InsureDAO contest

2022-03-15

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 code contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the code contest outlined in this document, C4 conducted an analysis of the InsureDAO smart contract system written in Solidity. The code contest took place between January 7—13, 2022.

Wardens

40 Wardens contributed reports to the InsureDAO contest:

1. WatchPug (jtp and ming)
2. leastwood
3. cmichel
4. danb
5. sirhashalot
6. Dravee
7. p4st13r4 (0xb4bb4 and 0x69e8)
8. pauliax
9. camden
10. robee
11. hyh
12. defsec
13. Jujic
14. 0x1f8b
15. gzeon
16. 0xngndev
17. loop

18. ye0lde
19. Ruhum
20. TomFrenchBlockchain
21. Fitraldys
22. cccz
23. hubble (ksk2345 and shri4net)
24. Tomio
25. tqts
26. pmerkleplant
27. ospwner
28. egjlmn1
29. Meta0xNull
30. OriDabush
31. 0x0x0x
32. saian
33. solgryn
34. csanuragjain
35. Kumpirmafyas
36. jayjonah8
37. bugwriter001

This contest was judged by 0xean (judge).

Final report assembled by CloudEllie and itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 65 unique vulnerabilities and 178 total findings. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 14 received a risk rating in the category of HIGH severity, 9 received a risk rating in the category of MEDIUM severity, and 42 received a risk rating in the category of LOW severity.

C4 analysis also identified 15 non-critical recommendations and 98 gas optimizations.

Scope

The code under review can be found within the C4 InsureDAO contest repository, and is composed of 10 smart contracts written in the Solidity programming language and includes 2445 source lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (14)

[H-01] Tokens can be burned with no access control

Submitted by sirhashalot

The Vault.sol contract has two address state variables, the `keeper` variable and the `controller` variable, which are both permitted to be the zero address. If both variables are zero simultaneously, any address can burn the available funds (available funds = balance - totalDebt) by sending these tokens to the zero address with the unprotected `utilitize()` function. If a user has no totalDebt, the user can lose their entire underlying token balance because of this.

Proof of Concept The problematic `utilize()` function is found here. To see how the two preconditions can occur:

1. The keeper state variable is only changed by the `setKeeper()` function found here. If this function is not called, the keeper variable will retain the default value of address(0), which bypasses the only access control for the utilize function.
2. There is a comment here on line 69 stating the controller state variable can be zero. There is no zero address check for the controller state variable in the Vault constructor.

If both address variables are left at their defaults of `address(0)`, then the `safeTransfer()` call on line 348 would send the tokens to address(0).

Recommended Mitigation Steps Add the following line to the very beginning of the `utilize()` function: `require(address(controller) != address(0))`

This check is already found in many other functions in Vault.sol, including the `_unutilize()` function.

oishun1112 (Insure) confirmed and resolved: > https://github.com/InsureDAO/pool-contracts/blob/audit/code4rena/contracts/Vault.sol#L382

[H-02] Typo in PoolTemplate unlock function results in user being able to unlock multiple times

Submitted by loop, also found by p4st13r4 and ye0lde

The function `unlock()` in PoolTemplate has a typo where it compares `insurances[_id].status` to `false` rather than setting it to `false`. If the conditions are met to unlock the funds for an id, the user should be able to call the `unlock()` function once for that id as `insurances[_id].amount` is subtracted from `lockedAmount`. However, since `insurances[_id].status` does not get set to `false`, a user can call `unlock()` multiple times for the same id, resulting in `lockedAmount` being way smaller than it should be since `insurances[_id].amount` is subtracted multiple times.

Impact `lockedAmount` is used to calculate the amount of underlying tokens available for withdrawals. If `lockedAmount` is lower than it should be users are able to withdraw more underlying tokens than available for withdrawals.

Proof of Concept Typo in `unlock()`:

- https://github.com/code-423n4/2022-01-insure/blob/main/contracts/PoolTemplate.sol#L360-L362

Calculation of underlying tokens available for withdrawal:

- https://github.com/code-423n4/2022-01-insure/blob/main/contracts/PoolTemplate.sol#L836

Recommended Mitigation Steps Change `insurances[_id].status == false;` to `insurances[_id].status = false;`

oishun1112 (Insure) confirmed and resolved: > https://github.com/InsureDAO/pool-contracts/blob/audit/code4rena/contracts/PoolTemplate.sol#L375

0xean (judge) commented: > upgrading to sev-3 based on assets being compromised.

[H-03] Malicious Market Creators Can Steal Tokens From Unsuspecting Approved Reference Accounts

Submitted by leastwood

The current method of market creation involves calling `Factory.createMarket()` with a list of approved `_conditions` and `_references` accounts. If a registered template address has `templates[address(_template)].isOpen == true`, then any user is able to call `createMarket()` using this template. If the template points to `PoolTemplate.sol`, then a malicious market creator can abuse `PoolTemplate.initialize()` as it makes a vault deposit from an account that they control. The vulnerable internal function, `_depositFrom()`, makes a vault deposit from the `_references[4]` address (arbitrarily set to an approved reference address upon market creation).

Hence, if approved `_references` accounts have set an unlimited approval amount for `Vault.sol` before deploying their market, a malicious user can frontrun market creation and cause these tokens to be transferred to the incorrect market.

This issue can cause honest market creators to have their tokens transferred to an incorrectly configured market, leading to unrecoverable funds. If their approval to `Vault.sol` was set to the unlimited amount, malicious users will also be able to force honest market creators to transfer more tokens than they would normally want to allow.

Proof of Concept https://github.com/code-423n4/2022-01-insure/blob/main/contracts/Factory.sol#L158-L231

```
function createMarket(
    IUniversalMarket _template,
    string memory _metaData,
    uint256[] memory _conditions,
    address[] memory _references
) public override returns (address) {
    //check eligibility
    require(
        templates[address(_template)].approval == true,
        "ERROR: UNAUTHORIZED_TEMPLATE"
    );
    if (templates[address(_template)].isOpen == false) {
        require(
            ownership.owner() == msg.sender,
            "ERROR: UNAUTHORIZED_SENDER"
        );
    }
    if (_references.length > 0) {
        for (uint256 i = 0; i < _references.length; i++) {
            require(
                reflist[address(_template)][i][_references[i]] == true ||
                    reflist[address(_template)][i][address(0)] == true,
                "ERROR: UNAUTHORIZED_REFERENCE"
            );
        }
    }

    if (_conditions.length > 0) {
        for (uint256 i = 0; i < _conditions.length; i++) {
            if (conditionlist[address(_template)][i] > 0) {
                _conditions[i] = conditionlist[address(_template)][i];
            }
        }
    }

    if (
        IRegistry(registry).confirmExistence(
            address(_template),
            _references[0]
        ) == false
    ) {
        IRegistry(registry).setExistence(
            address(_template),
            _references[0]
        );
    } else {
        if (templates[address(_template)].allowDuplicate == false) {
            revert("ERROR: DUPLICATE_MARKET");
        }
    }

    //create market
    IUniversalMarket market = IUniversalMarket(
        _createClone(address(_template))
    );

    IRegistry(registry).supportMarket(address(market));

    markets.push(address(market));


    //initialize
    market.initialize(_metaData, _conditions, _references);

    emit MarketCreated(
        address(market),
        address(_template),
        _metaData,
        _conditions,
        _references
    );

    return address(market);
}
```

https://github.com/code-423n4/2022-01-insure/blob/main/contracts/PoolTemplate.sol#L178-L221

```
function initialize(
    string calldata _metaData,
    uint256[] calldata _conditions,
    address[] calldata _references
) external override {
    require(
        initialized == false &&
            bytes(_metaData).length > 0 &&
            _references[0] != address(0) &&
            _references[1] != address(0) &&
            _references[2] != address(0) &&
            _references[3] != address(0) &&
            _references[4] != address(0) &&
            _conditions[0] <= _conditions[1],
        "ERROR: INITIALIZATION_BAD_CONDITIONS"
    );
    initialized = true;

    string memory _name = string(
        abi.encodePacked(
            "InsureDAO-",
            IERC20Metadata(_references[1]).name(),
            "-PoolInsurance"
        )
    );
    string memory _symbol = string(
        abi.encodePacked("i-", IERC20Metadata(_references[1]).symbol())
    );
    uint8 _decimals = IERC20Metadata(_references[0]).decimals();

    initializeToken(_name, _symbol, _decimals);

    registry = IRegistry(_references[2]);
    parameters = IParameters(_references[3]);
    vault = IVault(parameters.getVault(_references[1]));
    metadata = _metaData;

    marketStatus = MarketStatus.Trading;

    if (_conditions[1] > 0) {
        _depositFrom(_conditions[1], _references[4]);
    }
}
```

Tools Used Manual code review. Discussions with kohshiba.

Recommended Mitigation Steps After discussions with the sponsor, they have opted to parse a `_creator` address to `PoolTemplate.sol` which will act as the depositor and be set to `msg.sender` in `Factory.createMarket()`. This will prevent malicious market creators from forcing vault deposits from unsuspecting users who are approved in `Factory.sol` and have also approved `Vault.sol` to make transfers on their behalf.

oishun1112 (Insure) confirmed: > https://github.com/code-423n4/2022-01-insure-findings/issues/250

[H-04] Initial pool deposit can be stolen

Submitted by cmichel, also found by WatchPug

Note that the `PoolTemplate.initialize` function, called when creating a market with `Factory.createMarket`, calls a vault function to transfer an initial deposit amount (`conditions[1]`) *from* the initial depositor (`_references[4]`):

```
// PoolTemplate
function initialize(
     string calldata _metaData,
     uint256[] calldata _conditions,
     address[] calldata _references
) external override {
     // ...

     if (_conditions[1] > 0) {
          // @audit vault calls asset.transferFrom(_references[4], vault, _conditions[1])
          _depositFrom(_conditions[1], _references[4]);
     }
}

function _depositFrom(uint256 _amount, address _from)
     internal
     returns (uint256 _mintAmount)
```

```
{
     require(
          marketStatus == MarketStatus.Trading && paused == false,
          "ERROR: DEPOSIT_DISABLED"
     );
     require(_amount > 0, "ERROR: DEPOSIT_ZERO");

     _mintAmount = worth(_amount);
     // @audit vault calls asset.transferFrom(_from, vault, _amount)
     vault.addValue(_amount, _from, address(this));

     emit Deposit(_from, _amount, _mintAmount);

     //mint iToken
     _mint(_from, _mintAmount);
}
```

The initial depositor needs to first approve the vault contract for the `transferFrom` to succeed.

An attacker can then frontrun the `Factory.createMarket` transaction with their own market creation (it does not have access restrictions) and create a market *with different parameters* but still passing in `_conditions[1]=amount` and `_references[4]=victim`.

A market with parameters that the initial depositor did not want (different underlying, old whitelisted registry/parameter contract, etc.) can be created with their tokens and these tokens are essentially lost.

Recommended Mitigation Steps Can the initial depositor be set to `Factory.createMarket`'s `msg.sender`, instead of being able to pick a whitelisted one as `_references[4]`?

oishun1112 (Insure) confirmed: > https://github.com/code-423n4/2022-01-insure-findings/issues/224

[H-05] backdoor in `withdrawRedundant`

Submitted by cmichel, also found by camden, WatchPug, and Ruhum

The `Vault.withdrawRedundant` has wrong logic that allows the admins to steal the underlying vault token.

```
function withdrawRedundant(address _token, address _to)
     external
     override
     onlyOwner
{
     if (
            _token == address(token) &&
            balance < IERC20(token).balanceOf(address(this))
        ) {
            uint256 _redundant = IERC20(token).balanceOf(address(this)) -
                balance;
            IERC20(token).safeTransfer(_to, _redundant);
        } else if (IERC20(_token).balanceOf(address(this)) > 0) {
            // @audit they can rug users. let's say balance == IERC20(token).balanceOf(address
            IERC20(_token).safeTransfer(
                _to,
                IERC20(_token).balanceOf(address(this))
            );
        }
}
```

POC

- Vault deposits increase as `Vault.addValue` is called and the `balance` increases by `_amount` as well as the actual `IERC20(token).balanceOf(this)`. Note that `balance == IERC20(token).balanceOf(this)`
- Admins call `vault.withdrawRedundant(vault.token(), attacker)` which goes into the `else if` branch due to the balance inequality condition being `false`. It will transfer out all `vault.token()` amounts to the attacker.

Impact There's a backdoor in the `withdrawRedundant` that allows admins to steal all user deposits.

Recommended Mitigation Steps I think the devs wanted this logic from the code instead:

```
function withdrawRedundant(address _token, address _to)
    external
    override
    onlyOwner
{
    if (
        _token == address(token)
    ) {
        if (balance < IERC20(token).balanceOf(address(this))) {
            uint256 _redundant = IERC20(token).balanceOf(address(this)) -
                balance;
            IERC20(token).safeTransfer(_to, _redundant);
        }
    } else if (IERC20(_token).balanceOf(address(this)) > 0) {
        IERC20(_token).safeTransfer(
            _to,
            IERC20(_token).balanceOf(address(this))
        );
    }
}
```

oishun1112 (Insure) confirmed: > similar to PVE03 (Peckshield audit) > We will create a PR and merge after we merge both audit/code4rena and audit/peckshield branches in the InsureDAO repository.

[H-06] the first depositor to a pool can drain all users

Submitted by danb

https://github.com/code-423n4/2022-01-insure/blob/main/contracts/PoolTemplate.sol#L807 if there is no liquidity in the pool, the first deposit determines the total liquidity, if the amount is too small the minted liquidity for the next liquidity providers will round down to zero.

Impact An attacker can steal all money from liquidity providers.

Proof of Concept consider the following scenario: a pool is created. the attacker is the first one to deposit, they deposit with _amount == 1, the smallest amount possible. meaning the total liquidity is 1. then they join another pool in order to get attributions in the vault. they transfer the attributions to the pool using `transferAttribution`. for example, they transferred 1M dollar worth of attributions. the next person deposits in the index, for example, 500,000 dollars. https://github.com/code-423n4/2022-01-insure/blob/main/contracts/PoolTemplate.sol#L803 the amount they will get is:

```
_amount = (_value * _supply) / _originalLiquidity;
```

as we know: _amount = 500,000 dollar _supply = 1 _totalLiquidity = 1,000,000 dollar (the attacker transferred directly) the investor will get (500,000 dollar * 1) / (1,000,000 dollar) = 0 and they will pay 500,000 this money will go to the index, and the attacker holds all of the shares, so they can withdraw it and get 1,500,000 stealing 500,000 dollars from the second investor.

oishun1112 (Insure) acknowledged and disagreed with severity: > yes. Every address that has attributions can call transferAttribution(), however, the address has to call addValue() to earn attributions. addValue() has onlyMarket modifier. > To pass onlyMarket modifier, ownership has to be stolen, in short. > Since we assume ownership control is driven safely, we don't take this as an issue.

0xean (judge) commented: > Agree with warden that the privilege addresses should not be able to use approvals in a way that rugs users funds. > > Based on the fact that we have seen many rug pulls in the space based on compromised "owner" keys, this is a valid attack path. > > `> 3 - High: Assets can` `be stolen/lost/compromised directly (or indirectly if there is a valid attack path that does not have hand-wavy hypotheticals). >`

[H-07] Wrong design/implementation of permission control allows malicious/compromised Registry or Factory admin to steal funds from users' wallet balances

Submitted by WatchPug

The current design/implementation allows a `market` address (registered on `registry`) to call `Vault#addValue()` and transfer tokens from an arbitrary address to a specified `_beneficiary` up the approved amount at any time, and the `_beneficiary` can withdraw the funds by calling `Vault#withdrawAllAttribution()` immediately.

This poses a very dangerous risk to all the users that approved their tokens to the Vault contracts (each one holds all users' allowances for that token).

https://github.com/code-423n4/2022-01-insure/blob/19d1a7819fe7ce795e6d48 14e7ddf8b8e1323df3/contracts/Vault.sol#L52-L58

```
modifier onlyMarket() {
    require(
        IRegistry(registry).isListed(msg.sender),
        "ERROR_ONLY_MARKET"
    );
    _;
}
```

https://github.com/code-423n4/2022-01-insure/blob/19d1a7819fe7ce795e6d48 14e7ddf8b8e1323df3/contracts/Vault.sol#L124-L140

```
function addValue(
    uint256 _amount,
    address _from,
    address _beneficiary
) external override onlyMarket returns (uint256 _attributions) {

    if (totalAttributions == 0) {
        _attributions = _amount;
    } else {
        uint256 _pool = valueAll();
        _attributions = (_amount * totalAttributions) / _pool;
    }
    IERC20(token).safeTransferFrom(_from, address(this), _amount);
    balance += _amount;
    totalAttributions += _attributions;
    attributions[_beneficiary] += _attributions;
}
```

Registry owner can call `Registry#supportMarket()` and mark an arbitrary address as a `market`.

https://github.com/code-423n4/2022-01-insure/blob/19d1a7819fe7ce795e6d48 14e7ddf8b8e1323df3/contracts/Registry.sol#L49-L60

```
function supportMarket(address _market) external override {
    require(!markets[_market], "ERROR: ALREADY_REGISTERED");
    require(
        msg.sender == factory || msg.sender == ownership.owner(),
        "ERROR: UNAUTHORIZED_CALLER"
    );
    require(_market != address(0), "ERROR: ZERO_ADDRESS");

    allMarkets.push(_market);
    markets[_market] = true;
    emit NewMarketRegistered(_market);
}
```

Or, the owner of the Factory can call `createMarket()` to add a malicous market contract via a custom template contract to the `markets` list.

https://github.com/code-423n4/2022-01-insure/blob/19d1a7819fe7ce795e6d48 14e7ddf8b8e1323df3/contracts/Factory.sol#L214-L216

Proof of Concept A malicious/compromised Registry owner can:

1. Call `Registry#supportMarket()` and set `markets[attackerAddress]` to `true`;
2. Call `Vault#addValue(token.balanceOf(victimAddress), victimAddress, attackerAddress)` and transferring all the balanceOf victim's wallet to the vault, owned by `attackerAddress`.
3. Call `Vault#withdrawAllAttribution(attackerAddress)` and retrive the funds.

The malicious/compromised Registry owner can repeat the steps above for all the users who approved the Vault contract for all the Vault contracts.

As a result, the attacker can steal all the wallet balances of the tokens approved to the protocol.

Root Cause Improper access control for using users' allowances.

Recommendation Consider changing the design/implementation to make sure that the allowances approved by the users can only be used by themselves.

oishun1112 (Insure) acknowledged and disagreed with severity: > this is an issue only when ownership control has fail. This architecture is necessary to achieve simplicity of the code. > We assume ownership control works fine.

0xean (judge) commented: > Agree with warden that the privilege addresses should not be able to use approvals in a way that rugs users funds. > > Based on the fact that we have seen many rug pulls in the space based on compromised "owner" keys, this is a valid attack path. > > `> 3 - High: Assets can be stolen/lost/compromised directly (or indirectly if there is a valid attack path that does not have hand-wavy hypotheticals). >`

[H-08] `IndexTemplate.sol#compensate()` will most certainly fail

Submitted by WatchPug

Precision loss while converting between `the amount of shares` and `the amount of underlying tokens` back and forth is not handled properly.

---

https://github.com/code-423n4/2022-01-insure/blob/19d1a7819fe7ce795e6d4814e7ddf8b8e1323df3/contracts/IndexTemplate.sol#L438-L447

```
uint256 _shortage;
if (totalLiquidity() < _amount) {
    //Insolvency case
    _shortage = _amount - _value;
    uint256 _cds = ICDSTemplate(registry.getCDS(address(this)))
        .compensate(_shortage);
    _compensated = _value + _cds;
}
vault.offsetDebt(_compensated, msg.sender);
```

In the current implementation, when someone tries to resume the market after a pending period ends by calling `PoolTemplate.sol#resume()`, `IndexTemplate.sol#compensate()` will be called internally to make a payout. If the index pool is unable to cover the compensation, the CDS pool will then be used to cover the shortage.

However, while `CDSTemplate.sol#compensate()` takes a parameter for the amount of underlying tokens, it uses `vault.transferValue()` to transfer corresponding `_attributions` (shares) instead of underlying tokens.

Due to precision loss, the `_attributions` transferred in the terms of underlying tokens will most certainly be less than the shortage.

At L444, the contract believes that it's been compensated for `_value + _cds`, which is lower than the actual value, due to precision loss.

At L446, when it calls `vault.offsetDebt(_compensated, msg.sender)`, the tx will revert at `require(underlyingValue(msg.sender) >= _amount)`.

As a result, `resume()` can not be done, and the debt can't be repaid.

Proof of Concept Given:

- vault.underlyingValue = 10,000
- vault.valueAll = 30,000
- totalAttributions = 2,000,000
- _amount = 1,010,000

0. _shortage = _amount - vault.underlyingValue = 1,000,000
1. _attributions = (_amount * totalAttributions) / valueAll = 67,333,333
2. actualValueTransfered = (valueAll * _attributions) / totalAttributions = 1009999

Expected results: actualValueTransfered = _shortage;

Actual results: actualValueTransfered < _shortage.

Impact The precision loss isn't just happening on special numbers, but will most certainly always revert the txs.

This will malfunction the contract as the index pool can not `compensate()`, therefore the pool can not `resume()`. Causing the funds of the LPs of the pool and the index pool to be frozen, and other stakeholders of the same vault will suffer fund loss from an unfair share of the funds compensated before.

Recommendation Change to:

https://github.com/code-423n4/2022-01-insure/blob/19d1a7819fe7ce795e6d48 14e7ddf8b8e1323df3/contracts/IndexTemplate.sol#L439-L446

```
if (totalLiquidity() < _amount) {
    //Insolvency case
    _shortage = _amount - _value;
    uint256 _cds = ICDSTemplate(registry.getCDS(address(this))
        .compensate(_shortage);
    _compensated = vault.underlyingValue(address(this));
}
vault.offsetDebt(_compensated, msg.sender);
```

oishun1112 (Insure) confirmed and disagreed with severity

oishun1112 (Insure) resolved

[H-09] `Vault#setController()` owner of the Vault contracts can drain funds from the Vault

Submitted by WatchPug

https://github.com/code-423n4/2022-01-insure/blob/19d1a7819fe7ce795e6d48 14e7ddf8b8e1323df3/contracts/Vault.sol#L485-L496

```
function setController(address _controller) public override onlyOwner {
    require(_controller != address(0), "ERROR_ZERO_ADDRESS");

    if (address(controller) != address(0)) {
        controller.migrate(address(_controller));
        controller = IController(_controller);
    } else {
        controller = IController(_controller);
    }

    emit ControllerSet(_controller);
}
```

The owner of the Vault contract can set an arbitrary address as the `controller`.

https://github.com/code-423n4/2022-01-insure/blob/19d1a7819fe7ce795e6d48 14e7ddf8b8e1323df3/contracts/Vault.sol#L342-L352

```
function utilize() external override returns (uint256 _amount) {
    if (keeper != address(0)) {
        require(msg.sender == keeper, "ERROR_NOT_KEEPER");
    }
    _amount = available(); //balance
    if (_amount > 0) {
        IERC20(token).safeTransfer(address(controller), _amount);
        balance -= _amount;
        controller.earn(address(token), _amount);
    }
}
```

A malicious `controller` contract can transfer funds from the Vault to the attacker.

Proof of Concept A malicious/compromised can:

1. Call `Vault#setController()` and set `controller` to a malicious contract;
   - L489 the old controller will transfer funds to the new, malicious controller.
2. Call `Vault#utilize()` to deposit all the balance in the Vault contract into the malicious controller contract.
3. Withdraw all the funds from the malicious controller contract.

Recommendation Consider disallowing `Vault#setController()` to set a new address if a controller is existing, which terminates the possibility of migrating funds to a specified address provided by the owner. Or, putting a timelock to this function at least.

oishun1112 (Insure) acknowledged and disagreed with severity: > we assume ownership control is driven safely

0xean (judge) commented: > Agree with warden that the privilege addresses should not be able to use approvals in a way that rugs users funds. > > Based on the fact that we have seen many rug pulls in the space based on compromised "owner" keys, this is a valid attack path. > `> 3 - High: Assets can be stolen/lost/compromised directly (or indirectly if there is a valid attack path that does not have hand-wavy hypotheticals). >`

[H-10] A malicious/compromised Registry or Factory admin can drain all the funds from the Vault contracts

Submitted by WatchPug

https://github.com/code-423n4/2022-01-insure/blob/19d1a7819fe7ce795e6d48 14e7ddf8b8e1323df3/contracts/Vault.sol#L52-L58

```
modifier onlyMarket() {
    require(
        IRegistry(registry).isListed(msg.sender),
        "ERROR_ONLY_MARKET"
    );
    _;
}
```

https://github.com/code-423n4/2022-01-insure/blob/19d1a7819fe7ce795e6d48 14e7ddf8b8e1323df3/contracts/Vault.sol#L201-L206

```
function borrowValue(uint256 _amount, address _to) external onlyMarket override {
    debts[msg.sender] += _amount;
    totalDebt += _amount;

    IERC20(token).safeTransfer(_to, _amount);
}
```

The current design/implementation allows a market address (registered on the `registry`) to call `Vault#borrowValue()` and transfer tokens to an arbitrary address.

Proof of Concept See the PoC section on [WP-H24].

Recommendation

1. Consider adding constrains (eg. timelock) to `Registry#supportMarket()`.
2. Consdier adding constrains (upper bound for each pool, and index pool for example) to `Vault#borrowValue()`.

oishun1112 (Insure) acknowledged and disagreed with severity: > Ownership has to be stolen to drain funds using this method and we assume ownership control driven safely, so we don't treat this as issue

0xean (judge) commented: > Agree with warden that the privilege addresses should not be able to use approvals in a way that rugs users funds. > > Based on the fact that we have seen many rug pulls in the space based on compromised "owner" keys, this is a valid attack path. > `> 3 - High: Assets can be stolen/lost/compromised directly (or indirectly if there is a valid attack path that does not have hand-wavy hypotheticals). >`

[H-11] `PoolTemplate.sol#resume()` Wrong implementation of `resume()` will compensate overmuch redeem amount from index pools

Submitted by WatchPug, also found by danb

Wrong arithmetic.

---

https://github.com/code-423n4/2022-01-insure/blob/19d1a7819fe7ce795e6d4814e7ddf8b8e1323df3/contracts/PoolTemplate.sol#L700-L717

```
uint256 _deductionFromIndex = (_debt * _totalCredit * MAGIC_SCALE_1E6) /
            totalLiquidity();
    uint256 _actualDeduction;
    for (uint256 i = 0; i < indexList.length; i++) {
        address _index = indexList[i];
        uint256 _credit = indicies[_index].credit;
        if (_credit > 0) {
            uint256 _shareOfIndex = (_credit * MAGIC_SCALE_1E6) /
                _totalCredit;
            uint256 _redeemAmount = _divCeil(
                _deductionFromIndex,
                _shareOfIndex
            );
            _actualDeduction += IIndexTemplate(_index).compensate(
                _redeemAmount
            );
        }
    }
```

Proof of Concept

- totalLiquidity = 200,000* 10**18;
- totalCredit = 100,000 * 10**18;
- debt = 10,000 * 10**18;
- [Index Pool 1] Credit = 20,000 * 10**18;
- [Index Pool 2] Credit = 30,000 * 10**18;

```
uint256 _deductionFromIndex = (_debt * _totalCredit * MAGIC_SCALE_1E6) /
            totalLiquidity();
// _deductionFromIndex = 10,000 * 10**6 * 10**18;
```

[Index Pool 1]:

```
uint256 _shareOfIndex = (_credit * MAGIC_SCALE_1E6) / _totalCredit;
//  _shareOfIndex = 200000

uint256 _redeemAmount = _divCeil(
    _deductionFromIndex,
    _shareOfIndex
);

// _redeemAmount = 25,000 * 10**18;
```

[Index Pool 2]:

```
uint256 _shareOfIndex = (_credit * MAGIC_SCALE_1E6) / _totalCredit;
//  _shareOfIndex = 300000

uint256 _redeemAmount = _divCeil(
    _deductionFromIndex,
    _shareOfIndex
);

// _redeemAmount = 16666666666666666666667 (~ 16,666 * 10**18)
```

In most cases, the transaction will revet on underflow at:

```
uint256 _shortage = _deductionFromIndex /
    MAGIC_SCALE_1E6 -
    _actualDeduction;
```

In some cases, specific pools will be liable for unfair compensation:

If the CSD is empty, `Index Pool 1` only have `6,000 * 10**18` and `Index Pool 2` only have `4,000 * 10**18`, the `_actualDeduction` will be `10,000 * 10**18`, `_deductionFromPool` will be `0`.

`Index Pool 1` should only pay `1,000 * 10**18`, but actually paid `6,000 * 10**18`, the LPs of `Index Pool 1` now suffer funds loss.

Recommendation Change to:

```
uint256 _deductionFromIndex = (_debt * _totalCredit * MAGIC_SCALE_1E6) / totalLiquidity();
uint256 _actualDeduction;
for (uint256 i = 0; i < indexList.length; i++) {
    address _index = indexList[i];
    uint256 _credit = indicies[_index].credit;
    if (_credit > 0) {
        uint256 _shareOfIndex = (_credit * MAGIC_SCALE_1E6) /
            _totalCredit;
        uint256 _redeemAmount = _divCeil(
            _deductionFromIndex * _shareOfIndex,
            MAGIC_SCALE_1E6 * MAGIC_SCALE_1E6
        );
        _actualDeduction += IIndexTemplate(_index).compensate(
            _redeemAmount
        );
    }
}
```

oishun1112 (Insure) confirmed and resolved

[H-12] `IndexTemplate.sol` Wrong implementation allows lp of the index pool to resume a locked `PayingOut` pool and escape the responsibility for the compensation

Submitted by WatchPug, also found by leastwood

Based on the context, the system intends to lock all the lps during PayingOut period.

However, the current implementation allows anyone, including LPs to call `resume()` and unlock the index pool.

It allows a malicious LP to escape the responsibility for the compensation, at the expense of other LPs paying more than expected.

https://github.com/code-423n4/2022-01-insure/blob/19d1a7819fe7ce795e6d4814e7ddf8b8e1323df3/contracts/IndexTemplate.sol#L459-L471

```
function resume() external override {
    uint256 _poolLength = poolList.length;

    for (uint256 i = 0; i < _poolLength; i++) {
        require(
            IPoolTemplate(poolList[i]).paused() == false,
            "ERROR: POOL_IS_PAUSED"
        );
    }
    locked = false;
    emit Resumed();
}
```

Recommendation Change to:

```
function resume() external override {
   uint256 _poolLength = poolList.length;

   for (uint256 i = 0; i < _poolLength; i++) {
       require(
           IPoolTemplate(poolList[i]).marketStatus() == MarketStatus.Trading,
           "ERROR: POOL_IS_PAYINGOUT"
       );
   }

   locked = false;
   emit Resumed();
}
```

oishun1112 (Insure) confirmed

[H-13] Admin of the index pool can `withdrawCredit()` after `applyCover()` to avoid taking loss for the compensation paid for a certain pool

Submitted by WatchPug

In the current implementation, when an incident is reported for a certain pool, the index pool can still `withdrawCredit()` from the pool, which in the best interest of an index pool, the admin of the index pool is preferred to do so.

This allows the index pool to escape from the responsibility for the risks of invested pools.

Making the LPs of the pool take an unfair share of the responsibility.

Proof of Concept

- Pool A `totalCredit` = 10,000
- Pool A `rewardPerCredit` = 1

1. [Index Pool 1] allocates 1,000 credits to Pool `A`:

- `totalCredit` = 11,000
- indicies[Index Pool 1] = 1,000

2. After a while, Pool A `rewardPerCredit` has grown to `1.1`, and `applyCover()` has been called, [Index Pool 1] call `withdrawCredit()` get 100 premium

- `totalCredit` = 10,000
- indicies[Index Pool 1] = 0

3. After `pendingEnd`, the pool `resume()`,[Index Pool 1] will not be paying for the compensation since `credit` is 0.

In our case, [Index Pool 1] earned premium without paying for a part of the compensation.

Recommendation Change to:

https://github.com/code-423n4/2022-01-insure/blob/19d1a7819fe7ce795e6d4814e7ddf8b8e1323df3/contracts/PoolTemplate.sol#L416-L421

```
function withdrawCredit(uint256 _credit)
    external
    override
    returns (uint256 _pending)
{
    require(
        marketStatus == MarketStatus.Trading,
        "ERROR: WITHDRAW_CREDIT_BAD_CONDITIONS"
    );
    IndexInfo storage _index = indicies[msg.sender];
```

oishun1112 (Insure) confirmed and disagreed with severity: > to call PoolTemplate: withdrawCredit(), someone has to call IndexTemplate: withdraw(), set(), and adjustAlloc(). > > set() is onlyOwner, so we assume it's fine() > adjustAlloc() is public. this clean up and flatten the credit distribution. > withdraw() is public. this reduce totalCredit to distribute. when exceed upperSlack, call adjustAlloc(). > > We should lock the credit control when pool is in payout status. > This implementation, still allows small amount of withdraw, for users who were requested Withdraw.

oishun1112 (Insure) commented: > We have fixed with PVE02 (Peckshield audit) issue together.

Medium Risk Findings (9)

[M-01] repayDebt in Vault.sol could DOS functionality for markets

Submitted by p4st13r4

Any user can pay the debt for any borrower in `Vault.sol`, by using `repayDebt()`. This function allows anyone to repay any amount of borrowed value, up-to and including the `totalDebt` value; it works by setting the `debts[_target]` to zero, and decreasing `totalDebt` by the given amount, up to zero. However, all debts of the other borrowers are left untouched.

If a malicious (but generous) user were to repay the debt for all the borrowers, markets functionality regarding borrowing would be DOSed: the vault would try to decrease the debt of the market, successfully, but would fail to decrease `totalDebt` as it would result in an underflow

Proof of Concept https://github.com/code-423n4/2022-01-insure/blob/main/contracts/Vault.sol#L257

Recommended Mitigation Steps Make `repayDebt()` accept an amount up-to and including the value of the debt for the given borrower

oishun1112 (Insure) confirmed: > this needs to be specified how in more detail.

[M-02] Owner can call `applyCover` multiple times in `PoolTemplate.sol`

Submitted by camden

The owner could potentially extend the insurance period indefinitely in the `applyCover` function without ever allowing the market to resume. This is because there is no check in `applyCover` to ensure that the market is in a `Trading` state.

This can also allow the owner to emit fraudulent `MarketStatusChanged` events.

Recommended Mitigation Steps Require that the market be in a `Trading` state to allow another `applyCover` call.

oishun1112 (Insure) confirmed and resolved: > this behaviour is not intended, so confirmed.

0xean (judge) commented: > upgrading to medium severity since it alters the function of the protocol > > `> 2 - Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements. >`

[M-03] Signature replay

Submitted by 0x1f8b

Signature replay in `PoolTemplate`.

Proof of Concept The `redeem` method of `PoolTemplate` verifies the data stored in `incident`, and the verification logic of this process is performed as following:

```
require(
    MerkleProof.verify(
        _merkleProof,
        _targets,
        keccak256(
            abi.encodePacked(_insurance.target, _insurance.insured)
        )
    ) ||
        MerkleProof.verify(
            _merkleProof,
            _targets,
            keccak256(abi.encodePacked(_insurance.target, address(0)))
        ),
    "ERROR: INSURANCE_EXEMPTED"
);
```

As can be seen, the only data related to the `_insurance` are`target` and `insured`, so as the incident has no relation with the`Insurance`, apparently nothing prevents a user to call `insure` with high amounts, after receive the incident, the only thing that prevents this from being reused is that the owner creates the incident with an `_incidentTimestamp` from the past.

So if an owner create a incident from the future it's possible to create a new `insure` that could be reused by the same affected address.

Another lack of input verification that could facilitate this attack is the `_span=0` in the `insure` method.

Recommended Mitigation Steps It is mandatory to add a check in `applyCover` that`_incidentTimestamp` is less than the current date and the `span` argument is greater than 0 in the`insure` method.

oishun1112 (Insure) confirmed and resolved, but disagreed with severity: > agree on the _incidentTimestamp check. > disagree on span check since there already is > `> require( >             parameters.getMinDate(msg.sender) <= _span, >             "ERROR: INSURE_SPAN_BELOW_MIN" >         );` `>` > we are going to set default value of 1week for everyone

oishun1112 (Insure) commented: > we assume ownership control works fine. > this can lose money in-proper way, but not at risk since onlyOwner modifier applied.

0xean (judge) commented: > going to leave this as 2 > > `> 2 - Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a` `hypothetical attack path with stated assumptions, but external requirements.` > > > The external requirement here would be an incorrect timestamp from the owner which would cause assets to be at risk from the replay.

oishun1112 (Insure) commented: > there is > `> _span >= getMinDate()` `>` > so we don't implement _span > 0

[M-04] System Debt Is Not Handled When Insurance Pools Become Insolvent

Submitted by leastwood

If an incident has occurred where an insurance policy is to be redeemed. The market is put into the `MarketStatus.Payingout` mode where the `_insurance.insured` account is allowed to redeem their cover and receive a payout amount. Upon paying out the insurance cover, any user is able to resume the market by calling `PoolTemplate.resume()`. This function will compensate the insurance pool if it is insolvent by querying `IndexTemplate.compensate()` which in turn queries `CDSTemplate.compensate()` to cover any shortage.

In the event none of these entities are able to cover the shortage in debt, the system accrues the debt. However, there is currently no mechanism to ensure when `transferDebt()` is called in `PoolTemplate.resume()`, the accrued system debt is paid off. Therefore, the system may incorrectly handle insolvency on an extreme edge case, generating system instability.

Proof of Concept https://github.com/code-423n4/2022-01-insure/blob/main/contracts/PoolTemplate.sol#L691-L734

```
function resume() external {
    require(
        marketStatus == MarketStatus.Payingout &&
            pendingEnd < block.timestamp,
        "ERROR: UNABLE_TO_RESUME"
    );

    uint256 _debt = vault.debts(address(this));
    uint256 _totalCredit = totalCredit;
    uint256 _deductionFromIndex = (_debt * _totalCredit * MAGIC_SCALE_1E6) /
        totalLiquidity();
    uint256 _actualDeduction;
    for (uint256 i = 0; i < indexList.length; i++) {
        address _index = indexList[i];
        uint256 _credit = indicies[_index].credit;
        if (_credit > 0) {
            uint256 _shareOfIndex = (_credit * MAGIC_SCALE_1E6) /
```

```
                _totalCredit;
            uint256 _redeemAmount = _divCeil(
                _deductionFromIndex,
                _shareOfIndex
            );
            _actualDeduction += IIndexTemplate(_index).compensate(
                _redeemAmount
            );
        }
    }

    uint256 _deductionFromPool = _debt -
        _deductionFromIndex /
        MAGIC_SCALE_1E6;
    uint256 _shortage = _deductionFromIndex /
        MAGIC_SCALE_1E6 -
        _actualDeduction;

    if (_deductionFromPool > 0) {
        vault.offsetDebt(_deductionFromPool, address(this));
    }

    vault.transferDebt(_shortage);

    marketStatus = MarketStatus.Trading;
    emit MarketStatusChanged(MarketStatus.Trading);
}
```

- https://github.com/code-423n4/2022-01-insure/blob/main/contracts/IndexTemplate.sol#L421-L450
- https://github.com/code-423n4/2022-01-insure/blob/main/contracts/CDSTemplate.sol#L248-L277

Recommended Mitigation Steps Consider devising a mechanism to ensure system debt is properly handled. After discussions with the sponsor, it seems that they will be implementing a way to mint `INSURE` tokens which will be used to cover the shortfall.

oishun1112 (Insure) acknowledged > yes, PoolTemplate calls transferDebt() to make his debt to the system debt in case all Index and CDS layers couldn't cover the shortage. > In this case, we have to repay the system debt somehow since this is the situation that we over-lose money. One way is that someone calls repayDebt() and pay for it (not realistic at all). As we implement the way to payback, we are considering minting INSURE token or, other better mechanism. > > This is not developed yet, and acknowledged.

[M-05] `Vault.sol` Tokens with fee on transfer are not supported

Submitted by WatchPug, also found by pmerkleplant, cmichel, Ruhum, and Dravee

There are ERC20 tokens that charge fee for every `transfer()` / `transferFrom()`.

`Vault.sol#addValue()` assumes that the received amount is the same as the transfer amount, and uses it to calculate attributions, balance amounts, etc. While the actual transferred amount can be lower for those tokens.

https://github.com/code-423n4/2022-01-insure/blob/19d1a7819fe7ce795e6d4814e7ddf8b8e1323df3/contracts/Vault.sol#L124-L140

```
function addValue(
    uint256 _amount,
    address _from,
    address _beneficiary
) external override onlyMarket returns (uint256 _attributions) {

    if (totalAttributions == 0) {
        _attributions = _amount;
    } else {
        uint256 _pool = valueAll();
        _attributions = (_amount * totalAttributions) / _pool;
    }
    IERC20(token).safeTransferFrom(_from, address(this), _amount);
    balance += _amount;
    totalAttributions += _attributions;
    attributions[_beneficiary] += _attributions;
}
```

Recommendation Consider comparing before and after balance to get the actual transferred amount.

oishun1112 (Insure) acknowledged: > only USDC can be underwriting asset. We are trying to make it multi currency. > We won't implement this now due to the gas consumption, but we will when we develop new type of Vault

[M-06] Index compensate is 0 when totalLiquidity() is enough to cover the whole amount

Submitted by pauliax

In IndexTemplate, function compensate, When `\_amount > \_value`, and `<= totalLiquidity()`, the value of `\_compensated` is not set, so it gets a default value of 0:

```
if (_value >= _amount) {
    ...
    _compensated = _amount;
} else {
    ...
    if (totalLiquidity() < _amount) {
        ...
        _compensated = _value + _cds;
    }
    vault.offsetDebt(_compensated, msg.sender);
}
```

But nevertheless, in both cases, it calls `vault.offsetDebt`, even when the_`compensated` is 0 (no else block).

Recommended Mitigation Steps I think, in this case, it should try to redeem the premium (withdrawCredit?) to cover the whole amount, but I am not sure about the intentions as I didn't have enough time to understand this protocol in depth.

oishun1112 (Insure) confirmed and resolved: > Right. > totalLiquidity = underlyingValue + pendingPremium. > > I will discuss how to fix this issue with my team

[M-07] `requestWithdraw` without obligation to withdraw allow underwriter to avoid payout

Submitted by gzeon

To prevent withdrawal front-running, a lockup period is set between withdrawal request and withdrawal. However, there are no obligation to withdraw after the lockup period and the capital will keep earning premium during lockup. A strategy for underwriter is to keep requesting withdrawal every `lockup period` to keep their average lockup to `lockup period/2`.

Proof of Concept https://github.com/code-423n4/2022-01-insure/blob/19d1a7819fe7ce795e6d4814e7ddf8b8e1323df3/contracts/PoolTemplate.sol#L279

Assuming

1. Reporting DAO vote last for 24 hours (according to docs) plus there will be delay between the hack and vote creation
2. the `lockup period` is set to 86400 (24 hours) in the supplied test cases

It is very likely an underwriter can avoid payout by such strategy since their effective lockup would be 12 hours only. They will continue to earn yield in the pool and only require some extra gas cost for the `requestWithdraw` every 24 hours.

Recommended Mitigation Steps Extend the lockup period at least by a factor of 2 or force underwriter to withdraw after lockup period.

oishun1112 (Insure) acknowledged: > Yes, lock up period is going to be like a week~2week in production.

[M-08] Unbounded iteration over all indexes (2)

Submitted by Dravee, also found by robee, egjlmn1, danb, WatchPug, Fitraldys, and Ruhum

The transactions could fail if the array get too big and the transaction would consume more gas than the block limit. This will then result in a denial of service for the desired functionality and break core functionality.

Proof of Concept https://github.com/code-423n4/2022-01-insure/blob/main/contracts/PoolTemplate.sol#L703

Tools Used VS Code

Recommended Mitigation Steps Keep the array size small.

oishun1112 (Insure) confirmed

0xean (judge) commented: > Upgrading to sev-2 as this will eventually affect the availability of the protocol as transactions revert.

Low Risk Findings (42)

- [L-01] Missing validation of address argument could indefinitely lock Registry contract *Submitted by defsec*
- [L-02] Lack of inputs in Factory *Submitted by 0x1f8b, also found by robee, Dravee, ospwner, hyh, and Meta0xNull*
- [L-03] Wrong revert string in withdraw functions *Submitted by p4st13r4*
- [L-04] resume() can be called by anyone in IndexTemplate.sol *Submitted by p4st13r4*
- [L-05] No-op in CDSTemplate.sol' withdraw() *Submitted by p4st13r4, also found by tqts, 0xngndev, cmichel, Fitraldys, and pauliax*
- [L-06] Require with empty message *Submitted by robee*
- [L-07] No slippage control in CDSTemplate.sol = frontrun or sandwich *Submitted by sirhashalot*
- [L-08] Incorrect return value comment *Submitted by sirhashalot*
- [L-09] The fund function of the CDSTemplate contract does not match the description *Submitted by cccz, also found by WatchPug*
- [L-10] Edge case in withdrawValue may lead to failed transactions *Submitted by 0xngndev*
- [L-11] Incorrect Natspec can lead to errors *Submitted by 0xngndev*

- [L-12] Named return issue *Submitted by robee*
- [L-13] PoolTemplate worth function description is incorrect *Submitted by hyh*
- [L-14] Vault. withdrawValue will fail on subtraction if there are not enough _attributions *Submitted by hyh*
- [L-15] Factory. createMarket doesn't check if input array length is too big *Submitted by hyh*
- [L-16] Two Steps Verification before Transferring Ownership *Submitted by robee*
- [L-17] Assert instead require to validate user inputs *Submitted by robee, also found by 0xngndev, defsec, Dravee, hyh, and WatchPug*
- [L-18] Allowance checks not correctly implemented *Submitted by defsec*
- [L-19] Implement check effect interaction to align with best practices *Submitted by defsec, also found by sirhashalot*
- [L-20] Improper Upper Bound Definition on the Fee *Submitted by defsec, also found by Dravee and pauliax*
- [L-21] Inconsistency in pragma solidity version definition in InsureDAO-ERC20.sol *Submitted by hubble, also found by robee*
- [L-22] Input validation not done in few important functions in Parameters.sol *Submitted by hubble*
- [L-23] Can create market without some conditions *Submitted by cmichel*
- [L-24] Future owner is not cleared *Submitted by cmichel*
- [L-25] Lower slack can be higher than upper slack *Submitted by cmichel*
- [L-26] Premium payments can be timed *Submitted by cmichel*
- [L-27] Keeper, not controller: Vault.setKeeper and utilize descriptions are incorrect *Submitted by hyh*
- [L-28] `Vault#setController()` Lack of validation for the amount of migrated funds *Submitted by WatchPug*
- [L-29] `Vault#_unutilize()` Lack of validation for the amount of funds received *Submitted by WatchPug*
- [L-30] Wrong comment on fund function *Submitted by Fitraldys*
- [L-31] Inconsistent divide by 0 checks for `totalSupply()` *Submitted by Dravee*
- [L-32] Spec error on function: `Factory:approveTemplate` *Submitted by Dravee*
- [L-33] Spec error on function: `Factory:setCondition` (difference with code comment) *Submitted by Dravee*
- [L-34] Skip balance check in _beforeTokenTransfer if no withdrawalRequest exists *Submitted by TomFrenchBlockchain*
- [L-35] `targetLev` can be set to 0 in `IndexTemplate:setLeverage` *Submitted by Dravee*
- [L-36] getLockup and getWithdrawable can change after withdrawalReq is initiated *Submitted by pauliax*
- [L-37] Validate _to is not empty *Submitted by pauliax*
- [L-38] Add a timelock to `Parameters:setFeeRate` *Submitted by Dravee*
- [L-39] Pause check missing on the several functions (PoolTemplate) *Sub-*

mitted by defsec
- [L-40] Insurance Pool Locking Does Not Propagate To All Markets *Submitted by leastwood*
- [L-41] Parameters.sol lacks input validation *Submitted by cccz*
- [L-42] Uncontrolled call to controller, which can be the zero address *Submitted by camden*

Non-Critical Findings (15)

- [N-01] Emit an event in setKeeper() *Submitted by p4st13r4*
- [N-02] In PoolTemplate.sol, deposit() and _depositFrom() can re-use the same code *Submitted by p4st13r4*
- [N-03] Inaccurate return value from `getCDS()` possible *Submitted by sirhashalot*
- [N-04] anyone can get money from an incident without paying beforehand *Submitted by danb*
- [N-05] Race condition on ERC20 approval *Submitted by WatchPug*
- [N-06] Missing error messages in require statements *Submitted by WatchPug*
- [N-07] Never used parameters *Submitted by robee, also found by sirhashalot*
- [N-08] too much centralization in the vault, the vault owner can withdraw all the value in the vault *Submitted by bugwriter001*
- [N-09] Typo for withdawable in multiple places in Parameters.sol *Submitted by hubble*
- [N-10] Insurance NFT *Submitted by pauliax*
- [N-11] Ordering importance in a struct *Submitted by Kumpirmafyas*
- [N-12] Misleading comments and documentation *Submitted by pauliax*
- [N-13] pool can't be initialized *Submitted by danb, also found by cccz, jayjonah8, camden, defsec, hyh, loop, Meta0xNull, and Fitraldys*
- [N-14] _depositFrom() does not ensure that _from arg is not the contract itself *Submitted by jayjonah8*

Gas Optimizations (98)

- [G-01] Unused imports *Submitted by robee, also found by defsec, Jujic, saian, WatchPug, and Ruhum*
- [G-02] in function _sub, less gas used using unchecked *Submitted by Tomio*
- [G-03] avoid using 'else' code can save gas in function pendingPremium *Submitted by Tomio*
- [G-04] split one require to two require can save gas *Submitted by Fitraldys*
- [G-05] Gas saving caching the value *Submitted by 0x1f8b*
- [G-06] unnecessary double `totalLiquidity()` call in function availableBalance *Submitted by Tomio*
- [G-07] save Insurance data directly to storage can save gas *Submitted by Fitraldys*

- [G-08] call emit from storage is more expensive *Submitted by Fitraldys*
- [G-09] Check if amount is not zero to save gas *Submitted by robee*
- [G-10] Avoid expensive storage reads in Parameters.sol *Submitted by p4st13r4*
- [G-11] acceptTransferOwnership() could save gas by using msg.sender *Submitted by p4st13r4*
- [G-12] commitTransferOwnership() could save gas 1 *Submitted by p4st13r4*
- [G-13] Save gas in requestWithdraw() *Submitted by p4st13r4*
- [G-14] totalAllocPoint in IndexTemplate.sol can be cached *Submitted by p4st13r4*
- [G-15] Use `calldata` instead of `memory` for function parameters *Submitted by defsec*
- [G-16] Remove unnecessary address cast in Vault.sol *Submitted by sirhashalot*
- [G-17] Unnecessary use of _msgSender() *Submitted by Jujic, also found by defsec*
- [G-18] Moving Variable Declarations Before Error Checks Can Save Gas on Failure *Submitted by 0xngndev, also found by ospwner*
- [G-19] Grouping Repeated Logic Into a Modifier To Save on Deployment Costs *Submitted by 0xngndev*
- [G-20] Caching variables *Submitted by Jujic, also found by p4st13r4*
- [G-21] Checking non-zero value can avoid an external call to save gas *Submitted by Jujic*
- [G-22] Avoid use of state variables in event emissions to save gas *Submitted by Jujic*
- [G-23] Remove unnecessary if statements for gas optimization *Submitted by ospwner*
- [G-24] set `_mintAmount = _amount;` in the memory can save gas *Submitted by Tomio*
- [G-25] Multiple boolean comparrisons *Submitted by loop, also found by robee, Tomio, 0xngndev, 0x1f8b, OriDabush, 0x0x0x, and defsec*
- [G-26] Gas optimization in Vault.addValueBatch() *Submitted by tqts, also found by OriDabush*
- [G-27] Avoid unnecessary code execution can save gas *Submitted by Jujic*
- [G-28] `InsureDAOERC20#transferFrom()` Do not reduce approval on transferFrom if current allowance is type(uint256).max *Submitted by WatchPug*
- [G-29] `AuctionBurnReserveSkew.sol#deposit()` Implementation can be simpler and save some gas *Submitted by WatchPug*
- [G-30] Remove unnecessary variables can save gas *Submitted by WatchPug*
- [G-31] Remove redundant code can save gas *Submitted by WatchPug*
- [G-32] `InsureDAOERC20#transferFrom()` Check of allowance can be done earlier to save gas *Submitted by WatchPug*
- [G-33] Check of `_amount > 0` can be done earlier to save gas *Submitted by WatchPug*
- [G-34] Cache function call results in the stack can save gas *Submitted by WatchPug, also found by p4st13r4*

- [G-35] `AuctionBurnReserveSkew.sol#deposit()` Implementation can be simpler and save some gas *Submitted by WatchPug*
- [G-36] Avoiding repeated `marketStatus` checks can save gas *Submitted by WatchPug*
- [G-37] Changing bool to uint256 can save gas *Submitted by WatchPug*
- [G-38] Constructor not used *Submitted by Jujic*
- [G-39] Struct layout *Submitted by Jujic, also found by TomFrenchBlockchain and WatchPug*
- [G-40] sqrt can be made unchecked to save gas *Submitted by TomFrenchBlockchain*
- [G-41] PoolTemplate.availableBalance calls totalLiquidity twice *Submitted by hyh*
- [G-42] Cache external call results can save gas *Submitted by WatchPug*
- [G-43] Avoiding unnecessary storage read can save gas *Submitted by WatchPug*
- [G-44] Withdrawal struct can be packed to save gas *Submitted by TomFrenchBlockchain*
- [G-45] using operator `&&` used more gas *Submitted by Tomio*
- [G-46] Change `public` constant variables to `private` / `internal` can save gas *Submitted by WatchPug, also found by robee and Jujic*
- [G-47] Gas: Optimize Conditional Statements in `CDSTemplate.sol:deposit()` *Submitted by Dravee*
- [G-48] `Factory:approveTemplate` could make 1 SSTORE instead of 3 *Submitted by Dravee*
- [G-49] Unused state variables *Submitted by robee, also found by sirhashalot*
- [G-50] Gas: Consider making some constants as non-public to save gas *Submitted by Dravee*
- [G-51] Gas: Optimize Conditional Statements in `IndexTemplate.sol:deposit()` *Submitted by Dravee*
- [G-52] Gas: Cache `totalLiquidity()` in `IndexTemplate:leverage()` *Submitted by Dravee*
- [G-53] Eliminate else block *Submitted by pauliax*
- [G-54] Repeated external calls *Submitted by pauliax*
- [G-55] Repeated storage reads *Submitted by pauliax*
- [G-56] repayDebt optimization *Submitted by pauliax*
- [G-57] Repeated math operations *Submitted by pauliax, also found by tqts*
- [G-58] deposit and _depositFrom are almost similar *Submitted by pauliax*
- [G-59] Gas Optimization: Use unchecked for safe math *Submitted by gzeon*
- [G-60] Unnecessary market status check on redemption *Submitted by TomFrenchBlockchain*
- [G-61] Gas: Cache `_fee[_target]` in `Parameters.sol:getFeeRate()` *Submitted by Dravee*
- [G-62] Order of statements *Submitted by pauliax*
- [G-63] Gas: Cache `attributions[_target]` in `Vault.sol:underlyingValue()` *Submitted by Dravee*
- [G-64] Gas: Avoid expensive calculation by checking if `valueAll() == 0`

Submitted by Dravee

- [G-65] Gas: `PoolTemplate:initialize()::_conditions` should be a fixed array of size 2 *Submitted by Dravee, also found by Dravee*
- [G-66] Gas: `incident.payoutNumerator` is used only once. It shouldn't be stored in a variable. *Submitted by Dravee*
- [G-67] Gas: `incident.payoutDenominator` is used only once. It shouldn't be stored in a variable. *Submitted by Dravee*
- [G-68] Gas: Optimize Conditional Statements in `PoolTemplate.sol:worth()` *Submitted by Dravee*
- [G-69] Gas: `> 0` is less efficient than `!= 0` for unsigned integers *Submitted by Dravee, also found by 0xngndev, defsec, Jujic, WatchPug, 0x0x0x, Ruhum, gzeon, and solgryn*
- [G-70] Gas: Avoid expensive calculation by checking if `originalLiquidity() == 0` *Submitted by Dravee*
- [G-71] allocatedCredit and availableBalance are always read together so should be returned together. *Submitted by TomFrenchBlockchain*
- [G-72] Gas: SafeMath is not needed when using Solidity version 0.8.* *Submitted by Dravee, also found by Jujic and defsec*
- [G-73] Redundant tracking of markets in factory *Submitted by TomFrench-Blockchain*
- [G-74] Gas: No need to initialize variables with default values *Submitted by Dravee, also found by robee, egjlmn1, Tomio, defsec, Jujic, Meta0xNull, WatchPug, 0x0x0x, and gzeon*
- [G-99] Redundant if statements in market deployment function *Submitted by TomFrenchBlockchain*
- [G-75] Update to solc-0.8.10+ *Submitted by TomFrenchBlockchain, also found by defsec*
- [G-76] Gas: An array's length should be cached to save gas in for-loops *Submitted by Dravee*
- [G-77] Storage double reading. Could save SLOAD *Submitted by robee*
- [G-78] Gas optimization in IndexTemplate.requestWithdraw() *Submitted by tqts*
- [G-79] Gas optimization in IndexTemplate._adjustAlloc() *Submitted by tqts*
- [G-80] Gas optimization in PoolTemplate.withdraw() *Submitted by tqts*
- [G-81] Gas: Usage of a non-native 256 bits uint as a counter in for-loops increases gas cost *Submitted by Dravee, also found by robee and Jujic*
- [G-82] Gas: Unused Named Returns *Submitted by Dravee*
- [G-83] Gas: Use `calldata` instead of `memory` for external functions where the function argument is read-only. *Submitted by Dravee*
- [G-84] Gas: Unnecessary checked arithmetic when no overflow/underflow possible *Submitted by Dravee, also found by defsec, sirhashalot, Jujic, WatchPug, and hyh*
- [G-85] Gas: Storage variable `IndexTemplate:pendingEnd#62` is never used and should be deleted *Submitted by Dravee*
- [G-86] Gas: Costly operations inside a loop (`IndexTemplate._adjustAlloc()`)

Submitted by Dravee

- [G-87] Public functions to external *Submitted by robee, also found by Dravee, Fitraldys, defsec, and gzeon*
- [G-88] Gas: Consider making variables that aren't updated outside the constructor as `immutable` *Submitted by Dravee, also found by Jujic, robee, TomFrenchBlockchain, 0x1f8b, loop, csanuragjain, gzeon, and defsec*
- [G-89] unnecessary checked postfix arithmetics *Submitted by egjlmn1, also found by robee, Dravee, tqts, defsec, OriDabush, Jujic, 0x0x0x, and WatchPug*
- [G-90] Gas: Contracts inheriting `InsureDAOERC20` don't need to import some dependencies *again Submitted by Dravee*
- [G-91] `CDSTemplate.sol:compensate` code optimization *Submitted by Dravee*
- [G-92] Gas: Avoid double assignment on variable *Submitted by Dravee*
- [G-93] Gas: Redundant if-statement with the for-loop condition *Submitted by Dravee*
- [G-94] Gas: Short-circuiting in an if-statement *Submitted by Dravee*
- [G-95] Gas: Use `else if` to save gas and simplify code *Submitted by Dravee, also found by p4st13r4*
- [G-96] Loss of precision and increased gas cost with double assignment on a calculation *Submitted by Dravee*
- [G-97] Shorten Error Messages to Save Gas *Submitted by 0xngndev, also found by robee, Dravee, tqts, p4st13r4, sirhashalot, Jujic, Meta0xNull, and defsec*

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.