# Nested Finance contest Findings & Analysis Report

2022-04-27

## Table of contents

# 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 audit 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 audit contest outlined in this document, C4 conducted an analysis of the Nested Finance smart contract system written in Solidity. The audit contest took place between February 10—February 12 2022.

## Wardens

26 Wardens contributed reports to the Nested Finance contest:

1. Oxliumin

2. GreyArt (hickuphh3 and itsmeSTYJ)
3. 0x1f8b
4. pauliax
5. kenzo
6. Dravee
7. robee
8. IIIIIII
9. csanuragjain
10. hyh
11. Omik
12. gzeon
13. bobi
14. rfa
15. ShippooorDAO
16. samruna
17. Tomio
18. kenta
19. WatchPug (jtp and ming)
20. m_smirnova2020
21. ye0lde
22. sirhashalot
23. defsec
24. cmichel

This contest was judged by harleythedog.

Final report assembled by liveactionllama.

## Summary

The C4 analysis yielded 5 unique MEDIUM severity vulnerabilities. Additionally, the analysis included 15 reports detailing issues with a risk rating of LOW severity or

non-critical as well as 19 reports recommending gas optimizations. All of the issues presented here are linked back to their original finding.

Notably, 0 vulnerabilities were found during this audit contest that received a risk rating in the category of HIGH severity.

## Scope

The code under review can be found within the C4 Nested Finance contest repository, and is composed of 10 smart contracts written in the Solidity programming language and includes 974 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/non-critical.

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.

## Medium Risk Findings (5)

### [M-01] Destroy can avoid the bulk of fees

*Submitted by 0xliumin*

NestedFactory.sol#L446

A user can destroy their NFTs and not pay fees on most of their assets.

### Proof of Concept

Alice has an NFT portfolio with 100 gwei dai and 100 gwei uni. Alice calls destroy on this NFT with buy token marked as dai. We would expect after this destroy step that she would pay 1 gwei dai in fees + 1 gwei uni worth of dai in fees, no matter what orders she selects.

Alice selects the following orders:

```
[{ operator: ZeroEx, token: uni, calldata: performSwap with a ve
 { operator: Flat, token: dai, calldata: transfer a very small a
```

This set of orders doesn't violate any of the require statements in the destroy function. Each order will complete successfully given the constraints here: MixinOperatorResolver.sol#L100-L101.

Fees aren't collected on the leftover amount from the callOperator step, seen here: NestedFactory.sol#L446.

This means that Alice will only pay the fees on the dai that was output from the orders (a very small amount), and the rest of the 100 gwei uni and 100 gwei dai are transferred directly back to the attacker.

### Recommended Mitigation Steps

Replace the safeTransfer with your safeTransferWithFees function. Or, if you don't want users to have to pay fees on slippage, set a maximum "slippage" amount in the safeSubmitOrder function.

**maximebrugel (Nested Finance) confirmed, but disagreed with High severity and commented:**

> Good one

> PR: Med/High Risk Fixes

harleythedog (judge) decreased severity to Medium and commented:

> The warden has described a way to avoid fees even when large amounts are being withdrawn. An avoidance of fees more closely fits the criteria of a medium severity issue:
>
> 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.
>
> This is contrary to the previous audit that awarded a fee avoidance finding as high severity (which I disagree with).

## [M-02] `NestedFactory` does not track operators properly

*Submitted by lllllll, also found by 0x1f8b, hyh, and pauliax*

NestedFactory.sol#L99-L108
MixinOperatorResolver.sol#L30-L47
NestedFactory.sol#L110-L122
MixinOperatorResolver.sol#L49-L55

`NestedFactory` extends the `MixinOperatorResolver` contract which comes from the `synthetix/MixinResolver.sol` code base where the expectation is that `isResolverCached()` returns false until `rebuildCache()` is called and the cache is fully up to date. Due to a medium issue identified in a prior contest, the `OperatorResolver.importOperators()` step was made to be atomically combined with the `NestedFactory.rebuildCache()` step. However, the atomicity was not applied everywhere and the ability to add/remove operators from the `NestedFactory` also had other cache-inconsistency issues. There are *four separate instances* of operator tracking problems in this submission.

### Impact

As with the prior issue, many core operations (such as `NestedFactory.create()` and `NestedFactory.swapTokenForTokens()`) are dependant on the assumption

that the `operatorCache` cache is synced prior to these functions being executed, but this may not necessarily be the case. Unlike the prior issue which was about updates to the resolver not getting reflected in the cache, this issue is about changes to the factory not updating the cache.

## Proof of Concept

### 1. `removeOperator()` does not call `rebuildCache()`

1. `NestedFactory.removeOperator()` is called to remove an operator
2. A user calls `NestedFactory(MixinOperatorResolver).create()` using that operator and succeedes
3. `NestedFactory.rebuildCache()` is called to rebuild cache

This flow is not aware that the cache is not in sync

```
/// @inheritdoc INestedFactory
function addOperator(bytes32 operator) external override onlyOwr
  require(operator != bytes32(""), "NF: INVALID_OPERATOR_NAME");
  bytes32[] memory operatorsCache = operators;
  for (uint256 i = 0; i < operatorsCache.length; i++) {
      require(operatorsCache[i] != operator, "NF: EXISTENT_OPERA
  }
  operators.push(operator);
  emit OperatorAdded(operator);
}
```

[NestedFactory.sol#L99-L108]

### 2. Using both `removeOperator()` and `rebuildCache()` does not prevent `create()` from using the operator

Even if `removeOperator()` calls `rebuildCache()` the function will still not work because `resolverOperatorsRequired()` only keeps track of remaining operators, and `rebuildCache()` currently has no way of knowing that an entry was removed from that array and that a corresponding entry from `operatorCache` needs to be removed too.

```
    /// @notice Rebuild the operatorCache
    function rebuildCache() external {
        bytes32[] memory requiredOperators = resolverOperatorsRequi
        bytes32 name;
        IOperatorResolver.Operator memory destination;
        // The resolver must call this function whenever it updates
        for (uint256 i = 0; i < requiredOperators.length; i++) {
            name = requiredOperators[i];
            // Note: can only be invoked once the resolver has all t
            destination = resolver.getOperator(name);
            if (destination.implementation != address(0)) {
                operatorCache[name] = destination;
            } else {
                delete operatorCache[name];
            }
            emit CacheUpdated(name, destination);
        }
    }
```

[MixinOperatorResolver.sol#L30-L47]

### 3. `addOperator()` does not call `rebuildCache()`

1. `NestedFactory.addOperator()` is called to add an operator
2. A user calls `NestedFactory(MixinOperatorResolver).create()` using that operator and fails because the operator wasn't in the `resolverOperatorsRequired()` during the last call to `rebuildCaches()`, so the operator isn't in `operatorCache`
3. `NestedFactory.rebuildCache()` is called to rebuild cache

This flow is not aware that the cache is not in sync

```
    /// @inheritdoc INestedFactory
    function removeOperator(bytes32 operator) external override onl
      uint256 operatorsLength = operators.length;
      for (uint256 i = 0; i < operatorsLength; i++) {
          if (operators[i] == operator) {
              operators[i] = operators[operatorsLength - 1];
              operators.pop();
              emit OperatorRemoved(operator);
```

```
            return;
          }
        }
        revert("NF: NON_EXISTENT_OPERATOR");
      }
```

NestedFactory.sol#L110-L122

### 4. `isResolverCached()` does not reflect the actual updated-or-not state

This function, like `removeOperator()` is not able to tell that there is an operator that needs to be removed from `resolverCache`, causing the owner not to know a call to `rebuildCache()` is required to 'remove' the operator

```
    /// @notice Check the state of operatorCache
    function isResolverCached() external view returns (bool) {
      bytes32[] memory requiredOperators = resolverOperatorsRequirec
      bytes32 name;
      IOperatorResolver.Operator memory cacheTmp;
      IOperatorResolver.Operator memory actualValue;
      for (uint256 i = 0; i < requiredOperators.length; i++) {
```

MixinOperatorResolver.sol#L49-L55

### Recommended Mitigation Steps

Add calls to `rebuildCache()` in `addOperator()` and `removeOperator()`, have `INestedFactory` also track operators that have been removed with a new array, and have `isResolverCached()` also check whether this new array is empty or not.

**maximebrugel (Nested Finance) confirmed and commented:**

> With this fix => #18
> No need to add an array of removed operators, because we are now removing the operators from the cache at the same time. Only need to call rebuildCache when adding and removing operators.

> PR: Med/High Risk Fixes

## [M-03] Undesired behavior

*Submitted by robee, also found by 0x1f8b, csanuragjain, and Dravee*

[NestedRecords.sol#L117-L131]

You push a parameter into an array of tokens without checking if it already exists. And, if at first it's added with amount 0, it can later on be pushed with a greater amount and be twice in the array. Then in all processing it will consider the first occurrence and therefore the occurrence with amount 0.

```
NestedRecords.store pushed the parameter _token
```

**maximebrugel (Nested Finance) disagreed with High severity and commented:**

> Indeed, `_amount` is not checked and may result in the loss of funds for the user... If we only look at the `store` function.

> **However**, this situation can't happen because of the ÌNestedFactory' (the only one able to call).

> The Factory is calling with this private function :

```
/// @dev Transfer tokens to the reserve, and compute the amount
/// in the records. We need to know the amount received in case
/// @param _token The token to transfer (IERC20)
/// @param _amount The amount to send to the reserve
/// @param _nftId The Token ID to store the assets
function _transferToReserveAndStore(
    IERC20 _token,
    uint256 _amount,
    uint256 _nftId
) private {
    address reserveAddr = address(reserve);
    uint256 balanceReserveBefore = _token.balanceOf(reserveAddr)
    // Send output to reserve
    _token.safeTransfer(reserveAddr, _amount);
    uint256 balanceReserveAfter = _token.balanceOf(reserveAddr);
```

```
        nestedRecords.store(_nftId, address(_token), balanceReserveA
    }
```

> Here, the `store` amount parameter can be `0` if :

- `_amount` is equal to 0. Then `balanceReserveAfter - balanceReserveBefore = 0`.
- `_amount` is not equal to 0 but the `safeTransfer` function is transferring `0` tokens (100% fees, malicious contract,...).

> We can't consider the second option, It is an external cause and we are not able to manage the exotic behaviors of ERC20s. So, when the `_amount` parameter of this function can be equal to `0` ?

> => In `submitOutOrders` :

```
    amountBought = _batchedOrders.outputToken.balanceOf(address(this

    (...)

    amountBought = _batchedOrders.outputToken.balanceOf(address(this
    feesAmount = amountBought / 100; // 1% Fee
    if (_toReserve) {
        _transferToReserveAndStore(_batchedOrders.outputToken, amour
    }
```

> **But the `ZeroExOperator` or `FlatOperator` will revert if the amount bought is `0`.**

> => In `_submitOrder`

```
    (bool success, uint256[] memory amounts) = callOperator(_order,
    require(success, "NF: OPERATOR_CALL_FAILED");
    if (_toReserve) {
        _transferToReserveAndStore(IERC20(_outputToken), amounts[0],
    }
```

> **Same, the `ZeroExOperator` or `FlatOperator` will revert if the amount bought is `0`.**

> In conclusion, we should check this parameter, but in the actual code state it can't happen (without taking into account the exotic ERC20s that we do not manage). If we add an operator that does not check the amount bought it can happen, so, maybe reducing the severity ?

> PR: [Med/High Risk Fixes]

[harleythedog (judge) decreased severity to Medium and commented]:

> Agree with sponsor, this issue doesn't exist with the current operators, so it is not currently a threat. I am going to downgrade this to medium.

## [M-04] `NestedFactory` : User can utilise accidentally sent ETH funds via `processOutputOrders()` / `processInputAndOutputOrders()`

*Submitted by GreyArt*

[NestedFactory.sol#L71]
[NestedFactory.sol#L286-L296]
[NestedFactory.sol#L370-L375]
[NestedFactory.sol#L482-L492]

Should a user accidentally send ETH to the `NestedFactory`, anyone can utilise it to their own benefit by calling `processOutputOrders()` / `processInputAndOutputOrders()`. This is possible because:

1. `receive()` has no restriction on the sender
2. `processOutputOrders()` does not check `msg.value`, and rightly so, because funds are expected to come from `reserve`.
3. `transferInputTokens()` does not handle the case where `ETH` could be specified as an address by the user for an output order.

```
if (address(_inputToken) == ETH) {
  require(address(this).balance >= _inputTokenAmount, "NF: INVAI
  weth.deposit{ value: _inputTokenAmount }();
  return (IERC20(address(weth)), _inputTokenAmount);
}
```

Hence, the attack vector is simple. Should a user accidentally send ETH to the contract, create an output `Order` with `token` being `ETH` and amount corresponding to the NestedFactory's ETH balance.

### Recommended Mitigation Steps

1. Since plain / direct `ETH` transfers are only expected to solely come from `weth` (excluding payable functions), we recommend restricting the sender to be `weth`, like how it is done in `[FeeSplitter](https://github.com/code-423n4/2022-02-nested/blob/main/contracts/FeeSplitter.sol#L101-L104)`.

   We are aware that this was raised previously here: [code-423n4/2021-11-nested-findings#188](code-423n4/2021-11-nested-findings#188) and would like to add that restricting the sender in the `receive()` function will not affect `payable` functions. From from what we see, plain ETH transfers are also not expected to come from other sources like `NestedReserve` or operators.

```
receive() external payable {
  require(msg.sender == address(weth), "NF: ETH_SENDER_NOT_WETH'
}
```

2. Check that `_fromReserve` is false in the scenario `address(_inputToken) == ETH`.

```
if (address(_inputToken) == ETH) {
  require(!_fromReserve, "NF: INVALID_INPUT_TOKEN");
  require(address(this).balance >= _inputTokenAmount, "NF: INVAI
  weth.deposit{ value: _inputTokenAmount }();
  return (IERC20(address(weth)), _inputTokenAmount);
```

```
    }
```

maximebrugel (Nested Finance) confirmed and resolved:

> PR: Med/High Risk Fixes

## [M-05] Wrong logic around `areOperatorsImported`

*Submitted by 0x1f8b, also found by 0xliumin and kenzo*

OperatorResolver.sol#L42-L43

The logic related to the `areOperatorsImported` method is incorrect and can cause an operator not to be updated because the owner thinks it is already updated, and a vulnerable or defective one can be used.

### Proof of Concept

The `operators` mapping is made up of a key `bytes32 name` and a value made up of two values: `implementation` and `selector`, both of which identify the contract and function to be called when an operator is invoked.

The `areOperatorsImported` method tries to check if the operators to check already exist, however, the check is not done correctly, since && is used instead of ||.

If the operator with name `A` and value `{implementation=0x27f8d03b3a2196956ed754badc28d73be8830a6e,selector="performSwapVulnerable"}` exists, and the owner try to check if the operator with name `A` and value `{implementation=0x27f8d03b3a2196956ed754badc28d73be8830a6e,selector="performSwapFixed"}` exists, that function will return `true`, and the owner may decide not to import it , producing unexpected errors. Because operators manage the tokens, this error can produce a token lost.

### Recommended Mitigation Steps

Change && by ||

maximebrugel (Nested Finance) confirmed and resolved:

> PR: Med/High Risk Fixes

harleythedog (judge) commented:

> Good catch, I agree with severity.

## Low Risk and Non-Critical Issues

For this contest, 15 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by warden **pauliax** received the top score from the judge.

*The following wardens also submitted reports:* Dravee, Omik, IIIIIII, kenzo, samruna, gzeon, bobi, robee, rfa, WatchPug, GreyArt, csanuragjain, 0xliumin, *and* ShippooorDAO.

### [01]

Function releaseTokens uses weth, not eth when comparing against a native asset. if the token address is weth, it unwraps and sends the native asset to the user:

```
if (address(_tokens[i]) == weth)
  IWETH(weth).withdraw(amount);
  (bool success, ) = _msgSender().call{ value: amount }("");
  require(success, "FS: ETH_TRANFER_ERROR");
```

Releasing weth token should be left as a valid option if the user prefers wrapped ERC20 eth, and I think for this native purpose there is a not used storage variable named ETH:

```
address private constant ETH = 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEee
```

Based on my assumptions, the intention was:

```
if (address(_tokens[i]) == ETH)
  ...
```

or if you do not want to implement this change, then at least remove this unused variable to save some gas. However, the issue is small, because the user can always retrieve weth by using another function named releaseTokensNoETH.

## [02]

This was mentioned in the Red4Sec audit (NFSC09), but it wasn't fixed here: OwnableProxyDelegation is Context, but still uses msg.sender, not _msgSender():

```
require(StorageSlot.getAddressSlot(_ADMIN_SLOT).value == msg.ser
```

## [03]

function rebuildCache() in MixinOperatorResolver does not delete removed operators from operatorCache. resolverOperatorsRequired return current active operators, so it will not contain removed operators, e.g. operator was removed by calling removeOperator in the factory, then rebuildCache is called, and the cache will still contain this removed operator, and it will be possible to callOperator on this operator.

## [04]

Consider introducing an upper limit for _timestamp in updateLockTimestamp, e.g. max 1 year from current block timestamp, otherwise it may be possible to accidentally lock the token forever.

## [05]

If removeFactory has this check:

```
require(supportedFactories[_factory], "OFH: NOT_SUPPORTED");
```

then I think addFactory should have an analogous check:

```
require(!supportedFactories[_factory], "OFH: ALREADY_SUPPORTED")
```

## [06]

The revert message is a bit misleading here:

```
require(assetTokensLength > 1, "NF: UNALLOWED_EMPTY_PORTFOLIO");
```

## [07]

NestedFactory has a function unlockTokens that lets admin rescue any ERC20 token. Consider also adding support for rescuing the native asset (e.g. ETH).

[maximebrugel (Nested Finance) commented](): 

> **[01] Function releaseTokens uses weth, not eth**
> We are only sending fees with ERC20 (so WETH and not ETH). In the releaseTokens tokens we are unwrapping the WETH to transfer ETH. The `nestedFactory` is wrapping ETH before sending fees.

> But we should remove this variable => `address private constant ETH = 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE;` and change comment.

> **[02] OwnableProxyDelegation is Context, but still uses msg.sender, not _msgSender()**
> Acknowledged. No meta-transaction support for this admin function.

> **[03] Function rebuildCache() in MixinOperatorResolver does not delete removed operators from operatorCache**
> Confirmed. The mitigation found is to remove from cache in the `removeOperator` function.

> **[04] Consider introducing an upper limit for _timestamp in updateLockTimestamp**

Acknowledged. We are not sure about an upper limit to set.

**[05] addFactory should have an analogous check**
Disputed. No need for a require as long as `supportedFactories[_factory] = true` does not disrupt the protocol state.

**[06] The revert message is a bit misleading here**
Disputed. I don't really know what is misleading. You can't withdraw the last token and keep an empty portfolio.

**[07] adding support for rescuing the native asset**
Acknowledged. We will fix this issue by adding a require in the `receive` function. Also, the user can't send more ETH than needed with the `_checkMsgValue` function.

**harleythedog (judge) commented:**

My personal judgements:
**[01] "Function releaseTokens uses weth"**
This is a gas optimization. Will keep it in mind when scoring **#67** [G-07]. For here, invalid.

**[02] "OwnableProxyDelegation is Context"**
Valid and very-low-critical.

**[03] "Function rebuildCache() in MixinOperatorResolver does not delete removed operators from operatorCache"**
This has been upgraded to medium severity in **#77** [M-02]. Will not contribute to QA score.

**[04] "Consider introducing an upper limit for _timestamp in updateLockTimestamp"**
I think this is a good idea. Valid and low-critical.

**[05] "addFactory should have an analogous check"**
Just a consistency suggestion, valid and non-critical.

**[06] "The revert message is a bit missleading here"**
Warden doesn't explain enough why it is misleading. Invalid.

> **[07] “Consider also adding support for rescuing the native asset”**
> Valid and low-critical.

> Now, here is the methodology I used for calculating a score for each QA report. I first assigned each submission to be either non-critical (1 point), very-low-critical (5 points) or low-critical (10 points), depending on how severe/useful the issue is. The score of a QA report is the sum of these points, divided by the maximum number of points achieved by a QA report. This maximum number was 26 points, achieved by this report **#66**.

> The number of points achieved by this report is 26 points.
> Thus the final score of this QA report is (26/26)*100 = 100.

## Gas Optimizations

For this contest, 19 reports were submitted by wardens detailing gas optimizations. The **report highlighted below** by warden **pauliax** received the top score from the judge.

*The following wardens also submitted reports:* **robee**, **Dravee**, **kenzo**, **GreyArt**, **Tomio**, **Omik**, **kenta**, **rfa**, **m_smirnova2020**, **csanuragjain**, **0x1f8b**, **ye0lde**, **sirhashalot**, **ShippooorDAO**, **gzeon**, **defsec**, **cmichel**, *and* **bobi**.

### [G-01]

Function transfer in NestedReserve is never used and can only be called by the factory (onlyFactory), so consider removing it because I think the factory uses a withdraw function from the Reserve.

Currently never used:

```
function setReserve onlyFactory
```

You can remove it to save some gas, or leave it if it was intended for future use with other factories.

### [G-02]

Functions that add or remove operators or shareholders iterate over the whole array, so you can consider using EnumerableSet to store them:

EnumerableSet.sol

## [G-03]

Could just use msg.sender and do not call an owner() function here:

```
function unlockTokens(IERC20 _token) external override onlyOwner
  ...
  _token.safeTransfer(owner(), amount);
```

## [G-04]

There are several functions that call _checkMsgValue. This function is quite expensive as it iterates over all the _batchedOrders and is only relevant when the inputToken is ETH. Later the callers will have to iterate over all the _batchedOrders again anyway, so I think this function should be refactored to significantly reduce gas. My suggestion:

Because processInputOrders and processInputAndOutputOrders both call _processInputOrders, the logic from _checkMsgValue could be moved to _processInputOrders. function create then can be refactored to re-use _processInputOrders. I see 2 discrepancies here: _fromReserve is always false when _submitInOrders is called from create (could be solved if _processInputOrders takes extra parameter), and _processInputOrders has this extra line:

```
require(nestedRecords.getAssetReserve(_nftId) == address(reserve
```

but this could be solved if you first mint the NFT and then invoke _processInputOrders from create.

## [G-05]

Function withdraw calls nestedRecords twice:

```
uint256 assetTokensLength = nestedRecords.getAssetTokensLength(_
...
address token = nestedRecords.getAssetTokens(_nftId)[_tokenInde
```

I think it could just substitute these links by first fetching all the tokens, and then calculating the length itself instead of making 2 external calls for pretty much the same data.

## [G-06]

Could use 'unchecked' maths here, as underflow is not possible:

```
if (_amountToSpend > amounts[1]) {
  IERC20(_inputToken).safeTransfer(_msgSender(), _amountToSpend
}
```

**maximebrugel (Nested Finance) commented**:

> **[G-01] "You can remove it to save some gas, or leave it if it was intended for future use with other factories"**
> Acknowledged

> **[G-02] Consider using EnumerableSet to store operators**
> Acknowledged

> **[G-03] "Could just use msg.sender and do not call an owner() function here"**
> Confirmed

> **[G-04] `_checkMsgValue` refactoring**
> Disputed
> "Could be". You need to pre-calculate the amount of ETH needed to check msg.value in a simple way.

> **[G-05] "function withdraw calls nestedRecords twice"**
> Acknowledged

**[G-06] “Could use ‘unchecked’ maths here, as underflow is not possible”**
Confirmed

PR: [Gas Optimizations Fixes]

[harleythedog (judge) commented]:

My personal judgments:

**[G-01] “function transfer in NestedReserve is never used”.**
Valid and medium-optimization.

**[G-02] “EnumerableSet to store them”.**
Valid and small-optimization.

**[G-03] “Could just use msg.sender”.**
Valid and small-optimization.

**[G-04] ” `_checkMsgValue` refactoring”.**
The idea of refactoring the reserve check to be in the combined function is valid.
Valid and small-optimization.

**[G-05] “withdraw calls nestedRecords twice”.**
Valid and small-optimization.

**[G-06] “Could use ‘unchecked’ here”.**
This was disputed in other gas reports as this already surfaced in the first contest.
To be fair, this should be marked as invalid too.
Invalid.

Also, #66 has the additional issue:
**[G-07] “remove this unused variable to save some gas (ETH)“.**
Valid and small-optimization.

Now, here is the methodology I used for calculating a score for each gas report. I first assigned each submission to be either small-optimization (1 point), medium-optimization (5 points) or large-optimization (10 points), depending on how useful the optimization is. The score of a gas report is the sum of these points, divided by

> the maximum number of points achieved by a gas report. This maximum number was 10 points, achieved by this report **#67**.

> The number of points achieved by this report is 10 points.
> Thus the final score of this gas report is (10/10)*100 = 100.

## 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.

Top