fixed in the commit 1ea551.

Contracts:

- SSIP-SSRP-contracts/contracts/PremiumPool.sol

Location: Lines: 27-32.

Description:

According to the Solidity language naming conventions only constants can be uppercase. However in the `PremiumPool` contract there are several variables that look like constants and a constant that is not uppercase:

```
address public constant burnAddress =
0x000000000000000000000000000000000000dEaD; // @audit should be upper case
mapping(address => uint256) public SSRP_PREMIUM; // @audit should NOT be
uppercase
mapping(address => uint256) public SSIP_PREMIUM;
mapping(address => uint256) public BACK_BURN_UNO_PREMIUM;
uint256 public SSRP_PREMIUM_ETH;
uint256 public SSIP_PREMIUM_ETH;
uint256 public BACK_BURN_PREMIUM_ETH;
```

Some of the function names do not have camel-case:

```
function UnpausePool() external onlyRole(ADMIN_ROLE) { // @audit use camel
case
    _unpause();
}
```

Remediation:

Follow the Solidity naming convention.

References:

- https://docs.soliditylang.org/en/v0.8.13/style-guide.html#constants

5.22. Redundant value checks

Risk Level: Info

Status: Fixed in the commit b00476.

Contracts:

- SSIP-SSRP-contracts/contracts/factories/SalesPolicyFactory.sol

Description:

The `SalesPolicyFactory` contract already implements zero address check on `_exchangeAgent`:

```
contracts/SalesPolicy.sol:
  241:     function setExchangeAgent(address _exchangeAgent) external override
onlyFactory {
  242:         require(_exchangeAgent != address(0), "UnoRe: zero address");
  243:         exchangeAgent = _exchangeAgent;
  244:         emit LogSetExchangeAgentInPolicy(_exchangeAgent,
address(this));
  245:     }

contracts/factories/SalesPolicyFactory.sol:
  108:     function setExchangeAgentInPolicy(address _exchangeAgent) external
onlyOwner {
  109:         require(_exchangeAgent != address(0), "UnoRe: zero address");
  110:         ISalesPolicy(salesPolicy).setExchangeAgent(_exchangeAgent);
  111:     }
```

Excessive value checks also present in:

- `setBuyPolicyMaxDeadlineInPolicy()`
- `setPremiumPoolInPolicy()`

- `setSignerInPolicy()`
- `setCapitalAgentInPolicy()`

It's unnecessary to have values checks in both the `SalesPolicy` contract and the `SalesPolicyFactory` contract. This redundancy can be eliminated by removing the checks from the `SalesPolicyFactory` contract.

Remediation:

Modify the `setExchangeAgentInPolicy` function in the `SalesPolicyFactory` contract to remove these checks.

6. Appendix

6.1. About us

The Decurity team consists of experienced hackers who have been doing application security assessments and penetration testing for over a decade.

During the recent years, we've gained expertise in the blockchain field and have conducted numerous audits for both centralized and decentralized projects: exchanges, protocols, and blockchain nodes.

Our efforts have helped to protect hundreds of millions of dollars and make web3 a safer place.