SMART CONTRACT AUDIT REPORT

for

InsurAce Protocol

Prepared By: Yiqun Chen

PeckShield

July 12, 2021

Document Properties

Client	InsurAce
Title	Smart Contract Audit Report
Target	InsurAce
Version	1.0
Author	Xuxian Jiang
Auditors	Shulian Bie, Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 12, 2021	Xuxian Jiang	Final Release
1.0-rc1	July 11, 2021	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the **InsurAce** design document and related smart contract source code, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About InsurAce

`InsurAce` is a leading decentralized insurance protocol, providing reliable, robust and secure insurance services to DeFi users, allowing them to secure their investment funds against various risks. Being the first in the industry to offer cross-chain portfolio-based covers, `InsurAce` enables users to get unbeatable low premium. Users can also get sustainable investment returns through `InsurAce`'s investment portal and gain rewards though the mining program. `InsurAce` has a live product launched on Ethereum in April 2021 and on BSC in June 2021, have built a full-spectrum cross-chain insurance product line, covering protocols, CEX and IDO platform running on Ethereum, Solana, BSC, Heco, Polygon, Fantom and more in the future.

The basic information of InsurAce is as follows:

Table 1.1: Basic Information of InsurAce

Item	Description
Issuer	InsurAce
Website	https://www.InsurAce.io
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 12, 2021

In the following, we show the compressed file `smart-contracts-peckshield-review.zip` and its MD5/SHA checksum values:

- `MD5: b8391013a1156318659e85e9ed47f313`
- `SHA256: 4bc2996b66ae6bba927ad6a79f4bb78fba9f7b611fe9cc2e2ec4b07fb0576bf8`

And here are the checksum values after all fixes for the issues found in the audit have been checked in:

- `MD5: 3d716820badeaaf811348f2ffa8649e2`
- `SHA256: dc766210b8187eb7247130cc42144e8892510c5ca74f2a5be196546d960ae4bf`

1.2 About PeckShield

PeckShield Inc. [13] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [12]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;

- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [11], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

Table 1.3: The Full Audit Checklist

Category	Checklist Items
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `InsurAce` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	🟦
Medium	3	🟦🟦🟦
Low	2	🟦🟦
Informational	4	🟦🟦🟦🟦
Total	10	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 3 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 4 informational recommendations.

Table 2.1: Key InsurAce Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Strengthened Signature in settleMulti-Chain2BSCBridge()	Business Logic	Fixed
PVE-002	High	Lack Of Token Whitelisting for ClaimAssessor::stake()	Business Logic	Fixed
PVE-003	Medium	Proper getClaimFeeAmount() Calculation	Business Logic	Fixed
PVE-004	Low	Improved Validation Of proposeUnstake()	Coding Practices	Fixed
PVE-005	Informational	Gas Optimization in removeStakersPoolDataByIndex()	Coding Practices	Fixed
PVE-006	Informational	Improved Sanity Checks For System Parameters	Coding Practices	Confirmed
PVE-007	Medium	Necessity of Single-Shot Initialization	Init. and Cleanup	Confirmed
PVE-008	Informational	Redundant State And Code Removal	Coding Practices	Fixed
PVE-009	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-010	Informational	Funds Lockup in FeePool And ClaimSettlementPool	Business Logic	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Strengthened Signature in settleMultiChain2BSCBridge()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: BSCBridgePier
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The InsurAce protocol has a built-in bridge functionality that allows for assets to move across various bridges. Using the BSCBridgePier as the example, each bridge implements two key functions bridgeBSC2MultiChain() and settleMultiChain2BSCBridge(). The first one allows to lock the requested assets so that they can cross the bridge to the intended chain while the second one performs the reverse path by releasing the locked funds to the BSC chain.

While examining the asset movement across different bridges, we notice the following _checkSignature () routine needs to be revised. As the name indicates, it validates the signature to ensure it is indeed signed by the authorized singer. However, the way the signature is validated does not take into account the current bridge contract information. As a result, if the bridge needs to be re-deployed to another address, the previous valid signature may be replayed.

```
    function _checkSignature(
        address _tokenFrom,
        address _tokenTo,
        uint256 _amount,
        bytes32 _chainFrom,
        bytes32 _txHash,
        address _toAddress,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) internal view returns (bool) {
        bytes32 msgHash = keccak256(abi.encodePacked(_tokenFrom, _tokenTo, _amount,
            _chainFrom, _txHash, _toAddress));
        bytes memory prefix = "\x19Ethereum Signed Message:\n32";
        bytes32 prefixedHash = keccak256(abi.encodePacked(prefix, msgHash));
        address signer = ecrecover(prefixedHash, v, r, s);
        return settlementSignerFlagMap[signer];
    }
```

Listing 3.1: `BSCBridgePier::_checkSignature()`

Recommendation In order to block possible signature replay attacks, there is a need to take into account the current bridge address for the signature validation.

Status The issue has been fixed by including the current bridge address for the signature generation and validation.

3.2 Lack Of Token Whitelisting for ClaimAssessor::stake()

- ID: PVE-002
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `ClaimAssessor`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The `InsurAce` protocol supports assessors to evaluate each claim and submit their assessments. The various functions that can be exercised by assessors are mainly implemented in the `ClaimAssessor` contract. In the following, we examine one specific function `stake()`.

To elaborate, we show below its full implementation. This `stake()` function is designed to stake the supported `insurTokenAddress` and increase the assessor's voting power (measured in terms of total staked amount). It comes to our attention that the staked amount is denominated at the given `insurTokenAddress`, which unfortunately is not authenticated.

```
    function stake(address insurTokenAddress, uint256 insurAmount) external payable
        nonReentrant {
        require(insurTokenAddress != address(0), "STK: 1");
        require(insurAmount > 0, "STK: 2");

        address payable assessor = _msgSender();
        IClaimReward(claimReward).recalculateAssessor(assessor);

        IERC20Upgradeable(insurTokenAddress).safeTransferFrom(assessor, address(this),
            insurAmount);
        increaseVotes(assessor, insurAmount);
        emit AssessorStakeEvent(assessor, insurAmount);
    }
```

Listing 3.2: `ClaimAssessor::stake()`

Note the `unstake()` counterpart shares the same issue.

Recommendation Whitelist the given `insurTokenAddress` so that only authenticated tokens (e.g., `INSUR`) may be accepted for staking.

Status The issue has been fixed by ensuring only the intended `INSUR` token is used for staking and unstaking.

3.3 Proper getClaimFeeAmount() Calculation

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Claim`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The `InsurAce` protocol allows users to submit claims to recover possible loss. The various claim-related functions are mainly implemented in the `Claim` contract. By design, each claim may be charged for an associated claim fee and a helper routine `getClaimFeeAmount()` is provided to compute the claim fee.

```
    function getClaimFeeAmount(uint256 claimAmount) public view override returns (
        uint256) {
        return IClaimConfig(cfg).getComplainFeeRateX10000().mul(claimAmount).div(10**4);
    }
```

Listing 3.3: `Claim::getClaimFeeAmount()`

To elaborate, we show above the related `getClaimFeeAmount()` routine. It comes to our attention the claim fee rate is retrieved from the configuration using `IClaimConfig(cfg).getComplainFeeRateX10000()`, which should be `IClaimConfig(cfg).getClaimFeeRateX10000()`.

Recommendation Apply the right claim fee rate during the calculation of claim fee.

Status The issue has been fixed by using the right claim fee rate.

3.4 Improved Validation Of proposeUnstake()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `StakingV2Controller`
- Category: Coding Practices [8]
- CWE subcategory: CWE-561 [4]

Description

The `InsurAce` protocol has a core `StakingV2Controller` contract that provides the desired staking feature. In particular, the staked funds have a lifecycle that is transitioned by the following functions: `stakeTokens()`, `proposeUnstake()`, and `withdrawTokens()`. When examining these functions, we notice the `proposeUnstake()` function can be improved.

To elaborate, we show below the full implementation. Within this function, there is an internal variable `proposeUnstakeLP`. This variable has been assigned twice (lines 184 and 186). Note the first assignment (line 184) is not necessary and the restriction on the proposed unstake amount is not properly applied. Specifically, the current staked amount needs to be no less than the proposed unstake amount, i.e., `require(IStakersPoolV2(stakersPoolV2).getStakedAmountPT(_token)>= _amount)`, instead of the current requirement `require(IStakersPoolV2(stakersPoolV2).getStakedAmountPT(_token )!= 0)` (line 185).

```
180     function proposeUnstake(uint256 _amount, address _token) external override
            nonReentrant whenNotPaused onlyAllowedToken(_token) {
181         require(minUnstakeAmtPT[_token] <= _amount && maxUnstakeAmtPT[_token] >= _amount
                , "PU:1");
182         address lpToken = tokenToLPTokenMap[_token];
183         // eth/lpeth = constant = _amount/lpTokenAmount
184         uint256 proposeUnstakeLP = _amount;
185         require(IStakersPoolV2(stakersPoolV2).getStakedAmountPT(_token) != 0, "PU:2");
186         proposeUnstakeLP = _amount.mul(IERC20Upgradeable(lpToken).totalSupply()).div(
                IStakersPoolV2(stakersPoolV2).getStakedAmountPT(_token));
187         require(proposeUnstakeLP != 0, "PU:3");
188         ILPToken(lpToken).proposeToBurn(_msgSender(), proposeUnstakeLP, unstakeLockBlkPT
                [_token]);
189         emit ProposeUnstakeEvent(_msgSender(), lpToken, proposeUnstakeLP);
190     }
```

Listing 3.4: `StakingV2Controller::proposeUnstake()`

Recommendation Remove the redundant assignment to `proposeUnstakeLP` and properly apply the restriction on the proposed unstake amount.

Status The issue has been fixed by applying the above recommendation.

3.5 Gas Optimization in removeStakersPoolDataByIndex()

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `CapitalPool`
- Category: Business Logics [9]
- CWE subcategory: CWE-841 [6]

Description

The `InsurAce` protocol has an essential `CapitalPool` contract that holds funds to cover various claims. For each token in `CapitalPool`, there are associated internal storage states in `stakersTokenData` and `stakersTokenDataMap`. As there may have a number of tokens for inclusion, the contract provides related helper routines.

While reviewing the associated helper routines, we notice the removal of certain element indexed by `index` from the respective array could benefit from known best practice in reducing the gas consumption.

```
    function removeStakersPoolDataByIndex(uint256 _index) external onlyOwner {
        require(stakersTokenData.length > _index, "RSPDBI:1");
        address token = stakersTokenData[_index];
        delete stakersTokenDataMap[token];
        stakersTokenData[_index] = stakersTokenData[stakersTokenData.length - 1];
        stakersTokenData.pop();
    }
```

Listing 3.5: CapitalPool :: removeStakersPoolDataByIndex()

The idea is that we could simply replace the element to be removed with the last element in the array and `pop()` the last element out. This avoids unnecessary gas usage if the given `_index` happens to be the last one in the array (line 203).

Recommendation Replace the element to be removed with the last element and `pop()` the last element out.

Status The issue has been fixed by applying the above recommendation.

3.6 Improved Sanity Checks For System Parameters

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The InsurAce protocol is no exception. Specifically, if we examine the BSCBridgePier contract, it has defined a number of protocol-wide risk parameters, e.g., capOfTokenAmtDuringInterval and swapFeeBSC. In the following, we show an example routine that allows for their changes.

```
    function setupOutTxIntervalAndCap(
        uint256 _outTxInterval,
        uint256 _capOfTokenAmtDuringInterval,
        uint256 _minTokenAmtPerTx
    ) external onlyOwner {
        outTxInterval = _outTxInterval;
        capOfTokenAmtDuringInterval = _capOfTokenAmtDuringInterval;
        minTokenAmtPerTx = _minTokenAmtPerTx;
    }

    function setSwapFeeBSC(uint256 _swapFeeBSC, address payable _feeCollector) external
        onlyOwner {
        require(_feeCollector != address(0), "SSFE:1");
        swapFeeBSC = _swapFeeBSC;
        feeCollector = _feeCollector;
    }

    function setMultiChainAllowedTokenMapTo2From(
        bytes32 _chainType,
        address _tokenFrom,
        address _tokenTo
    ) external onlyOwner {
        multiChainAllowedTokenMapTo2From[_chainType][_tokenTo] = _tokenFrom;
    }
```

Listing 3.6: A number of setters in BSCBridgePier

Our result shows the update logic on the above parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of a large swapFeeBSC parameter will revert every single swap operation.

Note the RewardController contract also defines a number of risk parameters and their setters can also benefit from improved validation as well.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. Also, consider emitting related events for external monitoring and analytics tools.

Status The issue has been confirmed.

3.7 Necessity of Single-Shot Initialization

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Multiple Contracts
- Category: Initialization and Cleanup [10]
- CWE subcategory: CWE-1188 [2]

Description

The InsurAce protocol has a number of contracts and many of them have a setup() function which is used to set up a number of key parameters. Using the RewardController contract as an example, its setup() function is used to configure a number of contract addresses, including smx, cover, claim and stakingController. To facilitate our discussion, we show below the related code snippet.

```
    function setup(
        address _securityMatrixAddress,
        address _coverAddress,
        address _claimAddress,
        address _stakingControllerAddress,
        address _insur
    ) external onlyOwner {
        require(_securityMatrixAddress != address(0), "S:1");
        require(_coverAddress != address(0), "S:2");
        require(_claimAddress != address(0), "S:3");
        require(_stakingControllerAddress != address(0), "S:4");
        require(_insur != address(0), "S:5");
        smx = _securityMatrixAddress;
        cover = _coverAddress;
        claim = _claimAddress;
        stakingController = _stakingControllerAddress;
        insur = _insur;
        vestingDuration = 1;
    }
```

Listing 3.7: RewardController::setup()

Apparently the above logic only ensures the caller is authenticated and allowed by the system. But it does not provide the guarantee that the `setup()` function can be called only once. Considering multiple initializations could cause unexpected errors for the contract's execution, we strongly suggest to make sure `setup()` could only be called once.

The same issue is also applicable to other `setup()` routines in a number of contracts, including `LPToken`, `StakingV2Controller`, `ExchangeRate`, `ETHBridgePier`, `BSCBridgePier`, `HECOBridgePier`, `ETH2MultiChainVault`, `Product`, `RewardController`, `PremiumPool`, `StakersPoolV2`, etc.

Recommendation Consider the need of ensuring that the `setup()` function could only be called once during the entire lifetime.

Status This issue has been confirmed. And the team clarifies the intention to keep it open as of now, to allow flexibility of adding new features, and will surrender the ownership when things get stabilized.

3.8 Redundant State And Code Removal

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [5]

Description

The `InsurAce` protocol makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `SafeMath`, and `Address`, to facilitate its code implementation and organization. For example, the `RewardController` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `ExchangeRate` contract, there is a helper function `getTokenToTokenAmount()`. As the name indicates, it is used to compute the token amount after conversion. However, we notice the calculation can be simplified as the multiplication and division with the constant `Constant.MULTIPLIERX10E18)` are effectively canceled out.

```
    function getTokenToTokenAmount(
        address _fromToken,
        address _toToken,
        uint256 _amount
    ) external view override returns (uint256) {
        /**
```

```
            1. from_token_exchange_rate => the exchange rate (from_token currency / base
                currency)
            2. to_token_exchange_rate => the exchange rate (to_token currency / base
                currency)
            3. basic_exchange_rate = from_token_exchange_rate / to_token_exchange_rate
            4. target_amount = source_amount / source_decimal_multiplier * (
                basic_exchange_rate) * target_decimal_multiplier
            5. target_amount = source_amount * (10**18) / source_decimal_multiplier * (
                basic_exchange_rate) * target_decimal_multiplier / (10**18)
            6. target_amount = source_amount * (10**18) * target_decimal_multiplier * (
                basic_exchange_rate) / source_decimal_multiplier / (10**18)
            7. target_amount = source_amount * (10**18) * target_decimal_multiplier * (
                from_token_exchange_rate / to_token_exchange_rate) /
                source_decimal_multiplier / (10**18)
         */
        return _amount.mul(Constant.MULTIPLIERX10E18).mul(currencyDecimalMultiplierMap[
            _toToken]).mul(currencyExchangeRateMap[_fromToken]).div(
            currencyExchangeRateMap[_toToken]).div(currencyDecimalMultiplierMap[
            _fromToken]).div(Constant.MULTIPLIERX10E18);
    }
```

Listing 3.8: ExchangeRate::getTokenToTokenAmount()

Another similar cancel-out redundancy occurs in the ClaimReward::_getClaimAssessorRewardAmount () function on the use of the mul(10**4) constant.

Recommendation Consider the removal of the redundant state (or code) with a simplified, consistent implementation.

Status This issue has been fixed by removing redundant code and state.

3.9 Trust Issue of Admin Keys

- ID: PVE-009
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Multiple Contracts
- Category: Security Features [7]
- CWE subcategory: CWE-287 [3]

Description

In the InsurAce protocol, there is a dedicated SecurityMatrix contract that manages the access to all privileged functions. And this SecurityMatrix contract has a privileged owner account. This owner account plays a critical role in governing and regulating the protocol-wide operations (e.g., set various parameters and authorize signers for cover purchases). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we show the privileged `owner` account as well as representative privileged opeations.

```
    function withdrawReward(uint256 amount) external override allowedCaller
        whenNotPaused nonReentrant {
        require(ICoverData(data).getTotalInsurTokenRewardAmount() >= amount, "WR: 1");
        ICoverData(data).decreaseTotalInsurTokenRewardAmount(amount);
        IERC20Upgradeable(insur).safeTransfer(_msgSender(), amount);
    }

    event UnlockCoverRewardEvent(address indexed owner, uint256 amount);

    function unlockRewardByController(address _owner, address _to) external override
        allowedCaller whenNotPaused nonReentrant returns (uint256) {
        return _unlockReward(_owner, _to);
    }
```

Listing 3.9: `Cover::withdrawReward()/unlockRewardByController()`

These privileged operations are mediated with a special modifier `allowedCaller`, which allows flexible access control policies to be managed via `SecurityMatrix`. By doing so, the `owner` account can conveniently configure the intended access control via the following `setAllowdCallersPerCallee()` function in `SecurityMatrix`.

```
    function setAllowdCallersPerCallee(address _callee, address[] memory _callers)
        external onlyOwner {
        require(_callers.length != 0, "SACPC:1");
        // check if callee exist
        if (allowedCallersArray[_callee].length == 0) {
            // not exist, so add callee
            allowedCallees.push(_callee);
        } else {
            // if callee exist, then purge data
            for (uint256 i = 0; i < allowedCallersArray[_callee].length; i++) {
                delete allowedCallersMap[_callee][allowedCallersArray[_callee][i]];
            }
            delete allowedCallersArray[_callee];
        }
        // and overwrite
        for (uint256 index = 0; index < _callers.length; index++) {
            allowedCallersArray[_callee].push(_callers[index]);
            allowedCallersMap[_callee][_callers[index]] = 1;
        }
    }
```

Listing 3.10: `SecurityMatrix::setAllowdCallersPerCallee()`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the owner may also be a counter-party risk to the protocol users. It is worrisome if the privileged `owner` account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changes to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed. The InsurAce team will consider moving to DAO governance when the protocol becomes mature and stabilized in the near future.

3.10 Funds Lockup in FeePool And ClaimSettlementPool

- ID: PVE-010
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: FeePool, ClaimSettlementPool
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The InsurAce protocol defines a number of pools, including CapitalPool, FeePool, ClaimSettlementPool , PremiumPool, and StakersPoolV2. While examining these pools, we notice that both FeePool and ClaimSettlementPool can accept tokens, but there are no functions defined to take out the funds from them.

To elaborate, we show below the FeePool contract. It supports the influx of fees in terms of unstkFee, claimFee, and complainFee (via addUnstkFee(), addClaimFee(), and addComplainFee() respectively). However, the fee is currently locked up on the contract. Additional functions need to be defined to properly transfer the funds out for legitimate purposes, including the support of team development and community engagement.

```
contract FeePool is OwnableUpgradeable, PausableUpgradeable, ReentrancyGuardUpgradeable,
     IFeePool {
    using SafeERC20Upgradeable for IERC20Upgradeable;
    using SafeMathUpgradeable for uint256;
    using AddressUpgradeable for address;

    receive() external payable {} // solhint-disable-line no-empty-blocks

    function initializeFeePool() public initializer {
        __Ownable_init();
        __Pausable_init();
        __ReentrancyGuard_init();
    }

    address public securityMatrix;

    // token -> unstake fee
    mapping(address => uint256) public unstkFee;
    mapping(address => uint256) public claimFee;
    mapping(address => uint256) public complainFee;

    modifier allowedCaller() {
        require((SecurityMatrix(securityMatrix).isAllowdCaller(address(this), _msgSender()))  (_msgSender() == owner()), "allowedCaller");
        _;
    }

    function setup(address _securityMatrix) external onlyOwner {
        require(_securityMatrix != address(0), "S:1");
        securityMatrix = _securityMatrix;
    }

    function addUnstkFee(address _token, uint256 _fee) external payable override allowedCaller {
        unstkFee[_token] = unstkFee[_token].add(_fee);
    }

    function addClaimFee(address _token, uint256 _fee) external payable override allowedCaller {
        claimFee[_token] = claimFee[_token].add(_fee);
    }

    function addComplainFee(address _token, uint256 _fee) external payable override allowedCaller {
        complainFee[_token] = complainFee[_token].add(_fee);
    }
}
```

Listing 3.11: The `FeePool` contract

Recommendation Avoid funds to be locked up in `FeePool` and `ClaimSettlementPool`.

Status This issue has been confirmed. In particular, the fee and claim settlement pools are just placeholders and there is a need to review and revise the token economics in the near future.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `InsurAce` protocol. The system presents a unique, robust offering as a leading non-custodial, multi-chain decentralized insurance protocol, providing reliable, robust and secure insurance services to DeFi users and allowing them to secure their investment funds against various risks. The current code base is well structured and neatly organized, with considerable security measures implemented. Those identified issues are promptly confirmed and fixed.

Moreover, we need to emphasize that `Solidity-based` smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-1188: Insecure Default Initialization of Resource. https://cwe.mitre.org/data/definitions/1188.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[4] MITRE. CWE-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

[5] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[6] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[7] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[8] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[9] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[10] MITRE. CWE CATEGORY: Initialization and Cleanup Errors. https://cwe.mitre.org/data/definitions/452.html.

[11] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[12] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[13] PeckShield. PeckShield Inc. https://www.peckshield.com.