SMART CONTRACT AUDIT REPORT

for

InsurAce Protocol

Prepared By: Yiqun Chen

PeckShield
July 12, 2021
Document Properties

<table>
<thead>
<tr>
<th>Client</th>
<th>InsurAce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Smart Contract Audit Report</td>
</tr>
<tr>
<td>Target</td>
<td>InsurAce</td>
</tr>
<tr>
<td>Version</td>
<td>1.0</td>
</tr>
<tr>
<td>Author</td>
<td>Xuxian Jiang</td>
</tr>
<tr>
<td>Auditors</td>
<td>Shulian Bie, Jing Wang, Xuxian Jiang</td>
</tr>
<tr>
<td>Reviewed by</td>
<td>Yiqun Chen</td>
</tr>
<tr>
<td>Approved by</td>
<td>Xuxian Jiang</td>
</tr>
<tr>
<td>Classification</td>
<td>Public</td>
</tr>
</tbody>
</table>

Version Info

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>July 12, 2021</td>
<td>Xuxian Jiang</td>
<td>Final Release</td>
</tr>
<tr>
<td>1.0-rc1</td>
<td>July 11, 2021</td>
<td>Xuxian Jiang</td>
<td>Release Candidate #1</td>
</tr>
</tbody>
</table>

Contact

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Yiqun Chen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phone</td>
<td>+86 183 5897 7782</td>
</tr>
<tr>
<td>Email</td>
<td><a href="mailto:contact@peckshield.com">contact@peckshield.com</a></td>
</tr>
</tbody>
</table>
### Contents

1 **Introduction**  
   1.1 About InsurAce .......................... 4  
   1.2 About PeckShield .......................... 5  
   1.3 Methodology .............................. 5  
   1.4 Disclaimer ............................... 9  

2 **Findings**  
   2.1 Summary .................................. 10  
   2.2 Key Findings .............................. 11  

3 **Detailed Results**  
   3.1 Strengthened Signature in settleMultiChain2BSCBridge() ................. 12  
   3.2 Lack Of Token Whitelisting for ClaimAssessor::stake() .................. 13  
   3.3 Proper getClaimFeeAmount() Calculation .................................. 14  
   3.4 Improved Validation Of proposeUnstake() .................................. 15  
   3.5 Gas Optimization in removeStakersPoolDataByIndex() ....................... 16  
   3.6 Improved Sanity Checks For System Parameters ............................ 17  
   3.7 Necessity of Single-Shot Initialization .................................. 18  
   3.8 Redundant State And Code Removal ......................................... 19  
   3.9 Trust Issue of Admin Keys .................................................. 20  
   3.10 Funds Lockup in FeePool And ClaimSettlementPool ......................... 22  

4 **Conclusion**  

References  

PeckShield Audit Report #: 2021-193
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 through 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>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issuer</td>
<td>InsurAce</td>
</tr>
<tr>
<td>Website</td>
<td><a href="https://www.InsurAce.io">https://www.InsurAce.io</a></td>
</tr>
<tr>
<td>Type</td>
<td>Ethereum Smart Contract</td>
</tr>
<tr>
<td>Platform</td>
<td>Solidity</td>
</tr>
<tr>
<td>Audit Method</td>
<td>Whitebox</td>
</tr>
<tr>
<td>Latest Audit Report</td>
<td>July 12, 2021</td>
</tr>
</tbody>
</table>
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](https://t.me/peckshield)), Twitter ([http://twitter.com/peckshield](http://twitter.com/peckshield)), or Email ([contact@peckshield.com](mailto:contact@peckshield.com)).

![Vulnerability Severity Classification](image)

**Table 1.2: Vulnerability Severity Classification**

<table>
<thead>
<tr>
<th>Impact</th>
<th>High</th>
<th>Medium</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical</td>
<td>High</td>
<td>Medium</td>
<td>Low</td>
</tr>
<tr>
<td>High</td>
<td>Medium</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Medium</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
</tr>
</tbody>
</table>

### 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

<table>
<thead>
<tr>
<th>Category</th>
<th>Checklist Items</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic Coding Bugs</strong></td>
<td>Constructor Mismatch</td>
</tr>
<tr>
<td></td>
<td>Ownership Takeover</td>
</tr>
<tr>
<td></td>
<td>Redundant Fallback Function</td>
</tr>
<tr>
<td></td>
<td>Overflows &amp; Underflows</td>
</tr>
<tr>
<td></td>
<td>Reentrancy</td>
</tr>
<tr>
<td></td>
<td>Money-Giving Bug</td>
</tr>
<tr>
<td></td>
<td>Blackhole</td>
</tr>
<tr>
<td></td>
<td>Unauthorized Self-Destruct</td>
</tr>
<tr>
<td></td>
<td>Revert DoS</td>
</tr>
<tr>
<td></td>
<td>Unchecked External Call</td>
</tr>
<tr>
<td></td>
<td>Gasless Send</td>
</tr>
<tr>
<td></td>
<td>Send Instead Of Transfer</td>
</tr>
<tr>
<td></td>
<td>Costly Loop</td>
</tr>
<tr>
<td></td>
<td>(Unsafe) Use Of Untrusted Libraries</td>
</tr>
<tr>
<td></td>
<td>(Unsafe) Use Of Predictable Variables</td>
</tr>
<tr>
<td></td>
<td>Transaction Ordering Dependence</td>
</tr>
<tr>
<td></td>
<td>Deprecated Uses</td>
</tr>
<tr>
<td><strong>Semantic Consistency Checks</strong></td>
<td>Semantic Consistency Checks</td>
</tr>
<tr>
<td><strong>Advanced DeFi Scrutiny</strong></td>
<td>Business Logs Review</td>
</tr>
<tr>
<td></td>
<td>Functionality Checks</td>
</tr>
<tr>
<td></td>
<td>Authentication Management</td>
</tr>
<tr>
<td></td>
<td>Access Control &amp; Authorization</td>
</tr>
<tr>
<td></td>
<td>Oracle Security</td>
</tr>
<tr>
<td></td>
<td>Digital Asset Escrow</td>
</tr>
<tr>
<td></td>
<td>Kill-Switch Mechanism</td>
</tr>
<tr>
<td></td>
<td>Operation Trails &amp; Event Generation</td>
</tr>
<tr>
<td></td>
<td>ERC20 Idiosyncrasies Handling</td>
</tr>
<tr>
<td></td>
<td>Frontend-Contract Integration</td>
</tr>
<tr>
<td></td>
<td>Deployment Consistency</td>
</tr>
<tr>
<td></td>
<td>Holistic Risk Management</td>
</tr>
<tr>
<td><strong>Additional Recommendations</strong></td>
<td>Avoiding Use of Variadic Byte Array</td>
</tr>
<tr>
<td></td>
<td>Using Fixed Compiler Version</td>
</tr>
<tr>
<td></td>
<td>Making Visibility Level Explicit</td>
</tr>
<tr>
<td></td>
<td>Making Type Inference Explicit</td>
</tr>
<tr>
<td></td>
<td>Adhering To Function Declaration Strictly</td>
</tr>
<tr>
<td></td>
<td>Following Other Best Practices</td>
</tr>
<tr>
<td>Category</td>
<td>Summary</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Configuration</td>
<td>Weaknesses in this category are typically introduced during the configuration of the software.</td>
</tr>
<tr>
<td>Data Processing Issues</td>
<td>Weaknesses in this category are typically found in functionality that processes data.</td>
</tr>
<tr>
<td>Numeric Errors</td>
<td>Weaknesses in this category are related to improper calculation or conversion of numbers.</td>
</tr>
<tr>
<td>Security Features</td>
<td>Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)</td>
</tr>
<tr>
<td>Time and State</td>
<td>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.</td>
</tr>
<tr>
<td>Error Conditions, Return Values,</td>
<td>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.</td>
</tr>
<tr>
<td>Status Codes</td>
<td></td>
</tr>
<tr>
<td>Resource Management</td>
<td>Weaknesses in this category are related to improper management of system resources.</td>
</tr>
<tr>
<td>Behavioral Issues</td>
<td>Weaknesses in this category are related to unexpected behaviors from code that an application uses.</td>
</tr>
<tr>
<td>Business Logic</td>
<td>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.</td>
</tr>
<tr>
<td>Initialization and Cleanup</td>
<td>Weaknesses in this category occur in behaviors that are used for initialization and breakdown.</td>
</tr>
<tr>
<td>Arguments and Parameters</td>
<td>Weaknesses in this category are related to improper use of arguments or parameters within function calls.</td>
</tr>
<tr>
<td>Expression Issues</td>
<td>Weaknesses in this category are related to incorrectly written expressions within code.</td>
</tr>
<tr>
<td>Coding Practices</td>
<td>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.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Severity</th>
<th># of Findings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical</td>
<td>0</td>
</tr>
<tr>
<td>High</td>
<td>1</td>
</tr>
<tr>
<td>Medium</td>
<td>3</td>
</tr>
<tr>
<td>Low</td>
<td>2</td>
</tr>
<tr>
<td>Informational</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td>10</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>ID</th>
<th>Severity</th>
<th>Title</th>
<th>Category</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>PVE-001</td>
<td>Low</td>
<td>Strengthened Signature in settleMultiChain2BSCBridge()</td>
<td>Business Logic</td>
<td>Fixed</td>
</tr>
<tr>
<td>PVE-002</td>
<td>High</td>
<td>Lack Of Token Whitelisting for ClaimAssessor::stake()</td>
<td>Business Logic</td>
<td>Fixed</td>
</tr>
<tr>
<td>PVE-003</td>
<td>Medium</td>
<td>Proper getClaimFeeAmount() Calculation</td>
<td>Business Logic</td>
<td>Fixed</td>
</tr>
<tr>
<td>PVE-004</td>
<td>Low</td>
<td>Improved Validation Of proposeUnstake()</td>
<td>Coding Practices</td>
<td>Fixed</td>
</tr>
<tr>
<td>PVE-005</td>
<td>Informational</td>
<td>Gas Optimization in removeStakersPoolDataByIndex()</td>
<td>Coding Practices</td>
<td>Fixed</td>
</tr>
<tr>
<td>PVE-006</td>
<td>Informational</td>
<td>Improved Sanity Checks For System Parameters</td>
<td>Coding Practices</td>
<td>Confirmed</td>
</tr>
<tr>
<td>PVE-007</td>
<td>Medium</td>
<td>Necessity of Single-Shot Initialization</td>
<td>Init. and Cleanup</td>
<td>Confirmed</td>
</tr>
<tr>
<td>PVE-008</td>
<td>Informational</td>
<td>Redundant State And Code Removal</td>
<td>Coding Practices</td>
<td>Fixed</td>
</tr>
<tr>
<td>PVE-009</td>
<td>Medium</td>
<td>Trust Issue of Admin Keys</td>
<td>Security Features</td>
<td>Confirmed</td>
</tr>
<tr>
<td>PVE-010</td>
<td>Informational</td>
<td>Funds Lockup in FeePool And ClaimSettlementPool</td>
<td>Business Logic</td>
<td>Confirmed</td>
</tr>
</tbody>
</table>

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 signer. 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.

```solidity
function _checkSignature(  
    address _tokenFrom,  
    address _tokenTo,  
    uint256 _amount,  
    bytes32 _chainFrom,  
    bytes32 _txHash,  
    address _toAddress,  
    uint8 v,  
    bytes32 r,  
    bytes32 s  
) internal view returns (bool) {
```
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.

```solidity
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);
}
```
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.

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]

**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).

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

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.

```solidity
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.

```solidity
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.

```solidity
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 \texttt{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 \texttt{setup()} could only be called once.

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

**Recommendation**  Consider the need of ensuring that the \texttt{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 \texttt{InsurAce} protocol makes good use of a number of reference contracts, such as \texttt{ERC20}, \texttt{SafeERC20}, \texttt{SafeMath}, and \texttt{Address}, to facilitate its code implementation and organization. For example, the \texttt{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 \texttt{ExchangeRate} contract, there is a helper function \texttt{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 \texttt{Constant.MULTIPLIERX10E18} are effectively canceled out.

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

---

19/26  PeckShield Audit Report #: 2021-193
1. \texttt{from\_token\_exchange\_rate} => the exchange rate (from\_token currency / base currency)
2. \texttt{to\_token\_exchange\_rate} => the exchange rate (to\_token currency / base currency)
3. \texttt{basic\_exchange\_rate} = \texttt{from\_token\_exchange\_rate} / \texttt{to\_token\_exchange\_rate}
4. \texttt{target\_amount} = \texttt{source\_amount} / \texttt{source\_decimal\_multiplier} * ( \texttt{basic\_exchange\_rate} ) * \texttt{target\_decimal\_multiplier}
5. \texttt{target\_amount} = \texttt{source\_amount} * (10^{18}) / \texttt{source\_decimal\_multiplier} * ( \texttt{basic\_exchange\_rate} ) * \texttt{target\_decimal\_multiplier} / (10^{18})
6. \texttt{target\_amount} = \texttt{source\_amount} * (10^{18}) * \texttt{target\_decimal\_multiplier} * ( \texttt{basic\_exchange\_rate} ) / \texttt{source\_decimal\_multiplier} / (10^{18})
7. \texttt{target\_amount} = \texttt{source\_amount} * (10^{18}) * \texttt{target\_decimal\_multiplier} * ( \texttt{from\_token\_exchange\_rate} / \texttt{to\_token\_exchange\_rate} ) / \texttt{source\_decimal\_multiplier} / (10^{18})

\* /

\texttt{return \_amount.mul(\texttt{Constant.\_MULTIPLIERX10E18}).mul(currency\_Decimal\_Multiplier\_Map[\_to\_Token]).mul(currency\_Exchange\_Rate\_Map[\_from\_Token]).div(currency\_Exchange\_Rate\_Map[\_to\_Token]).div(currency\_Decimal\_Multiplier\_Map[\_from\_Token]).div(\texttt{Constant.\_MULTIPLIERX10E18});}

Listing 3.8: \texttt{ExchangeRate::getTokenToTokenAmount()}

Another similar cancel-out redundancy occurs in the \texttt{ClaimReward::getClaimAssessorRewardAmount()} function on the use of the \texttt{mul(10^{4})} constant.

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

\textbf{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]

\textbf{Description}

In the \texttt{InsurAce} protocol, there is a dedicated \texttt{SecurityMatrix} contract that manages the access to all privileged functions. And this \texttt{SecurityMatrix} contract has a privileged \texttt{owner} account. This \texttt{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 \texttt{owner} account as well as representative privileged operations.

```solidity
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: \texttt{Cover::withdrawReward()}/\texttt{unlockRewardByController()}

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

```solidity
function setAllowedCallersPerCallee(address _callee, address[] memory _callers) external onlyOwner {
    require(_callers.length != 0, "SACPC:1");
    if (allowedCallersArray[_callee].length == 0) {
        allowedCallees.push(_callee);
    } else {
        for (uint256 i = 0; i < allowedCallersArray[_callee].length; i++) {
            delete allowedCallersMap[_callee][allowedCallersArray[_callee][i]];
        }
        delete allowedCallersArray[_callee];
    }
    for (uint256 index = 0; index < _callers.length; index++) {
        allowedCallersArray[_callee].push(_callers[index]);
        allowedCallersMap[_callee][_callers[index]] = 1;
    }
}
```

Listing 3.10: \texttt{SecurityMatrix::setAllowedCallersPerCallee()}

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 \texttt{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 in-
tended 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() respect-
ively). 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.

```solidity
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();
```

---

**Contract Source Code**
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


