# SOLIDIFIED

Audit Report for InsureDAO - August 13, 2021

## Summary

Audit Report prepared by Solidified covering the InsureDAO smart contracts.

## Process and Delivery

Three (3) independent Solidified experts performed an unbiased and isolated audit of the code below. The final debrief took place on August 10, 2021, and the results are presented here.

## Audited Files

The source code has been supplied in a private source code repository:

https://github.com/insureDAO/pool-contracts (branch: `master`)

Commit number: `13000d2bb8ad5b46625e4351616aa37c0bb61881`

## Intended Behavior

InsureDAO is the composable and open insurance protocol in the Ethereum ecosystem which enables anyone to create, provide, and get insured from potential risks in crypto.

## Findings

Smart contract audits are an important step to improve the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of a smart contract system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**.

Note, that high complexity or lower test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than a security audit and vice versa.

| Criteria | Status | Comment |
|---|---|---|
| Code complexity | Medium | - |
| Code readability and clarity | High | - |
| Level of Documentation | High | - |
| Test Coverage | High | - |

## Issues Found

Solidified found that the InsureDAO contracts contain 1 critical issue, 2 major issues, 4 minor issues, 3 informational notes and 1 warning.

We recommend issues are amended, while informational notes are up to the team's discretion, as they refer to best practices.

| Issue # | Description | Severity | Status |
|---|---|---|---|
| 1 | CDS.sol: Function initialize() can be called after contract initialization | Critical | Pending |
| 2 | PoolTemplate.sol: The function _divFloor() implementation is incorrect | Major | Pending |
| 3 | Vault.sol: An attacker can compromise attribution calculations by directly depositing tokens to the vault | Major | Pending |
| 4 | IndexTemplate.sol: Function setLeverage() does not adjust credit allocation | Minor | Pending |
| 5 | IndexTemplate.sol: Providers can potentially be locked out of withdrawing their collateral from the pool | Minor | Pending |
| 6 | PoolTemplate.sol: Users can allocate credit while the pool is paused | Minor | Pending |
| 7 | IndexTemplate.sol: The owner can add any contract to pools by calling the set() function | Minor | Pending |
| 8 | Duplicate ownership logic | Note | - |
| 9 | MinterRole.sol: Any minter can add new minters. Minters cannot be removed - they can only renounce the role themselves | Note | - |
| 10 | Misc Notes | Note | - |
| 11 | There are no means to deal with an insurance | Warning | - |

| | | | |
|---|---|---|---|
| | fraud - InsureDAO would provide extra financial incentives to exploit the insured contracts | | |

## Critical Issues

### 1. CDS.sol: Function initialize() can be called after contract initialization

The `initialize()` function can be called multiple times after initial contract initialization, thus compromising the immutability of all initialized variables.

**Recommendation**
Require that `initialized == false`.

**Note**
The same issue exists in `IndexTemplate` and `PoolTemplate`.

## Major Issues

### 2. PoolTemplate.sol: The function _divFloor() implementation is incorrect

The function `_divFloor()` incorrectly subtracts `1` from `c`. Also in case `a < b`, the `c-1` will result in an arithmetic underflow.
Subsequently, the function `withdraw()` function is incorrectly calculating `attribution` to be sent to the user.

**Recommendation**
Consider replacing the `_divFloor()` function with the Solidity division operator.

### 3. `Vault.sol`: An attacker can compromise attribution calculations by directly depositing tokens to the vault

Function `addValue()` calculates the beneficiary's attribution using the formula `_attributions = _amount.mul(totalAttributions).div(_pool)`. If an attacker directly sends enough tokens to the vault to have `_pool` be greater than `_amount.mul(totalAttributions)`, this calculation will be equal to zero and thus compromise the attribution calculation.

**Recommendation**

Keep the actual deposited tokens amount in a state variable instead of calling `token.balanceOf()`.

## Minor Issues

### 4. `IndexTemplate.sol`: Function `setLeverage()` does not adjust credit allocation

Function `setLeverage()` does not call `adjustAlloc()` after setting the new leverage, thus leaving the credit allocation out of sync.

**Recommendation**

Call function `adjustAlloc()` from within the aforementioned function.

### 5. `IndexTemplate.sol`: Providers can potentially be locked out of withdrawing their collateral from the pool

Providers currently call the function `withdraw()` to withdraw their collateral from the pool, which in turn iterates over the `poolList` array. If this array were to eventually grow to a sufficiently large size, this will end up exceeding the current block gas limit, and function `withdraw()` will always fail.

**Recommendation**
Place a hard cap on the size of the `poolList` array that will not exceed the current block gas limit.

**Note**
Functions `IndexTemplate.withdrawable()`, `IndexTemplate.adjustAlloc()`, `IndexTemplate.compensate()`, `PoolTemplate.redeem()`, and `PoolTemplate.applyCover()` can all potentially also suffer from the same vulnerability.

## 6. PoolTemplate.sol: Users can allocate credit while the pool is paused

Function `allocateCredit()` allows users to allocate credit even when the pool is `paused`.

**Recommendation**
Restrict this function while the contract is `paused`, as is currently the case with `deposit()`.

## 7. IndexTemplate.sol: The owner can add any contract to pools by calling the set() function

The function `set()` does not check if the `_pool` parameter is a whitelisted `market` contract in the `Registry.sol`.
Effectively, the `owner` of the `IndexTemplate.sol` can drain funds from the contract..

**Recommendation**
Consider adding a check that the supplied `_pool` parameter is a whitelisted `market` contract in the `Registry.sol`

## Informational Notes

### 8. Duplicate ownership logic

The same ownership logic is duplicated across the following contracts: `Factory`, `FeeModel`, `Parameters`, `Registry`, `Vault`, `BondingPremium`, and `PremiumModel`.

**Recommendation**
Consider creating a base class that includes all ownership logic in order to avoid duplicate code.

### 9. `MinterRole.sol`: Any minter can add new minters. Minters cannot be removed - they can only renounce the role themselves

Care must be taken to protect `minter` accounts - if any of them becomes malicious/compromised the `MinterRole.sol` contract does not provide any means to remove the malicious `minter(s)`.

**Recommendation**
Consider providing a mechanism for removing malicious minters.

### 10. Misc Notes

- Vault.sol: Gas can be saved in function `addValue()` if `_pool` is only set when `totalAttributions > 0`.
- Vault.sol: Consider calling `withdrawAttribution()` from within `withdrawAllAttribution()` to avoid code duplication.
- Vault.sol: Inconsistent usage of `SafeERC20` library - some functions use `transfer()` and some use `safeTransfer()`.
- The `CDS.sol`, `IndexTemplate.sol` and `PoolTemplate.sol` contracts implement identical `ERC20` functionality. Consider removing duplicate code.

- Consider removing unused contracts from the codebase.

## Warnings

### 10. There are no means to deal with an insurance fraud - InsureDAO would provide extra financial incentives to exploit the insured contracts

A malicious actor who is going to exploit an insured contract can maximise his “profits” by buying insurance from InsureDAO before exploiting the contract. He might also short (borrow) the Pool/Index or CDS ownership tokens before exploiting the insured contract.

## Disclaimer