Introducing Code4rena Blue: Dedicated defense. Competitive bounties. Independent judging.
Learn more →

Ethos Reserve contest
Findings & Analysis Report

2023-05-02

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 Ethos Reserve smart contract system written in Solidity. The audit contest took place between February 16—March 7 2023.

Wardens

166 Wardens contributed reports to the Ethos Reserve contest:

1. 0x3b
2. 0x52
3. 0x6980
4. 0x73696d616f
5. 0xAgro
6. 0xBeirao
7. 0xDING99YA
8. 0xRobocop
9. 0xSmartContract
10. 0xTheC0der
11. 0xackermann
12. 0xbepresent
13. 0xhacksmithh
14. 0xmuxyz
15. 0xnev
16. 0xsomeone
17. 2997ms
18. 7siech

19. ABA
20. AkshaySrivastav
21. BRONZEDISC
22. Bauer
23. Bjorn_bug
24. Bnke0x0
25. Bobface
26. Bough
27. Breeje
28. Budaghyan
29. Co0nan
30. CodeFoxInc (thurendous and TerrierLover and retocrooman)
31. CodingNameKiki
32. DadeKuma
33. Darshan
34. DeFiHackLabs (SunSec and gbaleee and 0x4non and Aits and Rappie)
35. Deivitto
36. Franfran
37. GalloDaSballo
38. GreedyGoblin
39. Haipls
40. IceBear
41. Inspectah
42. JCN
43. Josiah
44. KIntern_NA (TrungOre and duc and Trumpero)
45. Kaysoft
46. KingNFT
47. Koolex

48. Krace
49. Lavishq
50. LethL
51. Madalad
52. MiloTruck
53. MiniGlome
54. MohammedRizwan
55. Morraez
56. Norah
57. P-384
58. PaludoX0
59. ParadOx
60. PawelK
61. Phantasmagoria
62. Praise
63. Qeew
64. RHaO-sec
65. Rageur
66. Raiders
67. RaymondFam
68. ReyAdmirado
69. Rickard
70. Rolezn
71. SaeedAlipoor01988
72. Saintcode_
73. Sathish9098
74. SleepingBugs (Deivitto and 0xLovesleep)
75. SuperRayss
76. TheSavageTeddy

77. Tomio
78. UdarTeam (ahmedov and tourist)
79. Udsen
80. V_B (Barichek and vlad_bochok)
81. Viktor_Cortess
82. abiih
83. adeolu
84. arialblack14
85. ast3ros
86. atharvasama
87. azhar
88. banky
89. bin2chen
90. brgltd
91. btk
92. c3phas
93. catellatech
94. ch0bu
95. chaduke
96. chrisdior4
97. codeislight
98. cryptonue
99. csanuragjain
100. d3e4
101. ddimitrov22
102. dec3ntraliz3d
103. decade
104. delfin454000
105. descharre

106. dharma09
107. dontonka
108. eierina
109. emmac002
110. eyexploit
111. favelanky
112. fs0c
113. gjaldon
114. hacker-dom
115. hansfriese
116. helios
117. hl_
118. hunter_w3b
119. hyh
120. imare
121. jasonxiale
122. joestakey
123. kaden
124. kodyvim
125. koxuan
126. ltyu
127. lukris02
128. luxartvinsec
129. martin
130. matrix_0wl
131. nemveer
132. oyc_109
133. pavankv
134. peakbolt

135. peanuts
136. peritoflores
137. rbserver
138. ronnyx2017
139. rvi0x
140. rvierdiiev
141. scokaf (Scoon and jauvany)
142. seeu
143. shark
144. supernova
145. tnevler
146. tonisives
147. trustindistrust
148. tsvetanovv
149. ulqiorra
150. vagrant
151. wtin
152. yamapyblack
153. yongskiws
154. zzzitron

This contest was judged by Trust.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 17 unique vulnerabilities. Of these vulnerabilities, 3 received a risk rating in the category of HIGH severity and 14 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 110 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 58 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Ethos Reserve contest repository, and is composed of 12 smart contracts written in the Solidity programming language and includes 3,221 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on 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

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (3)

[H-01] Re-balancing the vault allocation may always revert when distributing profits: resulting of a massive system DOS

Submitted by 0xBeirao, *also found by* bin2chen

updateRewardSum function call _computeRewardsPerUnitStaked with **_debtToOffset** set to 0. Meaning that the assignment L531 will revert if `lastLUSDLossError_Offset != 0` (which is likely the case) because we try to assign a negative value to an **uint**.

Impact

[_rebalance()](#) will be definitely DOS if the profit is greater than the [yieldClainThreshold](#) ⇒ [vars.profit != 0](#).

Because they call **_rebalance()** all these functions will be DOS :

In `BorrowerOperations` 100% DOS

- openTrove
- closeTrove
- _adjustTrove
- addColl, withdrawColl
- withdrawLUSD, repayLUSD

In `TroveManager` 80% DOS

- liquidateTroves
- batchLiquidateTroves
- redeemCloseTrove

Proof of Concept

Context : the vault has compound enough profit to withdraw. ([here](#))

Alice initiates a trove liquidation. [offset()](#) in `StabilityPool` is called to cancels out the trove debt against the LUSD contained in the Stability Pool.

A floor division errors occur so now [lastLUSDLossError_Offset](#) is not null.

Now, every time [_rebalance()](#) is called the transaction will revert.

Recommended Mitigation

In StabilityPool.sol#L504-L544, just skip the floor division errors calculation if `_debtToOffset == 0`

```
    if(_debtToOffset != 0){
            [StabilityPool.sol#L526-L538](https://github.com/code-42
    }
```

tess3rac7 (Ethos Reserve) confirmed

[H-02] User can lose up to whole stake on vault withdrawal when there are funds locked in the strategy

Submitted by hyh, *also found by* hansfriese, koxuan, Koolex, Parad0x, jasonxiale, *and* chaduke

ReaperVaultV2's `withdrawMaxLoss` isn't honoured when there are any locked funds in the strategy. Locked funds mean that there is a gap between requested and returned amount other than the loss reported. This is valid behavior of a strategy, but in this case realized loss is miscalculated in _withdraw() and a withdrawing user will receive less funds, while having all the shares burned.

Impact

Users can lose up to the whole asset amount due as all their requested shares can be burned, while only available amount be transferred to them. This amount can be arbitrary low.

The behaviour is not controlled by `withdrawMaxLoss` limit and is conditional only on a strategy having some funds locked (i.e. strategy experiencing liquidity squeeze).

Proof of Concept

`_withdraw()` resets `value` to be `token.balanceOf(address(this))` when the balance isn't enough for withdrawal:

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-

Vault/contracts/ReaperVaultV2.sol#L357-L412

```
    // Internal helper function to burn {_shares} of vault share
    // and return corresponding assets to {_receiver}. Returns t
    function _withdraw(
        uint256 _shares,
        address _receiver,
        address _owner
    ) internal nonReentrant returns (uint256 value) {
        ...

            vaultBalance = token.balanceOf(address(this));
            if (value > vaultBalance) {
                value = vaultBalance;
            }

            require(
                totalLoss <= ((value + totalLoss) * withdrawMaxI
                "Withdraw loss exceeds slippage"
            );
        }

        token.safeTransfer(_receiver, value);
        emit Withdraw(msg.sender, _receiver, _owner, value, _sha
    }
```

Each strategy can return less than `requested - loss` as some funds can be temporary frozen:

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/abstract/ReaperBaseStrategyv4.sol#L90-L103

```
    /**
     * @dev Withdraws funds and sends them back to the vault. Ca
     *      be called by the vault. _amount must be valid and se
     *      is deducted up-front.
     */
    function withdraw(uint256 _amount) external override returns
        require(msg.sender == vault, "Only vault can withdraw");
        require(_amount != 0, "Amount cannot be zero");
        require(_amount <= balanceOf(), "Ammount must be less th
        uint256 amountFreed = 0;
        (amountFreed, loss) = _liquidatePosition(_amount);
        IERC20Upgradeable(want).safeTransfer(vault, amountFreed)
    }
```

The invariant there is `liquidatedAmount + loss <= _amountNeeded`, so `liquidatedAmount + loss < _amountNeeded` is a valid state (due to the funds locked):

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/abstract/ReaperBaseStrategyv4.sol#L230-L243

```
    /**
     * Liquidate up to `_amountNeeded` of `want` of this strateg
     * irregardless of slippage. Any excess will be re-invested
     * This function should return the amount of `want` tokens m
     * liquidation. If there is a difference between them, `loss
     * difference is due to a realized loss, or if there is some
     * (e.g. locked funds) where the amount made available is le
     *
     * NOTE: The invariant `liquidatedAmount + loss <= _amountNe
     */
    function _liquidatePosition(uint256 _amountNeeded)
        internal
        virtual
        returns (uint256 liquidatedAmount, uint256 loss);
```

`_liquidatePosition()` is called in strategy withdraw():

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/abstract/ReaperBaseStrategyv4.sol#L90-L103

```
    /**
     * @dev Withdraws funds and sends them back to the vault. Ca
     *      be called by the vault. _amount must be valid and se
     *      is deducted up-front.
     */
    function withdraw(uint256 _amount) external override returns
        require(msg.sender == vault, "Only vault can withdraw");
        require(_amount != 0, "Amount cannot be zero");
        require(_amount <= balanceOf(), "Ammount must be less th

        uint256 amountFreed = 0;
        (amountFreed, loss) = _liquidatePosition(_amount);
        IERC20Upgradeable(want).safeTransfer(vault, amountFreed)
    }
```

This way there can be `lockedAmount = _amountNeeded - (liquidatedAmount + loss) >= 0`, which is neither a loss, nor withdraw-able at the moment.

As ReaperVaultV2's `_withdraw()` updates `value` per `if (value > vaultBalance) {value = vaultBalance;}`, the following `totalLoss <= ((value + totalLoss) * withdrawMaxLoss) / PERCENT_DIVISOR` check do not control for the real loss and allows user to lose up to the whole amount due as `_withdraw()` first burns the full amount of the `_shares` requested and this total loss check for the *rebased* `value` is the only guard in place:

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperVaultV2.sol#L359-L412

```
    function _withdraw(
        uint256 _shares,
        address _receiver,
        address _owner
    ) internal nonReentrant returns (uint256 value) {
        require(_shares != 0, "Invalid amount");
        value = (_freeFunds() * _shares) / totalSupply();
        _burn(_owner, _shares);

        if (value > token.balanceOf(address(this))) {
            ...

            vaultBalance = token.balanceOf(address(this));
            if (value > vaultBalance) {
                value = vaultBalance;
            }
            require(
                totalLoss <= ((value + totalLoss) * withdrawMaxI
                "Withdraw loss exceeds slippage"
            );
        }

        token.safeTransfer(_receiver, value);
        emit Withdraw(msg.sender, _receiver, _owner, value, _sha
    }
```

Suppose there is only one strategy and `90` of the `100` tokens requested is locked at the moment, and there is no loss, just a temporal liquidity squeeze. Say there is no tokens on the vault balance before strategy withdrawal.

ReaperBaseStrategyv4's `withdraw()` will transfer `10`, report `0` loss, `0 = totalLoss <= ((value + totalLoss) * withdrawMaxLoss) / PERCENT_DIVISOR = (10 + 0) * withdrawMaxLoss / PERCENT_DIVISOR` check will be satisfied for any viable `withdrawMaxLoss` setting.

Bob the withdrawing user will receive `10` tokens and have `100` tokens worth of the shares burned.

Recommended Mitigation Steps

Consider rewriting the controlling logic so the check be based on initial value:

Now:

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperVaultV2.sol#L399-L407

```
            vaultBalance = token.balanceOf(address(this));
            if (value > vaultBalance) {
                value = vaultBalance;
            }

            require(
                totalLoss <= ((value + totalLoss) * withdrawMaxI
                "Withdraw loss exceeds slippage"
            );
```

To be, as an example, if treat the loss attributed to the current user only as they have requested the withdrawal:

```
            require(
                totalLoss <= (value * withdrawMaxLoss) / PERCENT
                "Withdraw loss exceeds slippage"
            );

            value -= totalLoss;

            vaultBalance = token.balanceOf(address(this));
            require(
                value <= vaultBalance,
                "Not enough funds"
            );
```

Also, `shares` can be updated according to the real value obtained as it is done in yearn:

https://github.com/yearn/yearn-vaults/blob/master/contracts/Vault.vy#L1147-L1151

```
        if value > vault_balance:
            value = vault_balance
            # NOTE: Burn # of shares that corresponds to what Va
            #       including the losses that were incurred abov
            shares = self._sharesForAmount(value + totalLoss)
```

tess3rac7 (Ethos Reserve) confirmed via duplicate issue `#723`

[H-03] Rewards will be locked in LQTYStaking Contract

Submitted by gjaldon, also found by 0xBeirao, hyh, and 0xRobocop

https://github.com/code-423n4/2023-02-ethos/blob/main/Ethos-Core/contracts/LQTY/LQTYStaking.sol#L181-L183
https://github.com/code-423n4/2023-02-ethos/blob/main/Ethos-Core/contracts/RedemptionHelper.sol#L191-L197
https://github.com/code-423n4/2023-02-ethos/blob/main/Ethos-Core/contracts/ActivePool.sol#L296-L300

The state variable `F_Collateral` in the LQTYStaking contract is used to keep track of rewards for each of the collateral types used in the protocol. Every time the LQTYStaking contract is sent collateral assets for rewards by the ActivePool or the RedemptionHelper, `LQTYStaking.increaseF_Collateral` is called to record the rewards that are to be distributed to stakers.

However, if the state variable `totalLQTYStaked` is large enough in the LQTYStaking contract, zero rewards will be distributed to stakers even though LQTYStaking received assets. This issue is exarcebated when using WBTC as collateral due to its low number of decimals.

For example, given the following:

1. `totalLQTYStaked` = 1e25; LQTY/OATH token has 18 decimals; this means that a total of 10million LQTY has been staked
2. A redemption rate of 0.5% was applied on a redemption of 10e8 WBTC. This leads to a redemption fee of 5e6 WBTC that is sent to the LQTYStaking contract. This happens in this code.
3. Given the above, RedemptionHelper calls `LQTYStaking.increaseF_Collateral(WBTCaddress, 5e6)`

The issue is in this line in `increaseF_Collateral`:

```
    if (totalLQTYStaked > 0) {collFeePerLQTYStaked = _collFee.mul(DE
```

`_collFee` = 5e6; `DECIMAL_PRECISION` = 1e18; `totalLQTYStaked` = 1e25

If we substitute the variables with the actual values and represent the code in math, it looks like:

```
(5e6 * 1e18) / 1e25 = 5e24 / 1e25 = 0.5
```

Since the result of that math is a value less than 1 and in Solidity/EVM we only deal with integers and division rounds down, we get 0 as a result. That means the below code will only add `0` to `F_Collateral`:

```
F_Collateral[_collateral] = F_Collateral[_collateral].add(collFe
```

So even though LQTYStaking received 5e6 WBTC in redemption fee, that fee will never be distributed to stakers and will remain forever locked in the LQTYStaking contract. The minimum amount of redemption fee that is needed for the reward to be recognized and distributed to stakers is 1e7 WBTC. That means at least 0.1 BTC in collateral fee is needed for the rewards to be distributed when there is 1Million total LQTY staked.

Impact

This leads to loss of significant rewards for stakers. These collateral assets that are not distributed as rewards will remain forever locked in LQTYStaking.

If 1e25 LQTY is staked in LQTYStaking (10M LQTY), at least 1e7 (0.1) WBTC in redemption fee must be sent by the RedemptionHelper for that WBTC to be sent as rewards to the stakers. That means only redemptions of 20e8 (20) WBTC and more will lead to redemption fees high enough to be distributed as rewards to stakers. Redemption of 20e8 WBTC will rarely happen, so it’s likely that majority of rewards will be forever locked since most redemptions will be less than that.

Given the above, if only 3% of redemptions have amounts of 20e8 WBTC or greater, then 97% of redemptions will have their fees forever locked in the contract. The greater the amount of LQTY Staked, the higher the amount needed for the fees to be recorded.

Proof of Concept

First, comment out this line in `increaseF_Collateral` to disable the access control. This allows us to write a more concise POC. It is fine since the issue has nothing to do with access control.

Add the following test case to the `Ethos-Core/test/LQTYStakingFeeRewardsTest.js` file after the `beforeEach` clause:

```
it('does not increase F collateral even with large amount of c
  await stakingToken.mint(A, dec(10_000_000, 18))
  await stakingToken.approve(lqtyStaking.address, dec(10_000_(
  await lqtyStaking.stake(dec(10_000_000, 18), {from: A})

  const wbtc = collaterals[1].address
  const oldWBTC_FCollateral = await lqtyStaking.F_Collateral(w

  // .09 WBTC in redemption/collateral fee will not be distrik
  await lqtyStaking.increaseF_Collateral(wbtc, dec(9, 6))
  assert.isTrue(oldWBTC_FCollateral.eq(await lqtyStaking.F_Col

  // at least 0.1 WBTC in redemption/collateral fee is needed
  await lqtyStaking.increaseF_Collateral(wbtc, dec(1, 7))
  assert.isTrue(oldWBTC_FCollateral.lt(await lqtyStaking.F_Col
})
```

The test can then be run with the following command:

```
$ npx hardhat test --grep "does not increase F collateral even w
```

Recommended Mitigation

One way to address this issue is to use the same error-recording logic found in the `_computeLQTYPerUnitStaked` logic that looks like:

```
        uint LQTYNumerator = _LQTYIssuance.mul(DECIMAL_PRECISION

        uint LQTYPerUnitStaked = LQTYNumerator.div(_totalLUSDDep
        lastLQTYError = LQTYNumerator.sub(LQTYPerUnitStaked.mul(
```

The `lastLQTYError` state variable stores the LQTY issuance that was not distributed since they were just rounded off. The same approach can be used in `increaseF_Collateral`.

tess3rac7 (Ethos Reserve) confirmed

Medium Risk Findings (14)

[M-01] Low data feed frequency from Tellor makes your protocol vulnerable to flash loan attacks

Submitted by peritoflores

An attacker can stale Tellor Oracle for several hours cheaply and perform a flash loan attack to profit.

Proof of Concept

To explain this issue I will first compare Chainlink to Tellor.

Most ERC-20 tokens are in general much more volatile than ETH and BTC. In Chainlink, there are triggers of 0,5% for BTC and ETH and 1% for other assets. This is to ensure that you are cutting error by those values.

Tellor, on the other hand, is an *optimistic oracle*. Stakers use the oracle system to put data on chain `submitValue(..)` that are directly shown in the oracle. The security lies the fact that data consumers should wait some **dispute windows** in order to give time to others to dispute data and remove incorrect or malicious data.

This is what happened in a Liquity bug found last year, they were reading instant data.[2]

Being explained this and back to your code you have essentially two bugs

First bug: Default disputetime = 20 minutes

In `TellorCaller.sol` you have the following statement

```
 (bytes memory data, uint256 timestamp) = getDataBefore(_queryId,
block.timestamp - 20 minutes)
```

Maybe you are using 20 minutes because this is the default value in Tellor documentation. However, in Liquity they are using 15 minutes for ETH because they say that have been made an analysis of ETH volatility behaviour.[2]

Basically there is a tradeoff between the volatility of an asset and the dispute time. More time is safer to have time to dipute but more likely to read a so old value. Less dispute time you have less error but no time to dispute can put you at risk of reading a manipulated value.

In your case, you are using more volatile assets so in theory, if Liquity analysis is correct, you should be using less time for ERC20 assets.

Of course this requires a deeper analysis but I am not doing it because the second bug makes this unnecessary as it has a higher impact.

Second bug: Data feed frequency in Tellor is very low so it is cheap to break

I will briefly explain some Tellor security designs.

Tellor bases his security design in an exponential cost to dispute. They have a several-round voting to dispute a single value but we are interested in the *Cost of Stalling* (CoS) the System.

To stale the system we need to dipute every single value for a given period, for a given asset.

According to whitepaper cost starts at `baseFee` and increase with the following formula

$$disputeFee_{id,t,r>1} = disputeFee_i \times 2^{disputeRounds_{id,t}-1}$$

Where

$disputeFee_i$ is the initial dispute fee (`baseFee`)

$disputeRounds_{id}$ is the number of disputes open for a specific ID

In Ethereum, there is a block every 15 seconds so stalling the system for 8 minutes (32 blocks) will cost an attacker around `2^32 * 10 TRB` = 687 Billons of dollars! ... (10TRB = 160USD). Not bad at all.

Tellor team has similar values in different docs around internet.

However, this is nice if we **always assume that one data is sent every block (an ideal system).**

And here is where the real nightmare comes. Current frequency for data in Tellor is very low, **that you are reading data once an hour or less!!**.

Even worse for Optimism and Polygon `basedisputeFee` is only 1TRB . (vs 10 TRB in Ethereum)

This design was thought considering that these chains are faster so if you data is sent every block then breaking the system would be prohibitively expensive. Again, security depends on the frequency of data.

In our real word, Tellor is producing data in Optimism as low as in Ethereum so in the end it is 10 times cheaper to break.

Cost to Stale ETH/USD pair in Optimism

Lets calculate the CoS ETH/USD pair for 4 hours.

Watching this Tellor contracts we can get that baseFee is 1TRB

https://optimistic.etherscan.io/address/0x46038969d7dc0b17bc72137d07b4ede43859da45#readContract ==> `getDataFee()` = 1TRB

Now, read data in a four hour range using the function.

`getMultipleValuesBefore()`

Parameters passed

`queryId = 0x83a7f3d48786ac2667503a61e8c415438ed2922eb86a2906e4ee66d9a2ce4992` (ID for asking ETH/USD pair value)

`timestamp = 1678147200` (7 march 2023 at 0:00)

`_max age = 14400` (4 hours earlier = 14400 seconds)

`_maxCount = 1000` (doesn't really matter)

We get only 4 values with the following timestamps [1678135628,1678139237,1678142833,1678146437]

The *CoS* Tellor ETH/USD pair for these four hours would have been

```
1TRB + 2TRB + 4TRB + 8 TRB = 15TRB
```

```
15TRB * 16USD = 240 USD
```

This means that for a little 240 bucks you can Stale 4 hours the Oracle which is not acceptable at all as I will show you an attacking scenario.

Ethereum has moved only 1%, not so critical this time, however volatille ERC20 used as a collateral can have much bigger changes.

You can query more data with different timestamps

Attacking Scenario: Flash Loan to profit

Steps:

1. Write a contract that checks if Chainlink is working
2. Meanwhile a second script that tracks values for all your collateral assets
3. When Chainlink is broken do
4. Stall all your collateral data from Tellor using `dispute`. (10 collateral for `$2500`)

5. Suppose that you see an increase of 10% one of the colaterral call it ABC and ETH not moving so much
6. Ask for a Flash Loan ETH in Uniswap
7. Mint LUSD for ETH at Ethos
8. Redeem LUSD for collateral ABC. You get a 10% discount because Oracle is staled 4 hours ago
9. Exchange LUSD for ETHEREUM in Uniswap.
10. Return ETH to the flash loan plus interest
11. Enjoy!

Note that this attack can be improved if you perform the loan on a falling collateral to mint more LUSD.

The only level of protection you have is the fact that Chainlink is working, in Liquity it is more difficult because it should be an important change of ETH value in those 4 hours.

Recommended

There are two solutions in my opinion

Solution 1: Tellor tip mechanism

Tellor whitepaper:

"Parties who wish to build reporter support for their query should follow best practices when selecting data for their query (publish data specification on github, promote/ educate in the community), but will also need to tip a higher amount to incentivize activity"

This means that in order to use data safely you need to pay to be sure that frequency is secure taking into account the impact of the volatility and the time to dispute.

I didn't mention earlier but the cost to dispute is exponential but capped by the staking amount of the reporter, so no real billons of dollars in fact.

Here is the documentation how you can fund for a feed
https://docs.tellor.io/tellor/getting-data/funding-a-feed

Solution 2: Do not use Tellor

Unlike Liquity, you need to fund several feeds so I don't know if this is cost effective but you have options to fund fees only when Chainlink is broken but you need to investigate on that.

In any case you have a function to set the oracle that I am reporting as medium so no sure if you need to use two oracles.

References

1. Tellor White Paper https://tellor.io/whitepaper/
2. Liquity Tellor issue 2022 https://www.liquity.org/blog/tellor-issue-and-fix

c4-sponsor labeled sponsor disputed

Trust (judge) commented:

> This submission is interesting, but *may* be invalid. The cost of stalling calculation assumes that attacker would only need to stall 4 price updates in 4 hours, however once stalling starts it would make sense for new price reports to appear very quickly, as soon as the next block. In other words, the sample warden has looked at only contains 4 updates because they were all legit.

> I have not verified this reasoning, but ask for sponsor to take a look and give their thoughts.

tess3rac7 (Ethos Reserve) disputed and commented:

> It relies on **ALL** of the following to be true:

- chainlink broken
- tellor stalled for > 4 hours somehow by a malicious attacker
- no other tellor reporter realizing/recognizing this
- sharp movement in one collateral price during that timeframe

- not much movement in another collateral price during that timeframe

> Probability of all of the above happening together almost negligible. Moreover, it seems the report doesn't take into account:

- Redemption fee, which could be a very large % if the attacker is looking to drain the system

> Also forwarded this to our contacts at Tellor and this was their take:

> So for the first one, it's fair. A lot of times people have trouble waiting the 20 minutes and there always is a risk that the price will change drastically in those 20 minutes, but unfortunately it's just something that happens when you deal with decentralized systems (e.g. exchanges wait several blocks for confirmation, Maker waits an hour before updating its oracle). And not to mention, the price can move very drastically even in the 12 second block time of Ethereum, so if the goal is to never have a stale price, it's literally impossible. You just need to design a system that can slow down and won't break if this happens (which Liquity did and I'm presuming you guys did something similar)

> For the second "bug", the main flaw comes in the presumption of no outside actors looking to save the system or benefit from the attack. When a good value is disputed on tellor, this is actually a profitable opportunity for any reporter. Assuming the voting mechanism is not broken (as in the analysis), anyone who simply submits a good value will (after 4 rounds), double their TRB in 2 days. (the voting period). This means that for 4 hours, you would need all of the reporters to not realize this. This is a similar assumption as saying you could throw a uniswap pool and expect no one to see an arbitrage opportunity. The second false assumption is that the current optimism report rate is how often the ETH/USD feed will update forever. This is just wrong. There are few reports on Optimism because there are no tips on Optimism and no one has even told any reporters (or the team) that they are live and would like more reports. For relatively cheap, you can easily have several reports an hour, and even more if you'd like to pay for it. Honestly however, you should probably just keep it on a 4 hour pace or a something that looks at the price change so as you aren't over-paying or updating a same value.

> Additionally, there are measures that you as a team could take to dissuade an attacker. For one, you could move the 4 hour stale period longer (no reason 4

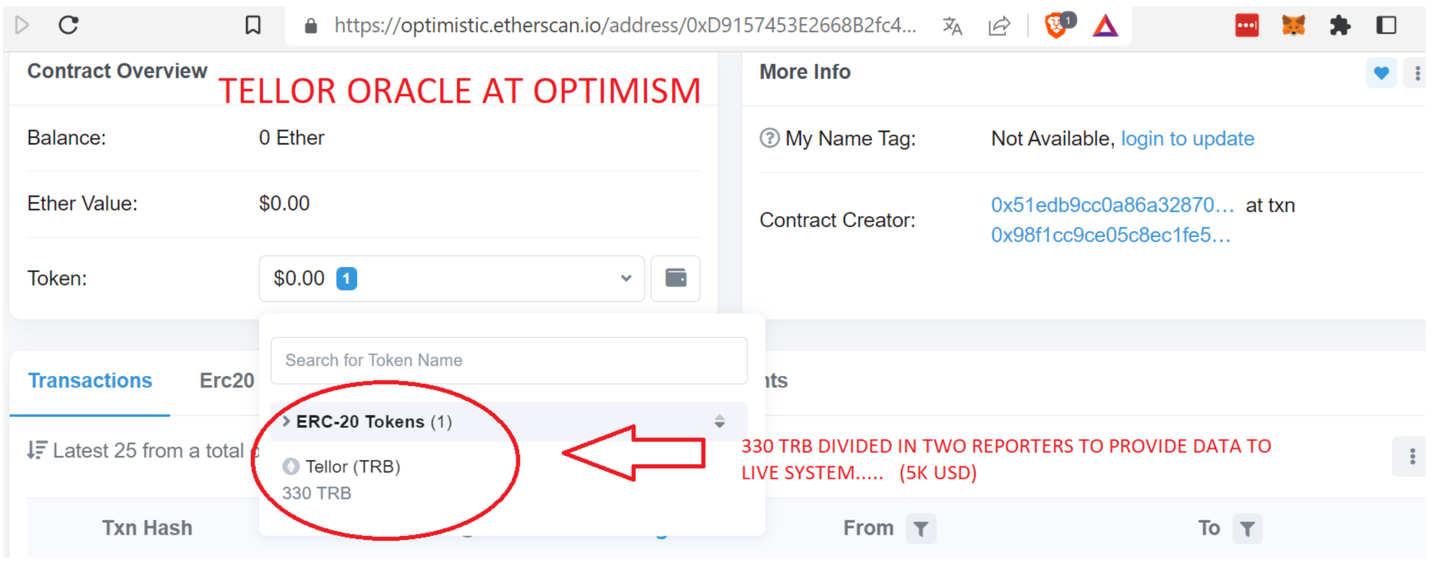
> hours is a golden number). Second, you can stake reporters and report yourself. If you have a large number of stakes ready to report on Optimism, you could simply listen for disputes and act as a reporter of last resort while you alert the other reporters to come and join the fun. Simply having this amount publicly known or ready would probably be enough to raise the costs to a high enough to prevent any attack.
> Based on all of the above, I'm leaning towards "not an issue."

peritoflores (warden) commented:

> Hi Team, sorry for that but it is either I am so wrong or you could be drained. First, I have explained @Trust that reporters cannot just send reports every block after being disputed. This depends of how much they have staked.
>
> **Notes:**
>
> Once a value is submitted, the reporter is then locked from submitting again for the reporterLock time period (usually 12 hours) divided by the number of full stakes. If the stake amount is 10 TRB and a reporter has 60 TRB staked, for example, that reporter can submit once every two hours.
>
> For all reported values, the `_queryId` must be the keccak256 hash of the `_queryData`. For information on how to get the queryId or how to parse the `_queryData`, see the Creating a Query section.

> Secondly, there are only two reporters to protect the real system with a total staked valued at 5kUSD

> ## TELLOR ORACLE AT OPTIMISM


https://optimistic.etherscan.io/address/0xD9157453E2668B2fc4...

TELLOR ORACLE AT OPTIMISM

Contract Overview	
Balance:	0 Ether
Ether Value:	$0.00
Token:	$0.00 1

More Info	
My Name Tag:	Not Available, login to update
Contract Creator:	0x51edb9cc0a86a32870... at txn 0x98f1cc9ce05c8ec1fe5...

Search for Token Name

ERC-20 Tokens (1)

Tellor (TRB)
330 TRB

330 TRB DIVIDED IN TWO REPORTERS TO PROVIDE DATA TO LIVE SYSTEM..... (5K USD)

> **Tellor reporters are**
> 0x50a86759d495ecfa7c301071d6b0bdd4bd664ab0 ---> 200 trb locked
> 0xaac7da260fb6d047314e213f672b7d3d9503a1f7 ---> 130 trb locked

> Unless people are watching off-chain to bridge TRB to optimism, dispute, stake and then submit a new value it is possible to take the oracle with only 5k or less. After disputing is not mandatory to submit a value.

> **About Tellor answer**
> I agree with almost all they said. The dispute is profitable even to double your money, but I am just interested in stale the system. The question is that if it is possible to get a correct value on time.
> What they explained about tips, of course I agree, in fact it is the solution that I proposed.

> **About your words**
> `chainlink broken`
>
> ---> This is the only condition

> `tellor stalled for > 4 hours somehow by a malicious attacker --->`
> This is just an example can be more or even less time.

> `no other tellor reporter realizing/recognizing this -->`

> As I explained there are only 2 reporters in the live system to protect you

> `sharp movement in one collateral price during that timeframe`
> `not much movement in another collateral price during that timeframe`

> That happened last weekend when USDC was falling down

> On March 11 From 1:00 AM to 5AM USDC lost almost 15% of its value.
> However other crypto where not moving.

> Note that uniswap fees are only 0,3% plus gas cost.

> **An improved flash loan attack to drain collateral**

> Note that this attack is devastator because it is like an incredibly profitable arbitrage that can be performed in every block.

> While in arbitrage oportunities prices got balanced after the swap.

> Here the attacker can still hold the oracle and repeat the attack.

Trust (judge) decreased severity to Medium and commented:

1. Ethos relies on Tellor and plenty of logic faciliates the use of both CL/Tellor oracles, so the assumption of Chainlink downtime is definitely in-scope for medium severity.
2. Having heard both sides, it seems the required attack indeed does not require *unreasonble* amount of effort by an attacker (esp. regarding two live reporters).
3. Given the execution of the attack is possible, likelihood of it generating big profits at protocol's expense is high.

> For these reasons, medium severity is most appropriate here.

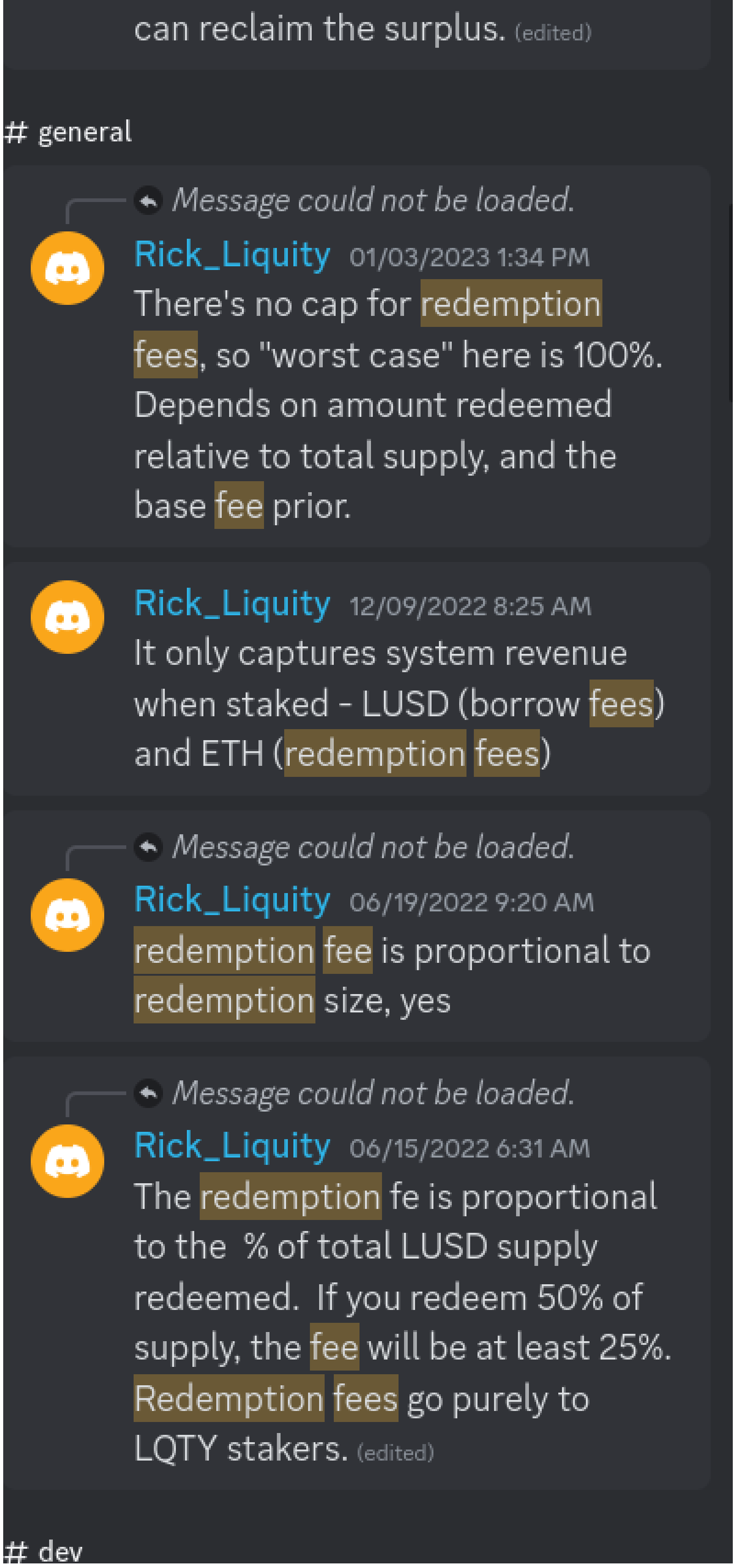
tess3rac7 (Ethos Reserve) commented:

> I'm still unclear as to how we can overlook redemption fees and claim that this will be profitable for an attacker looking to drain collateral.

> Redemption fees scales with:

- frequency of redemption, please see https://www.liquity.org/blog/on-price-stability-of-liquity
- amount being redeemed (proportional to the debt size of the market), please see https://github.com/code-423n4/2023-02-ethos/blob/main/Ethos-Core/contracts/TroveManager.sol#L1411

> If the attacker decides to redeem in one large TX, the fee will be very high, not nearly enough to offset the hypothetical 10% gain in the warden's example. Here are some discord messages of Liquity's dev explaining:


can reclaim the surplus. (edited)

general

Message could not be loaded.

Rick_Liquity 01/03/2023 1:34 PM

There's no cap for redemption fees, so "worst case" here is 100%. Depends on amount redeemed relative to total supply, and the base fee prior.

Rick_Liquity 12/09/2022 8:25 AM

It only captures system revenue when staked - LUSD (borrow fees) and ETH (redemption fees)

Message could not be loaded.

Rick_Liquity 06/19/2022 9:20 AM

redemption fee is proportional to redemption size, yes

Message could not be loaded.

Rick_Liquity 06/15/2022 6:31 AM

The redemption fe is proportional to the % of total LUSD supply redeemed. If you redeem 50% of supply, the fee will be at least 25%. Redemption fees go purely to LQTY stakers. (edited)

dev

> If the attacker decides to redeem in smaller successive TXs, the base rate would keep increasing as explained in the blog post linked above.

[M-02] `_harvestCore()` roi calculation error

Submitted by bin2chen, also found by rbserver

_harvestCore() roi calculation error,may double

Proof of Concept

The _harvestCore() will calculate the roi and repayment values.

The implementation code is as follows:

```
    function _harvestCore(uint256 _debt) internal override retur
        _claimRewards();
        uint256 numSteps = steps.length;
        for (uint256 i = 0; i < numSteps; i = i.uncheckedInc())
            address[2] storage step = steps[i];
            IERC20Upgradeable startToken = IERC20Upgradeable(ste
            uint256 amount = startToken.balanceOf(address(this))
            if (amount == 0) {
                continue;
            }
            _swapVelo(step[0], step[1], amount, VELO_ROUTER);
        }

        uint256 allocated = IVault(vault).strategies(address(thi
        uint256 totalAssets = balanceOf();
        uint256 toFree = _debt;

        if (totalAssets > allocated) {
            uint256 profit = totalAssets - allocated;
            toFree += profit;
            roi = int256(profit);
        } else if (totalAssets < allocated) {
            roi = -int256(allocated - totalAssets);
        }

        (uint256 amountFreed, uint256 loss) = _liquidatePositior
        repayment = MathUpgradeable.min(_debt, amountFreed);
        roi -= int256(loss);//<------这个地方可能会导致重复
    }
```

The last line may cause double counting of losses

For example, the current:

```
vault.allocated = 9
```

```
vault.strategy.allocBPS = 9000
```

```
strategy.totalAssets = 9
```

Suppose that after some time, strategy loses 2, then:

`strategy.totalAssets = 9 - 2 = 7`

Also the administrator sets `vault.strategy.allocBPS = 0`

This executes harvest()->_harvestCore(9) to get

`roi = 4`

`repayment = 7`

The actual loss of 2, but roi = 4 (double), test code as follows:

add to test/starter-test.js 'Vault Tests'

```
    it.only('test_roi', async function () {
      const {vault, strategy, wantHolder, strategist} = await lo
      const depositAmount = toWantUnit('10');
      await vault.connect(wantHolder)['deposit(uint256)'](deposi
      await strategy.harvest();

      const balanceOf = await strategy.balanceOf();
      console.log(`strategy balanceOf: ${balanceOf}`);
      // allocated = 9
      // 1.loss 2, left 7
      await strategy.lossFortest(toWantUnit('2'));
      // 2.modify bps=>0
      await vault.connect(strategist).updateStrategyAllocBPS(str
      // 3.so debt = 9
      await strategy.harvest();

      const {allocated, losses, allocBPS} = await vault.strategi
      console.log(`losses: ${losses}`);
      console.log(`allocated: ${allocated}`);
      console.log(`allocBPS: ${allocBPS}`);
    });
```

add to ReaperStrategyGranarySupplyOnly.sol

```
    function lossFortest(uint256 amout) external{
        ILendingPool(ADDRESSES_PROVIDER.getLendingPool()).withdr
    }
```

```
$ npx hardhat test test/starter-test.js

  Vaults
    Vault Tests
strategy balanceOf: 900000000
losses: 400000000     <--------will double
allocated: 0
allocBPS: 0
```

The last vault's allocated is correct, but the loss is wrong.
Statistics and bpsChange of _reportLoss() will be wrong.

Recommended Mitigation Steps

remove `roi -= int256(loss);`

tess3rac7 (Ethos Reserve) disagreed with severity and commented:

> Recommend low priority since it's an edge case that would only affect reporting data.

Trust (judge) commented:

> Reporting data is handled by the vault at:
> `debt = IVault(vault).report(roi, repayment);`
> We cannot rule out damage that may occur due to miscalculations on report / debt, so medium severity seems appropriate.

tess3rac7 (Ethos Reserve) confirmed

[M-03] `ReaperBaseStrategyv4.harvest()` might revert in an emergency.

Submitted by hansfriese

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/abstract/ReaperBaseStrategyv4.sol#L109 https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperStrategyGranarySupplyOnly.sol#L200

`ReaperBaseStrategyv4.harvest()` might revert in an emergency if there is no position on the lending pool.

As a result, the funds might be locked inside the strategy.

Proof of Concept

The main problem is that Aave lending pool doesn't allow 0 withdrawals.

```
    function validateWithdraw(
      address reserveAddress,
      uint256 amount,
      uint256 userBalance,
      mapping(address => DataTypes.ReserveData) storage reservesDa
      DataTypes.UserConfigurationMap storage userConfig,
      mapping(uint256 => address) storage reserves,
      uint256 reservesCount,
      address oracle
    ) external view {
      require(amount != 0, Errors.VL_INVALID_AMOUNT);
```

So the below scenario would be possible.

1. After depositing and withdrawing from the Aave lending pool, the current position is 0 and the strategy is in debt.
2. It's possible that the strategy has some want balance in the contract but no position on the lending pool. It's because `_adjustPosition()` remains the debt during reinvesting and also, there is an `authorizedWithdrawUnderlying()` for `STRATEGIST` to withdraw from the lending pool.
3. If the strategy is in an emergency, `harvest()` tries to liquidate all positions(=0 actually) and it will revert because of 0 withdrawal from Aave.
4. Also, `withdraw()` will revert at L98 as the strategy is in the debt.

As a result, the funds might be locked inside the strategy unless the `emergency` mode is canceled.

Recommended Mitigation Steps

We should check 0 withdrawal in `_withdrawUnderlying()`.

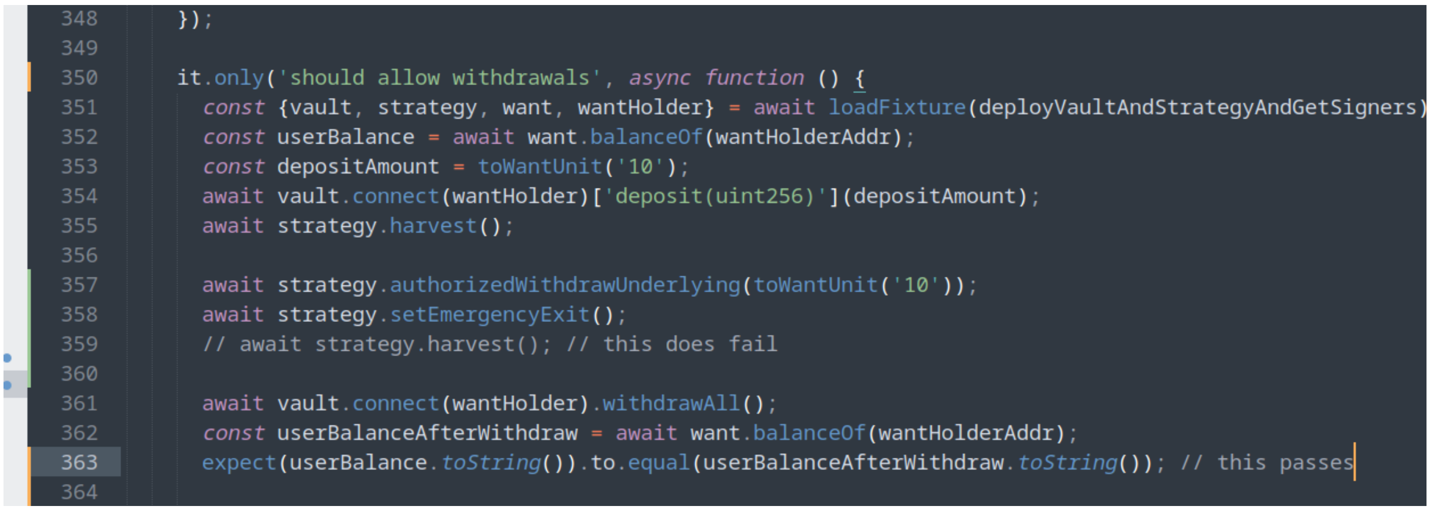
```
function _withdrawUnderlying(uint256 _withdrawAmount) intern
    uint256 withdrawable = balanceOfPool();
    _withdrawAmount = MathUpgradeable.min(_withdrawAmount, w

    if(_withdrawAmount != 0) {
        ILendingPool(ADDRESSES_PROVIDER.getLendingPool()).wi
    }
}
```

Trust (judge) commented:

> Very interesting edge case.

tess3rac7 (Ethos Reserve) disagreed with severity and commented:

> Valid edge case in as far as harvests would fail. However, funds won't get locked in the strategy. They can still be withdrawn through an appropriate withdraw() TX. Recommend downgrading to low since this is purely about state handling without putting any assets at risk. See screenshot below for simulation:
>
> 

```
  });

  it.only('should allow withdrawals', async function () {
    const {vault, strategy, want, wantHolder} = await loadFixture(deployVaultAndStrategyAndGetSigners)
    const userBalance = await want.balanceOf(wantHolderAddr);
    const depositAmount = toWantUnit('10');
    await vault.connect(wantHolder)['deposit(uint256)'](depositAmount);
    await strategy.harvest();

    await strategy.authorizedWithdrawUnderlying(toWantUnit('10'));
    await strategy.setEmergencyExit();
    // await strategy.harvest(); // this does fail

    await vault.connect(wantHolder).withdrawAll();
    const userBalanceAfterWithdraw = await want.balanceOf(wantHolderAddr);
    expect(userBalance.toString()).to.equal(userBalanceAfterWithdraw.toString()); // this passes
```

Trust (judge) commented:

> Medium severity is also appropriate when core functionality is impaired, even if there is no lasting damage.

tess3rac7 (Ethos Reserve) confirmed

[M-04] In `ReaperVaultV2`, we should update `lockedProfit` and `lastReport` before changing `lockedProfitDegradation`

Submitted by hansfriese

The locked profit degradation for the past will be changed with the new `lockedProfitDegradation`.

As a result, malicious users can steal others' rewards by frontrunning.

Proof of Concept

`setLockedProfitDegradation()` is used to change `lockedProfitDegradation` by admin.

```
    function setLockedProfitDegradation(uint256 degradation) ext
        _atLeastRole(ADMIN);
        require(degradation <= DEGRADATION_COEFFICIENT, "Degrada
        lockedProfitDegradation = degradation;
        emit LockedProfitDegradationUpdated(degradation);
    }
```

And `lockedProfitDegradation` is used to calculate the locked profit.

```
    function _calculateLockedProfit() internal view returns (uir
        uint256 lockedFundsRatio = (block.timestamp - lastReport
        if (lockedFundsRatio < DEGRADATION_COEFFICIENT) {
            return lockedProfit - ((lockedFundsRatio * lockedPrc
        }

        return 0;
    }
```

But it doesn't update `lockedProfit` and `lastReport` before changing `lockedProfitDegradation` so the below scenario would be possible.

1. Let's assume `lockedProfit = 200, lastReport = block.timestamp` after calling `report()`, `lockedProfitDegradation` are `6 hours in blocks`.
2. 3 hours later, 100 tokens of `lockedProfit` are released and added to the free funds. We can assume `report()` wasn't called for 3 hours.
3. At that time, `lockedProfitDegradation` is changed to `4 hours in blocks` and it means `200 * 3 / 4 = 150` tokens are released. As a result, free funds are increased by 50 inside the same block.
4. So a malicious user(it should be a pool) can front run `deposit()` with huge amounts before `lockedProfitDegradation` is changed and charge most of the new rewards(=50).

Similarly, already unlocked funds will be treated as locked again if `lockedProfitDegradation` is decreased.

Even if there is no front run as all depositors are pools, it's not fair to change locked/unlocked amounts that are confirmed already.

Recommended Mitigation Steps

We should modify `setLockedProfitDegradation()` like below.

```
    function setLockedProfitDegradation(uint256 degradation) ext
        _atLeastRole(ADMIN);
        require(degradation <= DEGRADATION_COEFFICIENT, "Degrada

        // update lockedProfit and lastReport
        lockedProfit = _calculateLockedProfit();
        lastReport = block.timestamp;

        lockedProfitDegradation = degradation;
        emit LockedProfitDegradationUpdated(degradation);
    }
```

Trust (judge) increased severity to High

tess3rac7 (Ethos Reserve) disagreed with severity and commented:

> I don't think this qualifies as a "high". It relies on a large harvest, followed by no harvests for a considerable amount of time, followed by the ADMIN (multisig) calling `setLockedProfitDegradation()` to set the degradation to a smaller value and being front-run by a malicious user calling `deposit()`.

- depositors are whitelisted via the `DEPOSITOR` role, so in this case they are trusted
- the ADMIN multisig wouldn't randomly change the degradation value without a really good reason to do so
- even if a whitelisted depositor turns malicious, and the ADMIN multisig ends up invoking `setLockedProfitDegradation()`, and ADMIN's TX gets front-run by said malicious depositor, it would, at best, result in some extra yield being squeezed by the malicious depositor—ONLY IF they have enough capital to overshadow the vault's TVL. There would be no loss of principal assets whatsoever for any depositor.

> There is slim chance for favorable conditions for this type of attack to succeed, and the capital requirements for the attacker would be quite high. And even after doing all this, they would at best earn a few more tokens out of the locked yield without impact any other user's deposits. It's not even a repeatable attack because ADMIN won't keep calling `setLockedProfitDegradation()` over and over. Very low risk in my opinion.

Trust (judge) decreased severity to Medium

[M-05] `upgradeProtocol` can create Peg Risk via Oracle Price Arbitrage

Submitted by GalloDaSballo

`upgradeProtocol` is meant to enable a new version of the protocol while retaining the same LUSD token.

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/LUSDToken.sol#L160-L181

```
        function upgradeProtocol(
```

In case of a migration, with the same collateral, but a new oracle, the system could open up to arbitrage between the two oracles via redemptions, allowing to extract value from the difference between the 2 prices.

This is because each oracle (e.g. chainlink), can change it's price based on two aspects:

- Hearbeat -> Maximum amount of time before the feed is updated
- Threshold -> % change at which the price is changed no matter what

In case of the oracle being different, for example having a different heartbeat setting, or simply having a different cadence (e.g. one refreshes at noon the other at 3 pm), the difference can open up to Arbitrage Strategies that can potentially increase risk to the system.

Arbitrage through Redemptions Explanation

The fact that that older version of the protocol can burn means they could allow for redemption arbitrage, leaking value.

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/LUSDToken.sol#L366-L367

```
            // old versions of the protocol may still burn
```

Burning of tokens can be performed via two operations:

- Repayment
- Redemptions

Repayment seems to be safest options and it's hard to imagine a scenario for exploit.

If the oracle offers a different price for redemptions, that can crate an incentive to go redeem against the older system, and since the older system cannot create new Troves, the CR for it could suffer.

The way in which this get's problematic is if there's positions that risk becoming under-collateralized in the old system and the debt from those positions is used to redeem against better collateralized positions on the new migrated system

This would create an economic incentive to leave the bad debt in older system as the new one is offering a more profitable opportunity.

Additional Resources

An example of desynch is what happened to a Gearbox Ninja, that got liquidated due to hearbeat differences

https://twitter.com/gearbox_intern/status/1587957233605918721

Remediation Steps

It will be best to ensure that a collateral is either in the old system, or on the new system, and if the same collateral is in both version, I believe the Oracle must be the same as to avoid inconsistent pricing.

It may also be best to change the migration pattern to one based on Zaps, which will offer good UX but reduce risk to the LUSD peg dynamic.

Trust (judge) commented:

> Warden did well to state all the hypotheticals, however imo the requirement that the two oracles must be different for this opportunity to arise is too theoretical for medium severity. Will leave for sponsor review.

tess3rac7 (Ethos Reserve) disputed and commented:

> however imo the requirement that the two oracles must different for this opportunity to arise is too theoretical for med severity. Will leave for sponsor

> review.
> Agree. Warden also overlooked redemption fee, which could be a very large % depending on the ratio of `redeemedAmount :: total outstanding debt of market in old protocol.` No one would arb if money is lost on each redemption.

> Seems more just like an informational warning to use the same oracles if we ever upgrade rather than a bug report.

[M-06] Denial of Liquidations and Redemptions by borrowing all reserves from AAVE

Submitted by GalloDaSballo, also found by Koolex and GalloDaSballo

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/ActivePool.sol#L239
https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperStrategyGranarySupplyOnly.sol#L200

Liquidations and Redemptions can be prevented by making `ActivePool._rebalance` revert by borrowing all collateral from AAVEs lendingPool.

The ActivePool will invest in the Vault, which will use the strategy to invest in the lending pool.

When withdrawing collateral, by Closing CDPs, Redeeming or Liquidating, `_rebalance` will be called.

In most logical cases (high capital efficiency), this will trigger a withdrawal from the Strategy

Which will trigger a withdrawawal from the LendingPool,

An attacker can deny this operation by borrowing all reserves from AAVE.

This will prevent all Liquidations, Redemptions as well as withdrawals, at will of the attacker.

This can be done to force the protocol to enter Recovery Mode, force re-absorptions and it can be pushed as far as to trigger bad debt.

Note that the attack can be performed maliciously without the need for a front-run, a sandwich (front-run + back-run) will just make it less costly (less interest paid) for the attacker but is not a way to prevent the attack.

Preamble to the POC

Any time funds are pulled from the ActivePool, `_rebalance` is called.

We know that if a withdrawal is sizeable enough, `_rebalance` will trigger `Strategy._withdraw` which will attempt to `withdraw` from the lending pool.

The goal of the POC then is to show how we can make it impossible to perform a withdrawal, guaranteeing a revert on all calls to `_rebalance` which consequently will brick Redemptions and Liquidations

Proof of Concept

The POC is coded in brownie, I have setup a MockFile to be able to fork optimism, with the final addresses hardcoded in the strategy (Granary).

Goal of the POC

The goal of the POC is to demonstrate that withdrawals from the pool can be denied.

This shows how we can trigger a revert against `LendingPool.withdraw` which we know will cause `_rebalance` to revert as well

Coded POC

The following mock allows us to interact with the forked contracts

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity ^0.8.0;

contract LendingPool {
  function deposit(
    address asset,
    uint256 amount,
    address onBehalfOf,
    uint16 referralCode
  ) external {}

  function borrow(
    address asset,
    uint256 amount,
    uint256 interestRateMode,
    uint16 referralCode,
    address onBehalfOf
  ) external {}

  function withdraw(
    address asset,
    uint256 amount,
    address to
  ) external {}
}
```

We can then fork optimism mainnet

```
brownie console --network optimism-main-fork
```

Run the following commands to show the attack

```
## Setup addresses
lp = LendingPool.at("0x8FD4aF47E4E63d1D2D45582c3286b4BD9Bb95DfE'
a_token = interface.ERC20("0xfF94cc8E2c4B17e3CC65d7B83c7e8c64303
weth = interface.ERC20("0x42000000000000000000000000000000000000
usdc = interface.ERC20("0x7F5c764cBc14f9669B88837ca1490cCa17c316

## Setup Actors
weth_whale = accounts.at("0xe50fa9b3c56ffb159cb0fca61f5c9d750e81
usdc_whale = accounts.at("0x625e7708f30ca75bfd92586e17077590c60e
strategy = a[0]
exploiter = a[1]

## Fund Strategy with WETH
weth.transfer(strategy, 20e18, {"from": weth_whale})

## Strategy Deposits WETH
weth.approve(lp, 20e18, {"from": strategy})
lp.deposit(weth, 20e18, strategy, 0, {"from": strategy})


## Fund exploiter with USDC, they will borrow WETH
usdc.transfer(exploiter, usdc.balanceOf(usdc_whale), {"from": us

## Setup collateral so we can dry up WETH
usdc.approve(lp, usdc.balanceOf(exploiter), {"from": exploiter})
lp.deposit(usdc, usdc.balanceOf(exploiter), exploiter, 0, {"from

## Borrow Max, so no WETH is borrowable
to_borrow = weth.balanceOf(a_token)
lp.borrow(weth, to_borrow, 2, 0, exploiter, {"from": exploiter})

print(weth.balanceOf(a_token))
0 ## No weth left, next withdrawal will revert

## Strategy will not be able withdraw
to_withdraw = a_token.balanceOf(strategy)
assert to_withdraw > 0

## REVERTS HERE
lp.withdraw(weth, to_withdraw, strategy, {"from": strategy})

>>> Transaction sent: 0x2d129abc6f69d74db7567de54d9932ac406d2212
  Gas price: 0.0 gwei   Gas limit: 20000000   Nonce: 3
  LendingPool.withdraw confirmed (SafeMath: subtraction overflow
```

Any time the Strategy needs to withdraw from the pool, because of `_rebalance` that withdrawal can be denied, which will consequently prevent Collateral from being pulled, which in turn will prevent Redemptions and Liquidations.

This means a overlevered malicious actor can bring down the peg of the system while preventing whichever liquidation or redemption they want

Remediation Steps

I’m unclear as to a specific remediation, as AAVE, by design, will lend out all of it’s reserves, meaning that the amount lent out should not be assumed as liquid.

Theoretically, re-balancing only manually should protect more assets, making the threshold for the attack higher.

However, any asset sent to the lending pool should be assumed illiquid, meaning that those amounts can be prevented from being withdrawable which will prevent Liquidations and Redemptions, potentially causing bad debt

Additional Considerations

If LUSD is liquid enough to be shorted, a goal as you’d assume the token to scale, then the attack not only can be performed against the system unconditionally, but can also become profitable as the attacker can arbitrarily force bad debt for the entire portion of collateral in the lendingPool, profiting from the loss of value.

Trust (judge) decreased severity to Medium and commented:

> Issue is valid. Severity is on the edge between med and high, because impact is “temporary freeze of funds” (cannot be long-term due to incredible interest costs), additionally users can always send a flashbot TX. High severity is almost always reserved for long-term impacts to the protocol, which I don’t see here (attacker sandwich cannot succeed for long-term).

tess3rac7 (Ethos Reserve) commented:

> I’m not sure what to make of such reports. To earn interest, there has to be some risk. The lowest risk option is to supply money for others to borrow on a battle-tested protocol such as Aave, Granary etc. Hundreds of yield-bearing strategies have been written by Reaper, Yearn etc. utilizing these protocols. The only way to truly mitigate this issue is to not utilize an external protocol, which conflicts with the ethos of the system.

> Considering this a medium/high bug means ignoring:

- dynamics of money markets, including kinked interest rates

- the fact that ethos will not allocate 100% of its capital to Aave/Granary as that would be reckless to do. Current value is 75% so wardens could easily replicate test scenarios but in the final deployment we'd start off with 10-25% and scale slowly, especially if we can add more strategies to the vault.

> I don't think either of the above can be ignored. Requesting judge review.

Trust (judge) commented:

> @GalloDaSballo - Would like to hear your fact-based opinion before landing on a severity.

GalloDaSballo (warden) commented:

> Multiple things to comment on but the juice of the finding is:

- We can deny getting the capital from AAVE V2 (Granary), because there's no borrow caps
- Denying the withdrawal makes `Vault.withdraw revert`
- `_rebalance` is called on all collateral changing operations
- Denying `_rebalance` via `Vault.withdraw` prevents the functionality of the protocol when it matters (liquidation only matters when there's debt to liquidate)

> This puts the debt collateralized by this collateral at risk (this is the base layer of the impact, but the impact is higher).

> Because `_rebalance` is called on all operations, every operation where `netAssetMovement` is negative, will call `Vault.withdraw`.

> This means that by denying the ability to recall the `yieldingAmount`, we have put the whole amount in the ActivePool at risk.

> Meaning that the comments about only the `yieldingAmount` being put at risk are not correct, the whole amount is at risk because `_rebalance` happens on all collateral changes.

> **Possible Solution**

> A LIFO solution would reduce the risk to what the sponsor commented, putting only the strategy capital at risk.

> A LIFO solution would always withdraw from the ActivePool first, and it would have `_rebalance` being called exclusively manually by the strategist / keeper.

> In that system, the attack would be limited to the % lent to AAVE (still at risk)

> However, the code in-scope is not offering a LIFO solution, it `_rebalance`s on each collateral change, meaning those operations will be denied.

> **On front-running and flashbots**

> Afaik private pools are not available on OP nor FTM, meaning that while front-running is “luck based”, it cannot be prevented.

> Also, front-running simply increases ROI, it’s not a pre-condition for the attack.

> **On interest rate / cost**

> If you were to use AAVEs linear model, even at 1,000% APR, the cost is just 3% per day.

> When enough collateral is present, this is not a high cost to bear to break the peg of the token and profit from it as well as the side effects. (Pay 3% per day, tank the token by 50%)

> **Summary**

> In summary, the whole collateral movement can be denied because of the rebalancing architecture.

> A LIFO solution could reduce the risk but doesn’t fully avoid the liquidity risk.

Trust (judge) commented:

> Too severe for QA by C4 standards, not severe enough for high, so will land on medium (impairment of core functionalities of the protocol under some preconditions).

[M-07] `DOMAIN_SEPARATOR()` is missing in LUSDToken.sol

Submitted by matrix_Owl, also found by Haipls

The `DOMAIN_SEPARATOR()` function in ERC2612 is an important part of the security of the standard. It is used to prevent replay attacks, which occur when a malicious user records a valid signed message and later sends it again to fraudulently perform an action on behalf of the original signer.

The `DOMAIN_SEPARATOR()` is generated based on specific contract parameters, including the contract's address, the chain ID, and a unique identifier. These parameters ensure that the domain separator is unique to the contract and the chain, and prevent attackers from using the same signature on a different chain or contract.

If the `DOMAIN_SEPARATOR()` function is missing from ERC2612, it can significantly impact the security of the standard. It can make it easier for attackers to replay valid signatures, since the domain separator provides a crucial part of the uniqueness and security of the signature.

Therefore, it's important to ensure that the `DOMAIN_SEPARATOR()` function is included and properly implemented in any contract that uses ERC2612.

Proof of Concept

Output form slither:

```
# Check LUSDToken

## Check functions
[✓] permit(address,address,uint256,uint256,uint8,bytes32,bytes3
        [✓] permit(address,address,uint256,uint256,uint8,bytes3
[✓] nonces(address) is present
        [✓] nonces(address) -> (uint256) (correct return type)
        [✓] nonces(address) is view
[ ] DOMAIN_SEPARATOR() is missing
[✓] totalSupply() is present
        [✓] totalSupply() -> (uint256) (correct return type)
        [✓] totalSupply() is view
```

```
    [✓] balanceOf(address) is present
            [✓] balanceOf(address) -> (uint256) (correct return type
            [✓] balanceOf(address) is view
    [✓] transfer(address,uint256) is present
            [✓] transfer(address,uint256) -> (bool) (correct return
            [✓] Transfer(address,address,uint256) is emitted
    [✓] transferFrom(address,address,uint256) is present
            [✓] transferFrom(address,address,uint256) -> (bool) (co
            [✓] Transfer(address,address,uint256) is emitted
    [✓] approve(address,uint256) is present
            [✓] approve(address,uint256) -> (bool) (correct return
            [✓] Approval(address,address,uint256) is emitted
    [✓] allowance(address,address) is present
            [✓] allowance(address,address) -> (uint256) (correct re
            [✓] allowance(address,address) is view
    [✓] name() is present
            [✓] name() -> (string) (correct return type)
            [✓] name() is view
    [✓] symbol() is present
            [✓] symbol() -> (string) (correct return type)
            [✓] symbol() is view
    [✓] decimals() is present
            [✓] decimals() -> (uint8) (correct return type)
            [✓] decimals() is view
        )
```

Tools Used

VS Code, Slither

Recommended Mitigation Steps

To mitigate this risk, it is recommended to follow the ERC2612 specification strictly and ensure that the `DOMAIN_SEPARATOR` is correctly implemented.

tess3rac7 (Ethos Reserve) confirmed via duplicate issue `#818`

[M-08] If the strategy incurs a loss the Active Pool will stop working until the shortfall is paid out entirely

Submitted by GalloDaSballo*, also found by* MiloTruck*,* bin2chen*,* hansfriese*,* imare*,* KingNFT*,* PaludoX0*,* 0xBeirao*,* AkshaySrivastav*,* kaden*, and* 0xRobocop

- Vaults are built with the idea that a loss could happen
- The scope mentions that a Loss scenario is in scope

This line, is written with the assumption that `sharesToAssets` will always be greater than or equal to `currentAllocated`

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/ActivePool.sol#L251-L252

```
            vars.profit = vars.sharesToAssets.sub(vars.currentAlloca
```

This is not the case as the Strategy MAY incur a loss.

In such cases, `_rebalance` on the ActivePool will not work until the subtraction stops underflowing `vars.sharesToAssets.sub(vars.currentAllocated);` will revert if any loss (even 1 wei) has happened.

Proof of Concept

When a loss happens, the `sharesToAssets` will decrease.

Because `vars.currentAllocated` tracks the amount deposited in the vault, this value will necessarily be greater than the `sharesToAsset` if any loss happened.

In that case this line will revert: `vars.profit = vars.sharesToAssets.sub(vars.currentAllocated);`

In the code shown, a loss could happen if the LendingPool has accounting errors.

For the in-scope codebase a loss could happen as a consequence of slashing or restructuring due to bad debt incurred by borrowers.

Coded POC

The following POC was built with brownie.

Mocked contract retain the core logic, but are rid of access control and other functions to keep the logic the same but reduce complexity of setup.

Setup brownie via `brownie console` (local environment is fine as I set-up mocks to make it easy).

```
## Setup tokens
token = MockToken.deploy({"from": a[0]})

## Deploy Vault
vault = ReaperVaultV2.deploy(token, {"from": a[0]})

## Deploy ActivePool
pool = MockActivePool.deploy(token, 2000, vault, 1, {"from": a[(

## Add to Active
token.approve(pool, 1e18, {"from": a[0]})
pool.depositColl(1e18, {"from": a[0]})

## Rebalance to invest
pool.manualRebalance(token, 0, {"from": a[0]})

## 20% of tokens are in the vault
print(token.balanceOf(vault))
200000000000000000

## Trigger loss to vault
vault.triggerLoss(1e17, {"from": a[0]})
## Confirm the loss has happened
print(vault.balance())
100000000000000000

## Now that a loss happened, any rebalance will revert
pool.manualRebalance(token, 1, {"from": a[0]})
Transaction sent: 0x798e759783ab59dda9c294178859fec5519179a2c31k
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 8
  MockActivePool.manualRebalance confirmed (Integer overflow)

<Transaction '0x798e759783ab59dda9c294178859fec5519179a2c31b89ak
pool.manualRebalance(token, 100, {"from": a[0]})
```

```
Transaction sent: 0x0c41ec1b74a05df6c7101522931cda6ba30139358ec2
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 9
  MockActivePool.manualRebalance confirmed (Integer overflow)

<Transaction '0x0c41ec1b74a05df6c7101522931cda6ba30139358ec239f(

## That's because the loss has been registered by the Vault
print(vault.convertToAssets(1e17))
50000000000000000

## But not by the Pool, triggering a revert at this line
> vars.profit = vars.sharesToAssets.sub(vars.currentAllocated);
```

Mocks Used

ActivePool.sol

```
// SPDX-License-Identifier: BUSL-1.1

pragma solidity ^0.8.0;

import {ReaperVaultV2} from "./ReaperVaultV2.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/utils/math/SafeMath.sol";


contract MockActivePool {
    using SafeMath for uint256;


    address immutable collateral;

    mapping(address => uint256) public collAmount;
    mapping(address => uint256) public yieldingPercentage; // co
    mapping(address => uint256) public yieldingAmount; // collat
    mapping(address => address) public yieldGenerator; // collat
    mapping(address => uint256) public yieldClaimThreshold; // c

    uint256 public yieldingPercentageDrift = 100; // rebalance i

    // Yield distribution params, must add up to 10k
    uint256 public yieldSplitTreasury = 20_00; // amount of yiel
    uint256 public yieldSplitSP = 40_00; // amount of yield to c
    uint256 public yieldSplitStaking = 40_00; // amount of yielc
    // Mock addresses, unused
    address public treasuryAddress = address(1);
    address public stabilityPoolAddress = address(2);
    address public lqtyStakingAddress = address(3);

    constructor(
        address _collateral,
        uint256 _yieldingPercentage,
        address _yieldGenerator,
        uint256 _yieldClaimThreshold
    ) {
        collateral = _collateral;
        yieldingPercentage[_collateral] = _yieldingPercentage;
        yieldGenerator[_collateral] = _yieldGenerator;
        yieldClaimThreshold[_collateral] = _yieldClaimThreshold;
    }

    function depositColl(uint256 amount) external {
      ERC20(collateral).transferFrom(msg.sender, address(this),
      collAmount[collateral] += amount;
    }

    function manualRebalance(address _collateral, uint256 _simul
        _rebalance(_collateral, _simulatedAmountLeavingPool);
    }

    struct LocalVariables_rebalance {
        uint256 currentAllocated;
        ReaperVaultV2 yieldGenerator;
        uint256 ownedShares;
        uint256 sharesToAssets;
        uint256 profit;
        uint256 finalBalance;
        uint256 percentOfFinalBal;
        uint256 yieldingPercentage;
        uint256 toDeposit;
        uint256 toWithdraw;
        uint256 yieldingAmount;
        uint256 finalYieldingAmount;
        int256 netAssetMovement;
        uint256 treasurySplit;
        uint256 stakingSplit;
        uint256 stabilityPoolSplit;
    }
    function _rebalance(address _collateral, uint256 _amountLeav
        LocalVariables_rebalance memory vars;

        // how much has been allocated as per our internal recor
        vars.currentAllocated = yieldingAmount[_collateral];

        // what is the present value of our shares?
        vars.yieldGenerator = ReaperVaultV2(yieldGenerator[_coll
        vars.ownedShares = vars.yieldGenerator.balanceOf(address
        vars.sharesToAssets = vars.yieldGenerator.convertToAsset

        // if we have profit that's more than the threshold, rec
        vars.profit = vars.sharesToAssets.sub(vars.currentAlloca
        if (vars.profit < yieldClaimThreshold[_collateral]) {
            vars.profit = 0;
        }

        // what % of the final pool balance would the current al
        vars.finalBalance = collAmount[_collateral].sub(_amountI
        vars.percentOfFinalBal =
            vars.finalBalance == 0 ? type(uint256).max : vars.cu

        // if abs(percentOfFinalBal - yieldingPercentage) > drif
        vars.yieldingPercentage = yieldingPercentage[_collateral
        vars.finalYieldingAmount = vars.finalBalance.mul(vars.yi
        vars.yieldingAmount = yieldingAmount[_collateral];
        if (
            vars.percentOfFinalBal > vars.yieldingPercentage
                && vars.percentOfFinalBal.sub(vars.yieldingPerce
        ) {
            // we will end up overallocated, withdraw some
            vars.toWithdraw = vars.currentAllocated.sub(vars.fir
            vars.yieldingAmount = vars.yieldingAmount.sub(vars.t
            yieldingAmount[_collateral] = vars.yieldingAmount;
        } else if (
            vars.percentOfFinalBal < vars.yieldingPercentage
                && vars.yieldingPercentage.sub(vars.percentOfFir
        ) {
            // we will end up underallocated, deposit more
            vars.toDeposit = vars.finalYieldingAmount.sub(vars.c
            vars.yieldingAmount = vars.yieldingAmount.add(vars.t
            yieldingAmount[_collateral] = vars.yieldingAmount;
        }

        // + means deposit, - means withdraw
        vars.netAssetMovement = int256(vars.toDeposit) - int256(
        if (vars.netAssetMovement > 0) {
            ERC20(_collateral).approve(yieldGenerator[_collatera
            ReaperVaultV2(yieldGenerator[_collateral]).deposit(u
        } else if (vars.netAssetMovement < 0) {
            ReaperVaultV2(yieldGenerator[_collateral]).withdraw(
                uint256(-vars.netAssetMovement), address(this),
            );
        }

        // if we recorded profit, recalculate it for precision a
        if (vars.profit != 0) {
            // profit is ultimately (coll at hand) + (coll alloc
            vars.profit =
                ERC20(_collateral).balanceOf(address(this)).add(
            if (vars.profit != 0) {
                // distribute to treasury, staking pool, and sta
                vars.treasurySplit = vars.profit.mul(yieldSplitT
                if (vars.treasurySplit != 0) {
                    ERC20(_collateral).transfer(treasuryAddress,
                }

                vars.stakingSplit = vars.profit.mul(yieldSplitSt
                if (vars.stakingSplit != 0) {
                    ERC20(_collateral).transfer(lqtyStakingAddre
                }

                vars.stabilityPoolSplit = vars.profit.sub(vars.t
                if (vars.stabilityPoolSplit != 0) {
                    ERC20(_collateral).transfer(stabilityPoolAdd
                }
            }
        }
    }
}
```

ReaperVaultV2.sol

```
// SPDX-License-Identifier: BUSL-1.1

pragma solidity ^0.8.0;

import "@openzeppelin/contracts/access/AccessControlEnumerable.s
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/extensions/IERC20Met
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol'
import "@openzeppelin/contracts/utils/math/Math.sol";

contract ReaperVaultV2 is ERC20, AccessControlEnumerable {
    uint256 totalAllocated = 0;

    IERC20Metadata public immutable token;

    constructor(address _token) ERC20("test", "TEST") {
        token = IERC20Metadata(_token);
    }

    function triggerLoss(uint256 amt) external {
        token.transfer(address(1337), amt);
    }

    function deposit(uint256 assets, address receiver) external
        shares = _deposit(assets, receiver);
    }

    function withdraw(uint256 assets, address receiver, address
        revert("No op");
    }

    function convertToAssets(uint256 shares) public view returns
        if (totalSupply() == 0) return shares;
        return (shares * _freeFunds()) / totalSupply();
    }

    function _deposit(uint256 _amount, address _receiver) intern
        require(_amount != 0, "Invalid amount");
        uint256 pool = balance();

        uint256 freeFunds = _freeFunds();
        uint256 balBefore = token.balanceOf(address(this));
        token.transferFrom(msg.sender, address(this), _amount);
        uint256 balAfter = token.balanceOf(address(this));
        _amount = balAfter - balBefore;
        if (totalSupply() == 0) {
            shares = _amount;
        } else {
            shares = (_amount * totalSupply()) / freeFunds; // 
        }
        _mint(_receiver, shares);
    }
```

```
        function balance() public view returns (uint256) {
            return token.balanceOf(address(this)) + totalAllocated;
        }

        // No harvest, so it's not going to make a difference
        function _freeFunds() public view returns (uint256) {
            return balance();
        }
    }
```

MockToken.sol

```
    // SPDX-License-Identifier: BUSL-1.1

    pragma solidity ^0.8.0;

    import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

    contract MockToken is ERC20 {
      constructor() ERC20("Mock", "Mock"){
        _mint(msg.sender, 1000e18);
      }
    }
```

Remediation Steps

A slashing mechanism would need to be added to account for a loss.

This should be fairly involved as to not create gotchas.

Intuitively, I believe, that the funds in the activePool would need to be mapped against the funds invested in Vaults as to reconcile the “deposited value” with the “slashed value”.

Alternatively, for the time being, a “ShortFall” fund could be instituted, fully knowing that if something goes wrong, the fund will have to cover the loss.

tess3rac7 (Ethos Reserve) disputed via duplicate issue #747

[M-09] Last Trove may be prevented from redeeming

Submitted by Ityu, also found by GalloDaSballo

In `redeemCollateral()` of RedemptionHelper.sol, the LUSD `balanceOf` the redeemer is checked against the specific collateral recorded LUSD debt (both active and defaulted).

```
    function redeemCollateral(
            address _collateral,
            address _redeemer,
            uint _LUSDamount,
            address _firstRedemptionHint,
            address _upperPartialRedemptionHint,
            address _lowerPartialRedemptionHint,
            uint _partialRedemptionHintNICR,
            uint _maxIterations,
            uint _maxFeePercentage
        )
            external override
        {
            _requireCallerIsTroveManager();
            _requireValidCollateralAddress(_collateral);
            RedemptionTotals memory totals;

            _requireValidMaxFeePercentage(_maxFeePercentage);
            _requireAfterBootstrapPeriod();
            totals.price = priceFeed.fetchPrice(_collateral);
            ICollateralConfig collateralConfigCached = collateralCor
            totals.collDecimals = collateralConfigCached.getCollater
            totals.collMCR = collateralConfigCached.getCollateralMCF
            _requireTCRoverMCR(_collateral, totals.price, totals.col
            _requireAmountGreaterThanZero(_LUSDamount);
            _requireLUSDBalanceCoversRedemption(lusdToken, _redeemer

            totals.totalLUSDSupplyAtStart = getEntireSystemDebt(_col
            // Confirm redeemer's balance is less than total LUSD su
            assert(lusdToken.balanceOf(_redeemer) <= totals.totalLUS
            ...
    }
```

This makes sense in a single collateral system such as Liquity, but is problematic in a multi-collateral one like Reserve. Since each collateral type tracks its own debt but mints the same LUSD token, LUSD supply (and thus balance) being less than the collateral debt is no longer an invariant. This can can result in:

- Last trove may be prevented from redeeming by griefers.
- Users that deposit into multiple Trove types may be prevented from redeeming.

Proof of Concept

Last trove may be prevented from redeeming

Consider the cases when

```
- There are 2 Trove types (wBTC and wETH).
- There is 10000 total LUSD debt in the wBTC Troves.
- Stability Pool has 150 LUSD deposited i.e. full liquidity to c
- There is 100 total LUSD debt in the wETH pool.
- ETH prices crash and all Troves get liquidated except the last
```

A griefer can front-run the last Trove from redeeming by sending the user weth with the amount `entireSystemDebt` + 1.

In a similar case as above, any users that may borrow from multiple Troves types such that their LUSD balance is greater than the total collateral debt will be prevented from redeeming. However, this is not as problematic because they can just send their excess tokens out.

Recommended Mitigation Steps

Consider removing this check as the invariant no longer applies.

tess3rac7 (Ethos Reserve) confirmed via duplicate issue `#549`

[M-10] P can be updated to zero which can cause a DOS when liquidating troves

Submitted by koxuan, also found by fs0c

P is asserted to never be zero in `_updateRewardSumAndProduct` which is called for every liquidation. However, there is an edge case that can cause P to be zero, causing DOS to certain liquidations.

Proof of Concept

In `_updateRewardSumAndProduct` notice `assert(newP > 0);`. However, the `SCALE_FACTOR` check is insufficient in ensuring P is always more than zero. Three cycles of very small P can cause P to drop to zero and hence causing revert for that particular liquidation of troves. POC steps are as follows,

1. First liquidation newProductFactor is 1e2, `1e18*1e2/1e18 = 1e2`, since its less than scale factor. 1e9 is multiplied before dividing with 1e18. P = 1e11 after 1st liquidation cycle.
2. Second liquidation newProductFactor is 1e3, `1e11*1e3/1e18 = 0`, since its less than scale factor. 1e9 is multipled. P = 1e5
3. Third liquidation newProductFactor is 1e3, `1e5*1e3/1e8 = 0`, since its less than scale factor, 1e9 is multipled. However, even after 1e9 is multiplied it will result in 1e17, which is less than DECIMAL_PRECISION of 1e18. P = 0.
4. Revert due to `assert(newP > 0);`.

```
        // If the Stability Pool was emptied, increment the epoc
        if (newProductFactor == 0) {
            currentEpoch = currentEpochCached.add(1);
            emit EpochUpdated(currentEpoch);
            currentScale = 0;
            emit ScaleUpdated(currentScale);
            newP = DECIMAL_PRECISION;


        // If multiplying P by a non-zero product factor would r
        } else if (currentP.mul(newProductFactor).div(DECIMAL_PR
            newP = currentP.mul(newProductFactor).mul(SCALE_FACT
            currentScale = currentScaleCached.add(1);
            emit ScaleUpdated(currentScale);
        } else {
            newP = currentP.mul(newProductFactor).div(DECIMAL_PR
        }

        assert(newP > 0);
        P = newP;

        emit P_Updated(newP);
    }
```

Reasons for putting high is because

1. It breaks the invariant of P being > 0 which according to the docs would break deposit tracking when Pool is not empty.
2. Even though in some lucky cases, smaller liquidation can be made if batch liquidation is done, in the event that P is at a precarious place whereby all permutations of liquidations done will result in P being 0, liquidation will be DOSed which can be detrimental to protocol as they cannot liquidate bad debt and hence might lead it to insolvency.

Recommended Mitigation Steps

Recommend setting `SCALE_FACTOR` to 1e18, even though the docs did explain that 1e9 is used instead of 1e18 to ensure negligible precision loss, the alternative option is redesigning the mitigation mechanism of rounding error for P.

tess3rac7 (Ethos Reserve) commented via duplicate issue `#444`:

> Since this is a disclosed issue in the Liquity repo as of 2 weeks before the time of this comment, and we have made no changes to this piece of code that makes it different from the liquity liquidation logic, I'm not sure if this is a valid bug report made as part of the C4A content, or if it's just relaying of information that has been public knowledge since before the contest began. Requesting judge to weigh in.

Trust (judge) commented via duplicate issue `#444`:

> The affected code is in scope, therefore imo the issue is valid unless communicated otherwise.

[M-11] `updateStrategyAllocBPS()` can cause loss of ActivePool's collateral during an emergency exit

Submitted by peakbolt*, also found by* 0xsomeone*,* 0xbepresent*,* codeislight*,* 0xbepresent*,* trustindistrust*, and* 0xTheC0der

https://github.com/code-423n4/2023-02-ethos/blob/main/Ethos-Vault/contracts/ReaperVaultV2.sol#L191-L199
https://github.com/code-423n4/2023-02-ethos/blob/main/Ethos-Vault/contracts/abstract/ReaperBaseStrategyv4.sol#L123

The function `updateStrategyAllocBPS()` can cause ActivePool to record an incorrect profit after `setEmergencyExit()` is triggered.

Impact

The incorrect profit will cause a large portion of the ActivePool's collateral to be distributed to Treasury, Staking Pool and Stability Pool. Depositors and Stakers can then withdraw the profits, leading to loss of ActivePool's collateral.

Background

In Ethos Reserve, the Vault rehypothecates the collateral from ActivePool using one or more Strategy, which will deposit the funds in other protocols (e.g. lending pool) to farm for yields.

Only Guardian and above roles are able to trigger `setEmergencyExit()` on a specific Strategy to force it to exit all its position upon the next harvest, depositing all funds from lending pool into the Vault. How it works, is that `setEmergencyExit()` will trigger Vault to `revokeStrategy()`, setting the strategy's `allocBPS` to `0`. This sets Strategy allocation to `0` and increases Strategy's debt, so that it will repay Vault all the funds.
(see https://github.com/code-423n4/2023-02-ethos/blob/main/Ethos-Vault/contracts/abstract/ReaperBaseStrategyv4.sol#L156-L160)

Note that `setEmergencyExit()` is not reversible. I believe this is to protects funds from being re-deployed into the lending pool during an emergency situation (e.g.

lending pool hacked or market crash). And it is different from Vault's EmergencyShutdown, which is effected on all Strategies and is reversible.

Detailed Explanation

The issue is that, Strategist (a lower privilege role than Guardian) is able to reverse `revokeStrategy()` by calling `updateStrategyAllocBPS()` with a non-zero value to increase the strategy allocation. This will lead to a reduction of the strategy's debt and cause an incorrect profit to be recorded when it liquidate its positions in the next harvest. Due to the incorrect profit, a fee will be charged on it and transferred to Treasury, leaving less funds for the Vault.

https://github.com/code-423n4/2023-02-ethos/blob/main/Ethos-Vault/contracts/ReaperVaultV2.sol#L191-L199

https://github.com/code-423n4/2023-02-ethos/blob/main/Ethos-Vault/contracts/abstract/ReaperBaseStrategyv4.sol#L123

Even worse, `updateStrategyAllocBPS()` will cause an increase to vault's totalAllocated value during `harvest()` , while the incorrect profit is transferred to the Vault during harvest. Both of these changes will lead to a discrepancy in the vault's `totalAllocated` and its token balance, causing the vault's total balance to be incorrect and higher than actual. This leads to a higher share price.

https://github.com/code-423n4/2023-02-ethos/blob/main/Ethos-Vault/contracts/ReaperVaultV2.sol#L521

https://github.com/code-423n4/2023-02-ethos/blob/main/Ethos-Vault/contracts/ReaperVaultV2.sol#L528

With a higher share price, ActivePool's owned asset value in the Vault will be inflated. This will cause ActivePool to record an incorrect profit in the next `_rebalance()` , and distrbute them to Treasury, Staking Pool and Stability Pool.

https://github.com/code-423n4/2023-02-ethos/blob/main/Ethos-Core/contracts/ActivePool.sol#L239-L309

Depositors and Stakers can then withdraw the profits, leading to loss of ActivePool's collateral.

Proof of Concept

Add the following test case to `Ethos-Vault/test/start-test.js`. This shows that ActivePool's asset value will be inflated due to the issue. The next test case will show that inflated asset value will cause ActivePool's `_rebalance()` to record a profit and distribute them to the respective pools, that can be withdrawn.

```
    it.only('updateStrategyAllocBPS can cause loss of ActivePool col
      const {vault, strategy, want, wantHolder, strategyAddress, str
        await loadFixture(deployVaultAndStrategyAndGetSigners);
      // intialize guardian account with ETH for gas
      const tx = await strategist.sendTransaction({
        to: guardianAddress,
        value: ethers.utils.parseEther('0.1'),
      });
      await tx.wait();

      // Treasury owned asset value starts with zero
      const treasurySharesBefore = await vault.balanceOf(treasuryAdd
      const treasuryAssetsBefore = await vault.convertToAssets(treas
      expect(treasuryAssetsBefore).to.equal(0);

      // ActivePool deposits 10 WBTC
      await vault.connect(wantHolder)['deposit(uint256)'](toWantUnit
      await strategy.harvest();

      // Expect ActivePool's owned share and asset to be equal to 10
      const activePoolSharesBefore = await vault.balanceOf(wantHolde
      const activePoolAssetsBefore = await vault.convertToAssets(act
      expect(activePoolSharesBefore).to.equal(toWantUnit('10'));
      expect(activePoolAssetsBefore).to.equal(toWantUnit('10'));

      /* Guardian calls setEmergencyExit().
      *  This triggers Vault to revokeStrategy() and set strategy's
      *  By design, this will force strategy to exit all its positio
      *  return funds to vault in the next harvest().
      */
      await strategy.connect(guardian).setEmergencyExit();

      /* Strategist set AllocBPS back to 10000 (100%).
      *  This will reverse the revokeStrategy() and cause strategy's
      *  to be reduced in next harvest()
      */
      await vault.connect(strategist).updateStrategyAllocBPS(strateg

      /* Strategy will liquidate all its position due to emergency e
      * However, it will also record an incorrect profit due to redu
      */
      await strategy.harvest();

      // Jump ahead for incorrect profit to unlock
      await moveTimeForward(3600*7);

      // Treasury will gain fees of 1.56 WBTC on the incorrect profi
      const treasurySharesAfter = await vault.balanceOf(treasuryAddr
      const treasuryAssetsAfter = await vault.convertToAssets(treasu
      expect(treasuryAssetsAfter).to.not.equal(treasuryAssetsBefore)
      expect(treasuryAssetsAfter).to.equal(156880733);

      // ActivePool's owned asset value is incorrectly inflated
      // This is due to increased share price from the incorrect pro
      const activePoolSharesAfter = await vault.balanceOf(wantHolder
      const activePoolAssetsAfter = await vault.convertToAssets(acti
      expect(activePoolAssetsAfter).to.equal("1743119266");
      expect(activePoolAssetsAfter).to.not.equal(activePoolAssetsBef

      /* ActivePool will record a profit of 7.43 WBTC (74% of initia
      *  In the next ActivePool's _rebalance(), the  incorrect profi
      *  Staking Pool and StabilityPool.
      *  Depositors and Stakers will be able to withdraw the profits
      */
      const estimatedActivePoolProfit = activePoolAssetsAfter - acti
      expect(estimatedActivePoolProfit).to.be.equal(743119266);

    });
```

Add the following test case to `Ethos-Core/test/PoolsTest.js`. Note that this is an test independent from the preivous test case just to show that ActivePool will record a profit when the share asset value increases, and the profit will be distributed to the respective pools.

```
  it.only('simulate incorrect profit to show that _rebalance() cal
          await setReasonableDefaultStateForYielding();
```

```
            // Simulate incorrect profit: mint 1 ether to vault.
            // This will increase the vault share price and inflate
            await collaterals[0].mint(vaults[0].address, dec(1, 'eth

            // Trigger ActivePool's _rebalance() via sendCollateral(
            // ActivePool will record a profit due to the inflated c
            const sendCollData = th.getTransactionData('sendCollater
              [collaterals[0].address, alice, web3.utils.toHex(dec(1
            await mockBorrowerOperations.forward(activePool.address,

            // The incorrect profit will be distributed to Treasury,
            assert.equal((await collaterals[0].balanceOf(treasury.ac
            assert.equal((await collaterals[0].balanceOf(stabilityPo
            assert.equal((await collaterals[0].balanceOf(lqtyStaking
    })
```

Recommended Mitigation Steps

The fix is to block all changes to strategy's `allocBPS` after `setEmergencyExit()`.

Since `allocBPS` is already tracked within `ReaperVaultV2.sol`, it is better to refactor and shift `emergencyExit` from `ReaperBaseStrategyv4.sol` to `ReaperVaultV2.StrategyParams`. With that, the fix can simply just to add a check for emergency exit within `updateStrategyAllocBPS()`.

[tess3rac7 (Ethos Reserve) disagreed with severity, but confirmed via duplicate issue `#716`]

[Trust (judge) decreased severity to Medium]

[M-12] ReaperVaultERC4626 is not EIP-4626 compliant and integrations can result in loss of funds

Submitted by [DadeKuma], *also found by* [rbserver]

Contracts that integrate with the `ReaperVaultERC4626` vault (including Ethos contracts) may wrongly assume that the functions are EIP-4626 compliant, which it might cause integration problems in the future, that can lead to a wide range of issues for both parties, including loss of funds.

Proof of Concept

This PoC describes this issue for the `withdraw` function, but there is the same problem with the `redeem` function.

EIP-4626 specification says that the `withdraw` function:

```
// Burns shares from owner and sends exactly assets of underlyir

// MUST revert if all of assets cannot be withdrawn (due to witł
// the owner not having enough shares, etc).
```

This is the `withdraw` function:

```
File: Ethos-Vault\contracts\ReaperVaultERC4626.sol

    function withdraw(
        uint256 assets,
        address receiver,
        address owner
    ) external override returns (uint256 shares) {
        shares = previewWithdraw(assets); // previewWithdra
        if (msg.sender != owner) _spendAllowance(owner, msg
        _withdraw(shares, receiver, owner);
    }
```

When the internal `_withdraw` is called, the `value` represents the total amount of assets that will be transferred to the receiver. There is a special case where there could be a withdrawal that exceeds the total balance of the vault:

```
File: Ethos-Vault\contracts\ReaperVaultV2.sol

            if (value > token.balanceOf(address(this))) {
                uint256 totalLoss = 0;
                uint256 queueLength = withdrawalQueue.length;
            uint256 vaultBalance = 0;
            for (uint256 i = 0; i < queueLength; i = i.unch
                vaultBalance = token.balanceOf(address(this
                if (value <= vaultBalance) {
                    break;
                }

                address stratAddr = withdrawalQueue[i];
                uint256 strategyBal = strategies[stratAddr]
                if (strategyBal == 0) {
                    continue;
                }

                uint256 remaining = value - vaultBalance;
                uint256 loss = IStrategy(stratAddr).withdra
                uint256 actualWithdrawn = token.balanceOf(a

                // Withdrawer incurs any losses from withdr
                if (loss != 0) {
                    value -= loss;
                    totalLoss += loss;
                    _reportLoss(stratAddr, loss);
                }

                strategies[stratAddr].allocated -= actualWi
                totalAllocated -= actualWithdrawn;
            }

            vaultBalance = token.balanceOf(address(this));
            if (value > vaultBalance) {
                value = vaultBalance;
            }

            require(
                totalLoss <= ((value + totalLoss) * withdra
                "Withdraw loss exceeds slippage"
            );
        }

        token.safeTransfer(_receiver, value);
```

In this case, if a strategy incurs any losses, the actual withdrawal amount will NOT be the exact same specified when calling the `withdraw` function, as it will be less than that, as **the loss is detracted from the withdrawn value:**

```
File: Ethos-Vault\contracts\ReaperVaultV2.sol

                    // Withdrawer incurs any losses from withdr
                    if (loss != 0) {
                        value -= loss;
                        totalLoss += loss;
                        _reportLoss(stratAddr, loss);
                    }
```

If that happens, then `assets requested > assets received`.

As the specification says that the withdrawal process `MUST revert if all of assets cannot be withdrawn (due to withdrawal limit being reached, slippage, the owner not having enough shares, etc)`, this makes the `ReaperVaultERC4626` non EIP-4626 compliant.

This might cause integration problems in the future, which can lead to a wide range of issues, including loss of funds.

Because EIP-4626 is aimed to create a consistent and robust implementation pattern for Tokenized Vaults, and even a slight deviation from the standard would break composability (and potentially lead to a loss of funds), this is categorized as a high risk.

Recommended Mitigation Steps

The `withdraw` and `redeem` functions should be modified to meet the specifications of EIP-4626: the `value` transferred must always be equal to the `assets` withdrawn. In case this is not true, the transaction must be reverted.

Trust (judge) decreased severity to Medium

tess3rac7 (Ethos Reserve) confirmed

[M-13] DOS by directly transferring assets to Reaper Vault

Submitted by gjaldon

https://github.com/code-423n4/2023-02-ethos/blob/main/Ethos-Core/contracts/ActivePool.sol#L251
https://github.com/code-423n4/2023-02-ethos/blob/main/Ethos-Core/contracts/ActivePool.sol#L264-L274
https://github.com/code-423n4/2023-02-ethos/blob/main/Ethos-Vault/contracts/ReaperVaultERC4626.sol#L207
https://github.com/code-423n4/2023-02-ethos/blob/main/Ethos-Vault/contracts/ReaperVaultERC4626.sol#L185

Attacker can DOS the protocol by directly transferring assets to the ReaperVault. This causes a DOS by underflow in the `ActivePool._rebalance` logic. All core ActivePool functionality will be disabled including all BorrowerOperations functionality and other functionality that relies on ActivePool. This essentially renders the protocol unusable until the protocol team figures out the fix, which is to directly transfer enough assets to ReaperVault to address the underflow. Even then, the attacker can just repeat the attack and make the protocol unusable once again.

Proof of Concept

With the below steps, an attacker can DOS the protocol:

1. A trove already exists and an initial deposit to the ReaperVault was already done by the ActivePool.
2. Bob opens a Trove using 190e3 WBTC as collateral (this could be any amount enough to satisfy the minimum borrowing amount of 90 LUSD).
3. Alice (the Attacker) frontruns Bob's openTrove transaction and transfers WBTC directly to the ReaperVault. The amount of assets transferred will be 1 + amount of assets ActivePool will deposit to the ReaperVault based on yieldingPercentage when Bob's openTrove tx is processed.
4. Due to Alice's frontrun, the below branch in `ActivePool._rebalance` will trigger during Bob's `openTrove` transaction:

```
        } else if (vars.netAssetMovement < 0) {
            IERC4626(yieldGenerator[_collateral]).withdraw(uint2
        }
```

`vars.netAssetMovement` will be -1 because of Alice's direct transfer of assets amounting to 1 + ActivePool's deposit amount.

5. Because of Alice's direct transfer of WBTC to ReaperVault, let's say 1 Vault share is now worth 15 units of WBTC. `ReaperVaultERC4626.withdraw` converts assets to shares with some rounding up behavior and then converts those shares to assets again to get the amount withdrawn. This leads to at least 15 units of WBTC being withdrawn even though `vars.netAssetMovement = 1`.
6. It is this difference between the amount of assets withdrawn from the ReaperVault and the accounting for total yield amount (`yieldingAmount[_collateral]`) that causes an underflow in subsequent calls to `ActivePool._rebalance`. The underflow happens here:

```
# `vars.sharesToAssets` is now less than `vars.currentAllocated`
# `vars.sharesToAssets` is total assets in the Vault while `vars
vars.profit = vars.sharesToAssets.sub(vars.currentAllocated);
```

Here is the git diff for a test POC that shows the attack. To run the POC, do the following:

1. Copy all contents of the gist to `/tmp/changes.patch`
2. Clone the project repo and cd to the Ethos-Core directory.
3. Run `git apply /tmp/changes.patch` in the Ethos-Core directory.
4. Run `npm install` from the command line from Ethos-Core's root directory
5. Run `npx hardhat test --grep "DOS attack"`. The test should pass

The git diff is huge since it includes copying of Ethos-Vault contracts into the Ethos-Core project so they can be used within the Ethos-Core tests. The test looks like:

```
    it("DOS attack", async () => {
      await activePool.setYieldingPercentage(collaterals[2].addr
        from: owner,
      });
      await activePool.setYieldClaimThreshold(collaterals[2].add
        from: owner,
      });
```

```
await collaterals[2].mint(alice, dec(10_000, 8));
await collaterals[2].approve(borrowerOperations.address, c
  from: alice,
});
await collaterals[2].mint(bob, dec(10_000, 8));
await collaterals[2].approve(borrowerOperations.address, c
  from: bob,
});
await collaterals[2].mint(carol, dec(10_000, 8));
await collaterals[2].approve(borrowerOperations.address, c
  from: carol,
});
await lusdToken.unprotectedMint(alice, dec(100_000, 18));
await lusdToken.unprotectedMint(bob, dec(100_000, 18));

await priceFeed.setPrice(collaterals[2].address, dec(100_(
const reaperVault = contracts.erc4626vaults[2];
await reaperVault.grantRole(
  await reaperVault.DEPOSITOR(),
  activePool.address
);

await borrowerOperations.openTrove(
  collaterals[2].address,
  dec(190, 3),
  th._100pct,
  dec(90, 18),
  alice,
  alice,
  { from: alice }
);

// Alice frontruns Bob's deposit by transfering 300_001 WE
// 300_001 WBTC is 1 unit of WBTC more than the yield that
// deposited to the ReaperVault.
await collaterals[2].transfer(reaperVault.address, 300001,
  from: alice,
});

await borrowerOperations.openTrove(
  collaterals[2].address,
  dec(3, 6),
  th._100pct,
  dec(90, 18),
  bob,
          bob,
          { from: bob }
        );

        // Protocol is now unusable
        await assertRevert(
          borrowerOperations.closeTrove(collaterals[2].address, {
            from: alice,
          })
        );

        await assertRevert(
          borrowerOperations.closeTrove(collaterals[2].address, {
            from: bob,
          })
        );

        await assertRevert(
          borrowerOperations.openTrove(
            collaterals[2].address,
            dec(190, 3),
            th._100pct,
            dec(90, 18),
            carol,
            carol,
            { from: carol }
          )
        );
      });
```

Tools Used

Manual Review, Hardhat/Truffle, VSCode

Recommended Mitigation Steps

Changing ReaperVaultERC4626 withdraw to round down instead of up fixes this issue. Below is the fixed code:

```
      function previewWithdraw(uint256 assets) public view overric
          if (totalSupply() == 0 || _freeFunds() == 0) return 0;
          shares = (assets * totalSupply()) / _freeFunds();
      }
```

OxBebis (Ethos Reserve) disagreed with severity and commented:

> This doesn't seem like a reasonable attack vector, but definitely good to know. I think it should be downgraded to low. In my opinion, changing the implementation carries far greater risk than the potential for this DOS.

Trust (judge) commented:

> I think this vector qualifies for medium. Warden has stated the conditions for the attack.

OxBebis (Ethos Reserve) confirmed and commented:

> We'll confirm this as medium, we're just going to remove the ERC4626 interface since this will be a private vault.

[M-14] `lastFeeOperationTime` is not modified correctly in function `_updateLastFeeOpTime()`, resulting a much slower decay model for borrowing base rate

Submitted by chaduke*, also found by* d3e4

`lastFeeOperationTime` is not modified correctly in function `_updateLastFeeOpTime()`. As a result, `decayBaseRateFromBorrowing()` will decay the base rate more slowly than expected (worst case half slower).

Since borrowing base rate is so fundamental to the protocol, I would rate this finding as High.

Proof of Concept

We show how `decayBaseRateFromBorrowing()` will decay the base rate more slowly than expected because of the wrong modification of `lastFeeOperationTime` in `_updateLastFeeOpTime()`:

1. `decayBaseRateFromBorrowing()` calls `_calcDecayedBaseRate()` to calculate the decayed base rate based how many minutes elapsed since last recorded

lastFeeOperationTime.

```
function decayBaseRateFromBorrowing() external override {
        _requireCallerIsBorrowerOperations();

        uint decayedBaseRate = _calcDecayedBaseRate();
        assert(decayedBaseRate <= DECIMAL_PRECISION);  // The ba

        baseRate = decayedBaseRate;
        emit BaseRateUpdated(decayedBaseRate);

        _updateLastFeeOpTime();
    }

   function _calcDecayedBaseRate() internal view returns (uint)
        uint minutesPassed = _minutesPassedSinceLastFeeOp();
        uint decayFactor = LiquityMath._decPow(MINUTE_DECAY_FACT

        return baseRate.mul(decayFactor).div(DECIMAL_PRECISION);
    }

 function _minutesPassedSinceLastFeeOp() internal view returns (
        return (block.timestamp.sub(lastFeeOperationTime)).div(S
 }
```

2. decayBaseRateFromBorrowing() then calls _updateLastFeeOpTime() to set lastFeeOperationTime to the current time if at least 1 minute pass.

```
 function _updateLastFeeOpTime() internal {
        uint timePassed = block.timestamp.sub(lastFeeOperationTi

        if (timePassed >= SECONDS_IN_ONE_MINUTE) {
            lastFeeOperationTime = block.timestamp;
            emit LastFeeOpTimeUpdated(block.timestamp);
        }
    }
```

3. The problem with such an update of lastFeeOperationTime is, if 1.999 minutes had passed, the base rate will only decay for 1 minute, at the same time, 1.999 minutes will be added on lastFeeOperationTime. In other words, in a

worst scenario, for every 1.999 minutes, the base rate will only decay for 1 minute. Therefore, the base rate will decay more slowly then expected.

4. The borrowing base rate is very fundamental to the whole protocol. Any small deviation is accumulative. In the worse case, the decay speed will slow down by half; on average, it will be 0.75 slower.

Tools Used

VSCode

Recommended Mitigation Steps

Using the effective elapsed time that is consumed by the model so far to revise `lastFeeOperationTime`.

```
    function _updateLastFeeOpTime() internal {
        uint timePassed = block.timestamp.sub(lastFeeOperationTi

        if (timePassed >= SECONDS_IN_ONE_MINUTE) {
-            lastFeeOperationTime = block.timestamp;
+            lastFeeOperationTime += _minutesPassedSinceLastFee(
            emit LastFeeOpTimeUpdated(block.timestamp);
        }
    }
```

Trust (judge) decreased severity to Medium

tess3rac7 (Ethos Reserve) disputed via duplicate issue `#850`

Low Risk and Non-Critical Issues

For this contest, 68 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **GalloDaSballo** received the top score from the judge.

The following wardens also submitted reports: MohammedRizwan, SleepingBugs, Raiders, Josiah, Udsen, 0xsomeone, yongskiws, Lavishq, rbserver, delfin454000, PawelK, bin2chen, CodingNameKiki, Kaysoft, tsvetanovv, hansfriese, catellatech, brgltd, zzzitron, imare, ch0bu, peanuts, ast3ros, lukris02, shark, CodeFoxInc, SuperRayss, hacker-dom, Bjorn_bug, PaludoX0, Phantasmagoria, martin, 0xackermann, ABA, Breeje, matrix_0wl, 0xnev, 0xTheC0der, luxartvinsec, Rickard, fs0c, IceBear, cryptonue, DadeKuma, vagrant, 0xSmartContract, emmac002, chrisdior4, 0xAgro, descharre, Rolezn, Viktor_Cortess, codeislight, RaymondFam, tnevler, Bnke0x0, trustindistrust, btk, 0x3b, Co0nan, BRONZEDISC, Sathish9098, arialblack14, UdarTeam, dontonka, DeFiHackLabs, *and* chaduke.

Executive Summary

The following QA Report is an aggregate of smaller reports divided by contract / topic.

The following criteria are used for suggested severity:

- L - Low Severity -> Some risk
- R - Refactoring -> Suggested changes for improvements
- NC - Non-Critical / Informational -> Minor suggestions

1. Vault

1.1. L - Strategy doesn't have `inCaseTokensGetStuck`

Most Yield Farming Strategies interact with other protocols and for this reason they are subject to airdrops.

Without a `inCaseTokensGetStuck` these extra tokens would not be claimable and would be lost forever

1.2. L - Deposit / Withdraw use FOT compatible math, but withdrawing from strategy doesn't

While some parts of the code seem to be written to support FeeOnTransfer Tokens, other parts do not, which may cause accounting issues

1.3. R - Roles for Vault and Strat are set separately

While a vault can have multiple strategies

A strategy can only ever be used with one vault

For this reason it may be best to have a single set of Role Management Logic, in the Vault and have the strategy ask the vault rather than duplicating storage values, which may desynch

1.4. L - MaxLoss hardcoded means the contract will revert if any loss bigger than BPS happens

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperVaultV2.sol#L55-L56

```
    uint256 public withdrawMaxLoss = 1;
```

Due to the logic handling of losses, and because of the value being hardcoded, the Strategy may be unable to withdraw until the value is changed.

1.5. R - Can refactor to simplify

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/mixins/ReaperAccessControl.sol#L21-L45

```
    function _atLeastRole(bytes32 _role) internal view {
        bytes32[] memory cascadingAccessRoles = _cascadingAccess
        uint256 numRoles = cascadingAccessRoles.length;
        bool specifiedRoleFound = false;
        bool senderHighestRoleFound = false;

        // {_role} must be found in the {cascadingAccessRoles} a
        // Also, msg.sender's highest role index <= specified ro
        for (uint256 i = 0; i < numRoles; i = i.uncheckedInc())
            // If the highest was not found and they have a role
            if (!senderHighestRoleFound && _hasRole(cascadingAcc
                senderHighestRoleFound = true;
            }
            // If we are at the this role part of the loop
            // E.g. we may or may have not found it here
            if (_role == cascadingAccessRoles[i]) {
                specifiedRoleFound = true;
                break;
            }
        }

        // require(senderHighestRoleFound)
        require(specifiedRoleFound && senderHighestRoleFound, "U
    }
```

1.6. R - Track stats via Ninja

By adding a return value via a struct of the form:

```
    struct TokenAmount {
        address token,
        uint256 amount
    }
```

You can track the return value from harvest on a block by block basis, allowing for better monitoring

An example of the dashboard we built:
https://badger-ninja.vercel.app/vault/ethereum/0xBA485b556399123261a5F9c95d413B4f93107407

1.7. R - Track harvest health via Health Check (by Yearn)

Newer versions of Yearn Strategies use an Health Check, this can help prevent an harvest when it could cause a loss.

It may be best to add this to enforce safe operations on-chain.

1.8. R - Consider forking Yearn.Watch

Because the Vault is heavily inspired by YearnV2, you should be able to fork yearn.watch and have it provide automatic reporting

Site: https://yearn.watch/

1.9. NC - `PERCENT_DIVISOR` is more a `BPS_DIVISOR`

The naming could be changed to reflect the unit used

1.10. R - Math for issuing shares can be turned into an internal function

The logic is reused multiple times, the code can be simplified by using an internal pure function

1.11. R - License may be incompatible with source code

```
// SPDX-License-Identifier: BUSL-1.1
```

Because the code is heavily inspired by YearnV2 and derivatives, a stricter license may not be valid

1.12. R - Performance Fee is Paid Out to Treasury Twice

-> On Harvest
https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperVaultV2.sol#L463-L471

```
    function _chargeFees(address strategy, uint256 gain) interna
        uint256 performanceFee = (gain * strategies[strategy].fe
        if (performanceFee != 0) {
            uint256 supply = totalSupply();
            uint256 shares = supply == 0 ? performanceFee : (per
            _mint(treasury, shares);
        }
        return performanceFee;
    }
```

-> On Active Pool `_rebalance`
https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/ActivePool.sol#L291-L294

```
                    vars.treasurySplit = vars.profit.mul(yieldSplitT
                if (vars.treasurySplit != 0) {
                    IERC20(_collateral).safeTransfer(treasuryAdc
                }
```

Disclose this to end users, or consider reducing the split / adding a caller incentive for the keeper to pay for harvests

1.13. NC - `_addLiquidityVelo` is unused

Delete or comment as to why there's an unused function

Strategy

1.14. NC - Steps not validated

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperStrategyGranarySupplyOnly.sol#L160-L171

```
    function setHarvestSteps(address[2][] calldata _newSteps) ex
        _atLeastRole(ADMIN);
        delete steps;

        uint256 numSteps = _newSteps.length;
        for (uint256 i = 0; i < numSteps; i = i.uncheckedInc())
            address[2] memory step = _newSteps[i];
            require(step[0] != address(0));
            require(step[1] != address(0));
            steps.push(step);
        }
    }
```

The steps would allow to sell the want for some other token, which can result in loss of assets as well as loss of yield

A simple check would be to ensure that `step[0]` is never the `want`

1.15. R - Equivalent to aToken.balanceOf(this)

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperStrategyGranarySupplyOnly.sol#L229-L236

```
    function balanceOfPool() public view returns (uint256) {
        (uint256 supply, , , , , , , , ) = IAaveProtocolDataProv
            address(want),
            address(this)
        );
        return supply;
    }
```

The main advantage to sticking to aToken is that you'll keep tracking the old implementation in case of changes

1.16. R - Amount non-zero is known

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperStrategyGranarySupplyOnly.sol#L121-L124

```
                if (amount == 0) {
                    continue;
                }
                _swapVelo(step[0], step[1], amount, VELO_ROUTER);
```

Can be removed from within `_swapVelo`

2. Redemption Helper

2.1. L - Griefing redemptions for a specific collateral by redeeming the other one

2.1.1. Impact

An attacker can grief redemptions for a collateral by redeeming some other collateral, as long as they are willing to pay the fee

2.1.2. POC

Because the fee is the same for all collateral, redeeming on collateral will raise the redemption fee for others, this can be used maliciously to grief others

Because of this, it may be easier to end up getting below MCR, meaning that if the price of the collateral keeps dropping redemptions will be impossible.

Because of this:

- The protocol will receive less than them sum of X partial redemptions
- This can be used to grief the last redeemer

2.1.3. Mitigation Steps

It may be best to have different fees per collateral

Alternatively, the redemption math could be changed to have the caller pay an increasing fee that scales with the amount being redeemed, instead of having the fee grow for the next caller

2.2. R - Could revert earlier if found is address(0)

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/RedemptionHelper.sol#L153

```
    totals.currentBorrower != address(0)
```

3. DefaultPool

3.1. NC - cannot be front-run

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/DefaultPool.sol#L90-L91

```
            IERC20(_collateral).safeIncreaseAllowance(activePool, _a
```

Increasing allowance is not different in this case

It could potentially not work with some collateral also, as you'd need to set approve(0) first

4. CollateralConfig

4.1. L - No check for collateral existence

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/CollateralConfig.sol#L90-L91

```
            Config storage config = collateralConfig[_collateral];
```

Checking for collateral existence is equivalent to an address(0) check, and can avoid mistakes

4.2. NC - 150% CCR can be either too low or too high

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/CollateralConfig.sol#L25-L26

```
        uint256 constant public MIN_ALLOWED_CCR = 1.5 ether; // 150%
```

Consider having custom values per collateral

4.3. R - No check unique

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/CollateralConfig.sol#L53-L57

```
        require(_MCRs.length == _collaterals.length, "Array leng
        require(_CCRs.length == _collaterals.length, "Array leng

        for(uint256 i = 0; i < _collaterals.length; i++) {
            address collateral = _collaterals[i];
```

The lack of validation allows to input the same collateral twice, causing unintended behavior

5. BorrowerOperations

5.1. R - Partially non-CEI conformant

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/BorrowerOperations.sol#L231-L232

```
            IERC20(_collateral).safeTransferFrom(msg.sender, address
```

Since all other calls are from a verified contract, but collateral is not, the code could be refactored to transfer collateral directly to the active pool, as the last operation as to reduce any potential risk.

5.2. NC - Function is called withdraw, but it issues new tokens

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/BorrowerOperations.sol#L235

```
    _withdrawLUSD
```

Consider renaming it

5.3. NC - Can XOR

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/BorrowerOperations.sol#L296-L297

```
            _requireSingularCollChange(_collTopUp, _collWithdrawal);
            _requireNonZeroAdjustment(_collTopUp, _collWithdrawal, _
```

For these two functions, instead of chaning `||` and having two functions, you could use a `XOR` to combine both.

6. ActivePool

6.1. R - Using hardcoded value instead of a constant

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/ActivePool.sol#L258

`10_000` could be turned into a `MAX_BPS` constant to improve refactoring and redability

6.2. R - Approve is fine as it cannot be front-run

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/ActivePool.sol#L279-L280

```
                IERC20(_collateral).safeIncreaseAllowance(yieldGener
```

6.3. NC - Old comment around borrowing fee

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/BorrowerOperations.sol#L195-L196

```
            // ICR is based on the composite debt, i.e. the requeste
```

There’s no borrowing fee, only gas compensation, consider deleting the comment

7. CommunityIssuance

7.1. L - Right after deployment this check will be true

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/LQTY/CommunityIssuance.sol#L86-L87

```
        if (lastIssuanceTimestamp < lastDistributionTime) {
```

Because `lastIssuanceTimestamp` will be 0

7.2. R - CEI Confirmity

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/LQTY/CommunityIssuance.sol#L103-L104

```
        OathToken.transferFrom(msg.sender, address(this), amount
```

To conform to CEI, you could simply edit the function to perform the transfer at the end

7.3. NC - Comments looks out of place

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/LQTY/CommunityIssuance.sol#L23-L36

```
    /* The issuance factor F determines the curvature of the issu
     *
     * Minutes in one year: 60*24*365 = 525600
     *
     * For 50% of remaining tokens issued each year, with minutes
     *
```

```
        * F ** 525600 = 0.5
        *
        * Re-arranging:
        *
        * 525600 * ln(F) = ln(0.5)
        * F = 0.5 ** (1/525600)
        * F = 0.999998681227695000
        */
```

This comment seems to be from liquity and should be deleted

7.4. NC - Capitalization looks off

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/LQTY/CommunityIssuance.sol#L37-L38

```
        IERC20 public OathToken;
```

Can change to: `oathToken`

8. Price Feed

L - The Tellor fix requires constant monitoring

`TellorCaller` was modified to have a 20 minute delay on the value that is being recorded:

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/Dependencies/TellorCaller.sol#L44

```
            (bytes memory data, uint256 timestamp) = getDataBefore(_
```

This should prevent the attack detailed in this [Liquity Disclosure](), however, do note that you'll have to constantly monitor the correctness of the feed and be able to react within 20 minutes to avoid the same attack from happening.

Because the cost of the attack is fairly inexpensive (`$15` I believe), monitoring, as well as setting up infrastructure to dispute the incorrect price will be crucial.

8.1. L - The Tellor price may leak value

Due to the 20 minute delay, the Tellor Price may end up offering too good of a redemption price during time in which the price tanks

It may also prevent liquidations from happening correctly as the delay will make it so that some positions which should be liquidatable will be so only after the extra delay.

8.2. L - Try Catch can still revert due to contract check

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/PriceFeed.sol#L617-L636

```
    try priceAggregator[_collateral].getRoundData(_currentRoundId -
            (
                uint80 roundId,
                int256 answer,
                uint256 /* startedAt */,
                uint256 timestamp,
                uint80 /* answeredInRound */
            )
            {
                // If call to Chainlink succeeds, return the respons
                prevChainlinkResponse.roundId = roundId;
                prevChainlinkResponse.answer = answer;
                prevChainlinkResponse.timestamp = timestamp;
                prevChainlinkResponse.decimals = _currentDecimals;
                prevChainlinkResponse.success = true;
                return prevChainlinkResponse;
            } catch {
                // If call to Chainlink aggregator reverts, return a
                return prevChainlinkResponse;
            }
```

If the aggregator doesn't exist, Solidity will still revert in-spite of the try/catch

8.3. R - Comment different from constants

The code will check for 5% deviation, but the comment refers to 3%

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/PriceFeed.sol#L53-L54

```
    uint constant public MAX_PRICE_DIFFERENCE_BETWEEN_ORACLES =
```

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/PriceFeed.sol#L496-L497

```
        * Return true if the relative price difference is <= 3%:
```

9. LQTY Staking

9.1. NC - Gas cap at around 400 collaterals

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/LQTY/LQTYStaking.sol#L238-L248

```
    function _sendCollGainToUser(address[] memory assets, uint[]
        uint numCollaterals = assets.length;
        for (uint i = 0; i < numCollaterals; i++) {
            if (amounts[i] != 0) {
                address collateral = assets[i];
                emit CollateralSent(msg.sender, collateral, amou
                IERC20(collateral).safeTransfer(msg.sender, amou
            }
        }
    }
```

The Sponsor mentioned a DOS, but ultimately the math will break at 400+ collaterals

Trust (judge) commented:

> Excellent report

0xBebis (Ethos Reserve) confirmed

Gas Optimizations

For this contest, 58 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **c3phas** received the top score from the judge.

The following wardens also submitted reports: Deivitto, 0xsomeone, ddimitrov22, 0xSmartContract, cryptonue, GalloDaSballo, Tomio, Budaghyan, PaludoX0, JCN, abiih, matrix_0wl, atharvasama, hl_, 0x6980, 0xackermann, Rickard, Rageur, ReyAdmirado, dharma09, banky, emmac002, dec3ntraliz3d, kaden, descharre, favelanky, Rolezn, Bough, Viktor_Cortess, RaymondFam, codeislight, Morraez, P-384, LethL, hunter_w3b, RHaO-sec, Bnke0x0, DeFiHackLabs, 0x3b, kodyvim, yamapyblack, MiniGlome, TheSavageTeddy, Madalad, Darshan, Phantasmagoria, 0x73696d616f, scokaf, Saintcode_, pavankv, SaeedAlipoor01988, seeu, Sathish9098, arialblack14, oyc_109, Praise, *and* 0xhacksmithh.

Summary

NB: Some functions have been truncated where necessary to just show affected parts of the code.

Through out the report some places might be denoted with audit tags to show the actual place affected.

Tightly pack storage variables/optimize the order of variable declaration(Total Gas saved: 16K gas)

Here, the storage variables can be tightly packed by putting data type that can fit together next to each other.

Some packing is only possible as the variables involved are timestamps and thus we can reduce their size from uint256 to uint64 safely

LUSDToken.sol: Pack bool mintingPaused with address troveManagerAddress(Saves 1 SLOT: 2k gas)

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/LUSDToken.sol#L37-L71

```
File: /Ethos-Core/contracts/LUSDToken.sol
37:    bool public mintingPaused = false;

51:    bytes32 private immutable _HASHED_NAME;
52:    bytes32 private immutable _HASHED_VERSION;

54:    mapping (address => uint256) private _nonces;

67:    address public troveManagerAddress;
```

```
diff --git a/Ethos-Core/contracts/LUSDToken.sol b/Ethos-Core/cor
index 9c5424d..49f2194 100644
--- a/Ethos-Core/contracts/LUSDToken.sol
+++ b/Ethos-Core/contracts/LUSDToken.sol
@@ -34,7 +34,7 @@ contract LUSDToken is CheckContract, ILUSDToke
     string constant internal _VERSION = "1";
     uint8 constant internal _DECIMALS = 18;

-    bool public mintingPaused = false;
+

     // --- Data for EIP2612 ---

@@ -62,6 +62,8 @@ contract LUSDToken is CheckContract, ILUSDToke
     mapping (address => bool) public troveManagers;
     mapping (address => bool) public stabilityPools;
     mapping (address => bool) public borrowerOperations;
+    bool public mintingPaused = false;
     // simple address variables track current version that can
     // this design makes it so that only the latest version car
     address public troveManagerAddress;
```

ReaperVaultV2.sol: pack treasury with emergencyShutdown and lastReport(4k gas)

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperVaultV2.sol#L46-L78

lastReport being a timestamp we can get away with using uint64 - should be safe for around 530 years(Following this direction we can save 2 SLOTS : 4k gas)

If not willing to to reduce the timestamp variable we can pack `address treasury` and `bool emergencyShutdown` saving 1 SLOT: 2k gas

```
File: /Ethos-Vault/contracts/ReaperVaultV2.sol
46:    uint256 public lastReport;

48:    uint256 public immutable constructionTime;
49:    bool public emergencyShutdown;

57:    uint256 public lockedProfit;

78:    address public treasury;
```

```
diff --git a/Ethos-Vault/contracts/ReaperVaultV2.sol b/Ethos-Vau
index b5f2a58..e462619 100644
--- a/Ethos-Vault/contracts/ReaperVaultV2.sol
+++ b/Ethos-Vault/contracts/ReaperVaultV2.sol
@@ -43,10 +43,12 @@ contract ReaperVaultV2 is ReaperAccessContro

     uint256 public totalAllocBPS; // Sum of allocBPS across all
     uint256 public totalAllocated; // Amount of tokens that hav
-    uint256 public lastReport; // block.timestamp of last repor
+    uint64 public lastReport; // block.timestamp of last report
+    bool public emergencyShutdown;
+    address public treasury; // address to whom performance fee

     uint256 public immutable constructionTime;
-    bool public emergencyShutdown;
+

     // The token the vault accepts and looks to maximize.
     IERC20Metadata public immutable token;
@@ -75,7 +77,6 @@ contract ReaperVaultV2 is ReaperAccessControl,
     bytes32 public constant GUARDIAN = keccak256("GUARDIAN");
     bytes32 public constant ADMIN = keccak256("ADMIN");

-    address public treasury; // address to whom performance fee
```

Save 1 SLOT(2k gas): Pack lastFeeOperationTime with owner

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/TroveManager.sol#L21-L66

lastFeeOperationTime is a timestamps thus we can use uint64 which would be safe for around 532 years

```
File: /Ethos-Core/contracts/TroveManager.sol
21:    address public owner;

65:    // The timestamp of the latest fee operation (redemption
66:    uint public lastFeeOperationTime;
```

```
diff --git a/Ethos-Core/contracts/TroveManager.sol b/Ethos-Core/
index 6f587d9..38da844 100644
--- a/Ethos-Core/contracts/TroveManager.sol
+++ b/Ethos-Core/contracts/TroveManager.sol
@@ -20,6 +20,9 @@ contract TroveManager is LiquityBase, /*Ownabl

     address public owner;

+        // The timestamp of the latest fee operation (redemptic
+    uint64 public lastFeeOperationTime;
+
     // --- Connected contract declarations ---

     address public borrowerOperationsAddress;
```

Save 2 SLOTS(4k gas): see description

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/LQTY/CommunityIssuance.sol#L21-L46 lastDistributionTime and lastIssuanceTimestamp are timestamps thus we can use uint64 which would be safe for around 532 years

```
File: /Ethos-Core/contracts/LQTY/CommunityIssuance.sol
21:    bool public initialized = false;

37:    IERC20 public OathToken;

39:    address public stabilityPoolAddress;

41:    uint public totalOATHIssued;
42:    uint public lastDistributionTime;
43:    uint public distributionPeriod;
44:    uint public rewardPerSecond;

46:    uint public lastIssuanceTimestamp;
```

```
diff --git a/Ethos-Core/contracts/LQTY/CommunityIssuance.sol b/E
index c79adfe..e7f21ff 100644
--- a/Ethos-Core/contracts/LQTY/CommunityIssuance.sol
+++ b/Ethos-Core/contracts/LQTY/CommunityIssuance.sol
@@ -19,6 +19,7 @@ contract CommunityIssuance is ICommunityIssuar
     string constant public NAME = "CommunityIssuance";

     bool public initialized = false;
+    uint64 public lastDistributionTime;

    /* The issuance factor F determines the curvature of the iss
     *
@@ -37,13 +38,14 @@ contract CommunityIssuance is ICommunityIssu
     IERC20 public OathToken;

     address public stabilityPoolAddress;
+    uint64 public lastIssuanceTimestamp;

     uint public totalOATHIssued;
-    uint public lastDistributionTime;
+
     uint public distributionPeriod;
     uint public rewardPerSecond;

-    uint public lastIssuanceTimestamp;
```

```
+
```

Save 2 SLOTS(4k gas): Pack lastHarvestTimestamp and upgradeProposalTime

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/abstract/ReaperBaseStrategyv4.sol#L28-L58

lastHarvestTimestamp and upgradeProposalTime are timestamps thus we can use uint64 which would be safe for around 532 years

```
File: /Ethos-Vault/contracts/abstract/ReaperBaseStrategyv4.sol
28:    address public want;

30:    bool public emergencyExit;
31:    uint256 public lastHarvestTimestamp;

33:    uint256 public upgradeProposalTime;

58:    address public vault;
```

```diff
diff --git a/Ethos-Vault/contracts/abstract/ReaperBaseStrategyv4
index db020f2..a824d97 100644
--- a/Ethos-Vault/contracts/abstract/ReaperBaseStrategyv4.sol
+++ b/Ethos-Vault/contracts/abstract/ReaperBaseStrategyv4.sol
@@ -28,9 +28,9 @@ abstract contract ReaperBaseStrategyv4 is
     address public want;

     bool public emergencyExit;
-    uint256 public lastHarvestTimestamp;
+    uint64 public lastHarvestTimestamp;

-    uint256 public upgradeProposalTime;
+    uint64 public upgradeProposalTime;
```

Pack structs by putting data types that fit together next to each other(Saves: 2K gas)

As the solidity EVM works with 32 bytes, variables less than 32 bytes should be packed inside a struct so that they can be stored in the same slot, this saves gas when writing to storage ~20000 gas

ReaperVaultV2.sol: Pack activation together with lastReport(Save 1 SLOT: 2k gas)

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperVaultV2.sol#L25-L33

activation and lastReport are timestamps thus changing them to uint64 should be safe for around 532 years

```
File: /Ethos-Vault/contracts/ReaperVaultV2.sol
25:    struct StrategyParams {
26:        uint256 activation; // Activation block.timestamp
27:        uint256 feeBPS; // Performance fee taken from profit,
28:        uint256 allocBPS; // Allocation in BPS of vault's tot
29:        uint256 allocated; // Amount of capital allocated to
30:        uint256 gains; // Total returns that Strategy has rea
31:        uint256 losses; // Total losses that Strategy has rea
32:        uint256 lastReport; // block.timestamp of the last ti
33:    }
```

```diff
diff --git a/Ethos-Vault/contracts/ReaperVaultV2.sol b/Ethos-Vau
index b5f2a58..44f84fb 100644
--- a/Ethos-Vault/contracts/ReaperVaultV2.sol
+++ b/Ethos-Vault/contracts/ReaperVaultV2.sol
@@ -23,13 +23,14 @@ contract ReaperVaultV2 is ReaperAccessContro
     using SafeERC20 for IERC20Metadata;

     struct StrategyParams {
-        uint256 activation; // Activation block.timestamp
+        uint64 activation; // Activation block.timestamp
+        uint64 lastReport; // block.timestamp of the last time
         uint256 feeBPS; // Performance fee taken from profit, i
         uint256 allocBPS; // Allocation in BPS of vault's total
         uint256 allocated; // Amount of capital allocated to th
         uint256 gains; // Total returns that Strategy has reali
         uint256 losses; // Total losses that Strategy has reali
-         uint256 lastReport; // block.timestamp of the last time
+
      }
```

Emitting storage values instead of the memory one.

Here, the values emitted shouldn't be read from storage. The existing memory values should be used instead:

Emit `_newTvlCap` instead of `tvlCap`

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperVaultV2.sol#L578-L582

```
File: /Ethos-Vault/contracts/ReaperVaultV2.sol
578:     function updateTvlCap(uint256 _newTvlCap) public {
579:         _atLeastRole(ADMIN);
580:         tvlCap = _newTvlCap;
581:         emit TvlCapUpdated(tvlCap);
582:     }
```

```
diff --git a/Ethos-Vault/contracts/ReaperVaultV2.sol b/Ethos-Vau
index b5f2a58..c12e679 100644
--- a/Ethos-Vault/contracts/ReaperVaultV2.sol
+++ b/Ethos-Vault/contracts/ReaperVaultV2.sol
@@ -578,7 +578,7 @@ contract ReaperVaultV2 is ReaperAccessContro
      function updateTvlCap(uint256 _newTvlCap) public {
          _atLeastRole(ADMIN);
          tvlCap = _newTvlCap;
-         emit TvlCapUpdated(tvlCap);
+         emit TvlCapUpdated(_newTvlCap);
      }
```

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/StabilityPool.sol#L575-L579

```
File: /Ethos-Core/contracts/StabilityPool.sol
575:        if (newProductFactor == 0) {
576:            currentEpoch = currentEpochCached.add(1);
577:            emit EpochUpdated(currentEpoch);
578:            currentScale = 0;
579:            emit ScaleUpdated(currentScale);
```

As currentScale is being assigned value 0, we should just emit the 0 here

```
diff --git a/Ethos-Core/contracts/StabilityPool.sol b/Ethos-Core
index db163b7..62c0042 100644
--- a/Ethos-Core/contracts/StabilityPool.sol
+++ b/Ethos-Core/contracts/StabilityPool.sol
@@ -576,7 +576,7 @@ contract StabilityPool is LiquityBase, Owna
             currentEpoch = currentEpochCached.add(1);
             emit EpochUpdated(currentEpoch);
             currentScale = 0;
-            emit ScaleUpdated(currentScale);
+            emit ScaleUpdated(0);
             newP = DECIMAL_PRECISION;
```

IF’s/require() statements that check input arguments should be at the top of the function

Checks that involve constants should come before checks that involve state variables, function calls, and calculations. By doing these checks first, the function is able to revert before wasting a Gcoldsload (2100 gas) in a function that may ultimately revert in the unhappy case.

Reorder the require statement to have cheaper one before the gas consuming one

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/CollateralConfig.sol#L46-L54

```
File: /Ethos-Core/contracts/CollateralConfig.sol
46:    function initialize(
47:        address[] calldata _collaterals,
48:         uint256[] calldata _MCRs,
49:         uint256[] calldata _CCRs
50:     ) external override onlyOwner {
51:         require(!initialized, "Can only initialize once");
52:         require(_collaterals.length != 0, "At least one colla
53:         require(_MCRs.length == _collaterals.length, "Array l
54:         require(_CCRs.length == _collaterals.length, "Array l
```

The first check `require(!initialized, "Can only initialize once");` involves reading from the state as `initialized` is a storage variable which is quite expensive. As we would also revert on the other conditions which involves reading function arguments(cheaper) we should reorder the checks as follows

```
diff --git a/Ethos-Core/contracts/CollateralConfig.sol b/Ethos-C
index d524f23..1458836 100644
--- a/Ethos-Core/contracts/CollateralConfig.sol
+++ b/Ethos-Core/contracts/CollateralConfig.sol
@@ -48,10 +48,10 @@ contract CollateralConfig is ICollateralConf
         uint256[] calldata _MCRs,
         uint256[] calldata _CCRs
     ) external override onlyOwner {
-        require(!initialized, "Can only initialize once");
         require(_collaterals.length != 0, "At least one collate
         require(_MCRs.length == _collaterals.length, "Array len
         require(_CCRs.length == _collaterals.length, "Array len
+        require(!initialized, "Can only initialize once");

         for(uint256 i = 0; i < _collaterals.length; i++) {
             address collateral = _collaterals[i];
```

Cheaper to check function argument against a constant as compared to checking it against a storaga variable

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/CollateralConfig.sol#L85-L100

```
File: /Ethos-Core/contracts/CollateralConfig.sol
85:     function updateCollateralRatios(
86:         address _collateral,
        uint256 _MCR,
        uint256 _CCR
    ) external onlyOwner checkCollateral(_collateral) {
        Config storage config = collateralConfig[_collateral]
        require(_MCR <= config.MCR, "Can only walk down the 
        require(_CCR <= config.CCR, "Can only walk down the 

        require(_MCR >= MIN_ALLOWED_MCR, "MCR below allowed 
        config.MCR = _MCR;

        require(_CCR >= MIN_ALLOWED_CCR, "CCR below allowed 
        config.CCR = _CCR;
        emit CollateralRatiosUpdated(_collateral, _MCR, _CCR)
    }
```

In the above, we have two types of checks, one that involves comparing a function argument to a storage variable `config` and another that compares function argument to a constant. It's obviously cheaper to read a constant as compared to the storage one, thus should have checks for constant come before the one for storage

We can also save alot of gas if we revert before running the line `Config storage config = collateralConfig[_collateral];` as it involves an SSTORE

```
diff --git a/Ethos-Core/contracts/CollateralConfig.sol b/Ethos-
index d524f23..f71570d 100644
--- a/Ethos-Core/contracts/CollateralConfig.sol
+++ b/Ethos-Core/contracts/CollateralConfig.sol
@@ -87,15 +87,17 @@ contract CollateralConfig is ICollateralCon
         uint256 _MCR,
         uint256 _CCR
     ) external onlyOwner checkCollateral(_collateral) {
+        require(_MCR >= MIN_ALLOWED_MCR, "MCR below allowed mi
+        require(_CCR >= MIN_ALLOWED_CCR, "CCR below allowed mi
+
         Config storage config = collateralConfig[_collateral];
+
         require(_MCR <= config.MCR, "Can only walk down the MC
         require(_CCR <= config.CCR, "Can only walk down the CC

-        require(_MCR >= MIN_ALLOWED_MCR, "MCR below allowed mi
         config.MCR = _MCR;
```

```
-
-           require(_CCR >= MIN_ALLOWED_CCR, "CCR below allowed mir
            config.CCR = _CCR;
+
            emit CollateralRatiosUpdated(_collateral, _MCR, _CCR);
        }
```

Reorder the require statements to have the cheaper one before the gas consuming one

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/ActivePool.sol#L71-L94

```
File: /Ethos-Core/contracts/ActivePool.sol
71:     function setAddresses(

78:         address _treasuryAddress,

81:     )
82:         external
83:         onlyOwner
84:     {
85:         require(!addressesSet, "Can call setAddresses only or


87:         checkContract(_collateralConfigAddress);
88:         checkContract(_borrowerOperationsAddress);
89:         checkContract(_troveManagerAddress);
90:         checkContract(_stabilityPoolAddress);
91:         checkContract(_defaultPoolAddress);
92:         checkContract(_collSurplusPoolAddress);
93:         require(_treasuryAddress != address(0), "Treasury car
94:         checkContract(_lqtyStakingAddress);
```

The check on Line 93 would cause our function to revert and since it involves a check for a function parameter, it would be cheaper to have it come before any other operation. In this case we have another check that involves reading from storage which is expensive. Fail early and cheaply

```
diff --git a/Ethos-Core/contracts/ActivePool.sol b/Ethos-Core/cc
```

```
index 753fcd0..a6903af 100644
--- a/Ethos-Core/contracts/ActivePool.sol
+++ b/Ethos-Core/contracts/ActivePool.sol
@@ -82,6 +82,8 @@ contract ActivePool is Ownable, CheckContract,
         external
         onlyOwner
     {
+        require(_treasuryAddress != address(0), "Treasury cannc
+
         require(!addressesSet, "Can call setAddresses only once

         checkContract(_collateralConfigAddress);
@@ -90,7 +92,6 @@ contract ActivePool is Ownable, CheckContract,
         checkContract(_stabilityPoolAddress);
         checkContract(_defaultPoolAddress);
         checkContract(_collSurplusPoolAddress);
-        require(_treasuryAddress != address(0), "Treasury cannc
         checkContract(_lqtyStakingAddress);

         collateralConfigAddress = _collateralConfigAddress;
```

Reorder the checks to avoid wasting gas checking computing external functionality

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/ActivePool.sol#L125-L130

```
File: /Ethos-Core/contracts/ActivePool.sol
125:    function setYieldingPercentage(address _collateral, uint
126:        _requireValidCollateralAddress(_collateral);
127:        require(_bps <= 10_000, "Invalid BPS value");
128:        yieldingPercentage[_collateral] = _bps;
129:        emit YieldingPercentageUpdated(_collateral, _bps);
130:    }
```

If we end up reverting on the Line `require(_bps <= 10_000, "Invalid BPS value");` we would end up wasting alot of gas on the line `_requireValidCollateralAddress(_collateral);` . The later is a function call and the function has the following implementation

```
    function _requireValidCollateralAddress(address _collateral)
        require(
            ICollateralConfig(collateralConfigAddress).isCollate
            "Invalid collateral address"
        );
    }
```

Note in the above function, the require statement involves an external function call which is pretty expensive.

Recommend to implement the following changes so that incase of an early revert on the `function parameter check` we would not waste gas

```
diff --git a/Ethos-Core/contracts/ActivePool.sol b/Ethos-Core/cc
index 753fcd0..d83ee3a 100644
--- a/Ethos-Core/contracts/ActivePool.sol
+++ b/Ethos-Core/contracts/ActivePool.sol
@@ -123,8 +123,8 @@ contract ActivePool is Ownable, CheckContrac
     }

     function setYieldingPercentage(address _collateral, uint256
+         require(_bps <= 10_000, "Invalid BPS value");
         _requireValidCollateralAddress(_collateral);
-        require(_bps <= 10_000, "Invalid BPS value");
         yieldingPercentage[_collateral] = _bps;
         emit YieldingPercentageUpdated(_collateral, _bps);
     }
```

Reorder the require statement to have the cheaper one come first

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperVaultV2.sol#L144-L156

```
File: /Ethos-Vault/contracts/ReaperVaultV2.sol
144:    function addStrategy(
145:        address _strategy,
146:        uint256 _feeBPS,
147:        uint256 _allocBPS
148:    ) external {
```

```
149:        _atLeastRole(DEFAULT_ADMIN_ROLE);
150:        require(!emergencyShutdown, "Cannot add strategy du
151:        require(_strategy != address(0), "Invalid strategy a
152:        require(strategies[_strategy].activation == 0, "Stra
153:        require(address(this) == IStrategy(_strategy).vault(
154:        require(address(token) == IStrategy(_strategy).want(
155:        require(_feeBPS <= PERCENT_DIVISOR / 5, "Fee cannot
156:        require(_allocBPS + totalAllocBPS <= PERCENT_DIVISO
```

Refactor the code as follows to have the cheaper check first

```
diff --git a/Ethos-Vault/contracts/ReaperVaultV2.sol b/Ethos-Va
index b5f2a58..48316b9 100644
--- a/Ethos-Vault/contracts/ReaperVaultV2.sol
+++ b/Ethos-Vault/contracts/ReaperVaultV2.sol
@@ -146,13 +146,15 @@ contract ReaperVaultV2 is ReaperAccessCont
         uint256 _feeBPS,
         uint256 _allocBPS
     ) external {
+        require(_strategy != address(0), "Invalid strategy addr
+        require(_feeBPS <= PERCENT_DIVISOR / 5, "Fee cannot be
         _atLeastRole(DEFAULT_ADMIN_ROLE);
         require(!emergencyShutdown, "Cannot add strategy during
-        require(_strategy != address(0), "Invalid strategy addr
+
         require(strategies[_strategy].activation == 0, "Strateg
         require(address(this) == IStrategy(_strategy).vault(),
         require(address(token) == IStrategy(_strategy).want(),
-        require(_feeBPS <= PERCENT_DIVISOR / 5, "Fee cannot be
+
         require(_allocBPS + totalAllocBPS <= PERCENT_DIVISOR, '
```

Move the cheaper require check on top

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperVaultV2.sol#L178-L184

```
File: /Ethos-Vault/contracts/ReaperVaultV2.sol
178:    function updateStrategyFeeBPS(address _strategy, uint25
179:        _atLeastRole(ADMIN);
```

```
180:        require(strategies[_strategy].activation != 0, "Inva
181:        require(_feeBPS <= PERCENT_DIVISOR / 5, "Fee cannot 
182:        strategies[_strategy].feeBPS = _feeBPS;
183:        emit StrategyFeeBPSUpdated(_strategy, _feeBPS);
184:    }
```

The require on Line 181 involves checking a function argument which is cheaper compared to it's storage counterpart which is being checked on line 180. Refactor the code as follows to have the cheaper check first

```
diff --git a/Ethos-Vault/contracts/ReaperVaultV2.sol b/Ethos-Vau
index b5f2a58..8532ded 100644
--- a/Ethos-Vault/contracts/ReaperVaultV2.sol
+++ b/Ethos-Vault/contracts/ReaperVaultV2.sol
@@ -176,9 +176,10 @@ contract ReaperVaultV2 is ReaperAccessContr
       * @param _feeBPS The new performance fee in basis points.
       */
      function updateStrategyFeeBPS(address _strategy, uint256 _f
+        require(_feeBPS <= PERCENT_DIVISOR / 5, "Fee cannot be 
         _atLeastRole(ADMIN);
         require(strategies[_strategy].activation != 0, "Invalic
-        require(_feeBPS <= PERCENT_DIVISOR / 5, "Fee cannot be 
```

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperVaultV2.sol#L319-L338

```
File: /Ethos-Vault/contracts/ReaperVaultV2.sol
319:    function _deposit(uint256 _amount, address _receiver) ir
320:        _atLeastRole(DEPOSITOR);
321:        require(!emergencyShutdown, "Cannot deposit during e
322:        require(_amount != 0, "Invalid amount");
```

The require on Line 322 involves checking a function argument which is cheaper compared to it's storage counterpart which is being checked on line 321. Refactor the code as follows to have the cheaper check first

```
diff --git a/Ethos-Vault/contracts/ReaperVaultV2.sol b/Ethos-Vau
```

```
index b5f2a58..ab8c6c3 100644
--- a/Ethos-Vault/contracts/ReaperVaultV2.sol
+++ b/Ethos-Vault/contracts/ReaperVaultV2.sol
@@ -317,9 +317,10 @@ contract ReaperVaultV2 is ReaperAccessContr
     // Internal helper function to deposit {_amount} of assets
     // shares to {_receiver}. Returns the number of shares that
     function _deposit(uint256 _amount, address _receiver) inter
+        require(_amount != 0, "Invalid amount");
         _atLeastRole(DEPOSITOR);
         require(!emergencyShutdown, "Cannot deposit during emer
-        require(_amount != 0, "Invalid amount");
+
```

Move the require statement at the beginning of the function

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/abstract/ReaperBaseStrategyv4.sol#L63-L88

```
File: /Ethos-Vault/contracts/abstract/ReaperBaseStrategyv4.sol
63:     function __ReaperBaseStrategy_init(
64:         address _vault,
65:         address _want,
66:         address[] memory _strategists,
67:         address[] memory _multisigRoles
68:     ) internal onlyInitializing {
69:         __UUPSUpgradeable_init();
70:         __AccessControlEnumerable_init();

72:         vault = _vault;
73:         want = _want;
74:         IERC20Upgradeable(want).safeApprove(vault, type(uint2

76:         uint256 numStrategists = _strategists.length;
77:         for (uint256 i = 0; i < numStrategists; i = i.uncheck
78:             _grantRole(STRATEGIST, _strategists[i]);
79:         }

81:         require(_multisigRoles.length == 3, "Invalid number c
```

As we might revert if a function argument check does not succeed, it is better to check the argument first before performing any other operations as we might waste gas doing this operations then revert on failure to meet the conditions required for the function argument

```
diff --git a/Ethos-Vault/contracts/abstract/ReaperBaseStrategyv4
index db020f2..7c3c5a8 100644
--- a/Ethos-Vault/contracts/abstract/ReaperBaseStrategyv4.sol
+++ b/Ethos-Vault/contracts/abstract/ReaperBaseStrategyv4.sol
@@ -68,6 +68,8 @@ abstract contract ReaperBaseStrategyv4 is
     ) internal onlyInitializing {
         __UUPSUpgradeable_init();
         __AccessControlEnumerable_init();
+        require(_multisigRoles.length == 3, "Invalid number of
+

         vault = _vault;
         want = _want;
@@ -78,7 +80,6 @@ abstract contract ReaperBaseStrategyv4 is
             _grantRole(STRATEGIST, _strategists[i]);
         }

-        require(_multisigRoles.length == 3, "Invalid number of
```

Reorder the require statement here to have the cheaper one before the gas consuming one

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/abstract/ReaperBaseStrategyv4.sol#L95-L103

```
File: /Ethos-Vault/contracts/abstract/ReaperBaseStrategyv4.sol
95:     function withdraw(uint256 _amount) external override retu
96:         require(msg.sender == vault, "Only vault can withdraw
97:         require(_amount != 0, "Amount cannot be zero");
98:         require(_amount <= balanceOf(), "Ammount must be less
```

Cheaper checks should come before the more expensive ones.

```
diff --git a/Ethos-Vault/contracts/abstract/ReaperBaseStrategyv4
index db020f2..71504f1 100644
--- a/Ethos-Vault/contracts/abstract/ReaperBaseStrategyv4.sol
```

```diff
+++ b/Ethos-Vault/contracts/abstract/ReaperBaseStrategyv4.sol
@@ -93,8 +93,8 @@ abstract contract ReaperBaseStrategyv4 is
      *      is deducted up-front.
      */
     function withdraw(uint256 _amount) external override returr
-        require(msg.sender == vault, "Only vault can withdraw")
         require(_amount != 0, "Amount cannot be zero");
+        require(msg.sender == vault, "Only vault can withdraw")
         require(_amount <= balanceOf(), "Ammount must be less t
```

The result of a function call should be cached rather than re-calling the function

External calls are expensive. Consider caching the following:

ReaperVaultV2.sol._deposit(): Results of totalSupply() should be cached(saves 1 call on sad path)

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperVaultV2.sol#L319-L338

```
File: /Ethos-Vault/contracts/ReaperVaultV2.sol
319:    function _deposit(uint256 _amount, address _receiver) ir

331:        if (totalSupply() == 0) {
332:            shares = _amount;
333:        } else {
334:            shares = (_amount * totalSupply()) / freeFunds;
335:        }
```

ReaperVaultERC4626.sol.convertToShares(): Results of totalSupply() and _freeFunds() should be cached

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperVaultERC4626.sol#L51-L54

```
File: /Ethos-Vault/contracts/ReaperVaultERC4626.sol
51:    function convertToShares(uint256 assets) public view over
52:        if (totalSupply() == 0 || _freeFunds() == 0) return a
```

```
53:         return (assets * totalSupply()) / _freeFunds();
54:     }
```

ReaperVaultERC4626.sol.convertToAssets(): Results of totalSupply() should be cached

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperVaultERC4626.sol#L66-L69

```
File: /Ethos-Vault/contracts/ReaperVaultERC4626.sol
66:     function convertToAssets(uint256 shares) public view over
67:         if (totalSupply() == 0) return shares;
68:         return (shares * _freeFunds()) / totalSupply();
69:     }
```

ReaperVaultERC4626.sol.maxDeposit(): balance() should be cached

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperVaultERC4626.sol#L79-L83

```
File: /Ethos-Vault/contracts/ReaperVaultERC4626.sol
79:     function maxDeposit(address) external view override retur
80:         if (emergencyShutdown || balance() >= tvlCap) return
81:         if (tvlCap == type(uint256).max) return type(uint256)
82:         return tvlCap - balance();
83:     }
```

ReaperVaultERC4626.sol.maxMint(): balance() should be cached

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperVaultERC4626.sol#L122-L126

```
File: /Ethos-Vault/contracts/ReaperVaultERC4626.sol
122:     function maxMint(address) external view override return
123:         if (emergencyShutdown || balance() >= tvlCap) retur
```

```
124:         if (tvlCap == type(uint256).max) return type(uint25
125:         return convertToShares(tvlCap - balance());
126:     }
```

ReaperVaultERC4626.sol.previewMint(): totalSupply() should be cached

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperVaultERC4626.sol#L139-L142

```
File: /Ethos-Vault/contracts/ReaperVaultERC4626.sol
139:     function previewMint(uint256 shares) public view overri
140:         if (totalSupply() == 0) return shares;
141:         assets = roundUpDiv(shares * _freeFunds(), totalSup
142:     }
```

ReaperVaultERC4626.sol.previewWithdraw(): results of totalSupply() and _freeFunds() should be cached

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperVaultERC4626.sol#L183-L186

```
File: /Ethos-Vault/contracts/ReaperVaultERC4626.sol
183:     function previewWithdraw(uint256 assets) public view ove
184:         if (totalSupply() == 0 || _freeFunds() == 0) return
185:        shares = roundUpDiv(assets * totalSupply(), _freeFun
186:     }
```

Internal/Private functions only called once can be inlined to save gas

Not inlining costs 20 to 40 gas because of two extra JUMP instructions and additional stack operations needed for function calls.

Affected code:
https://github.com/code-423n4/2023-02-

ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/BorrowerOperations.sol#L438-L445

```
File: /Ethos-Core/contracts/BorrowerOperations.sol
438:    function _getCollChange(
439:        uint _collReceived,
440:        uint _requestedCollWithdrawal
441:    )
442:        internal
443:        pure
444:        returns(uint collChange, bool isCollIncrease)
445:    {


455:    function _updateTroveFromAdjustment
456:    (
457:        ITroveManager _troveManager,
458:        address _borrower,
459:        address _collateral,
460:        uint _collChange,
461:        bool _isCollIncrease,
462:        uint _debtChange,
463:        bool _isDebtIncrease
464:    )
465:        internal
466:        returns (uint, uint)
467:    {

476:    function _moveTokensAndCollateralfromAdjustment
477:    (
478:        IActivePool _activePool,
479:        ILUSDToken _lusdToken,
480:        address _borrower,
481:        address _collateral,
482:        uint _collChange,
483:        bool _isCollIncrease,
484:        uint _LUSDChange,
485:        bool _isDebtIncrease,
486:        uint _netDebtChange
487:    )
488:        internal
489:    {


533:    function _requireSingularCollChange(uint _collTopUp, ui
537:    function _requireNonZeroAdjustment(uint _collTopUp, uint

546:    function _requireTroveisNotActive(ITroveManager _troveMa

551:    function _requireNonZeroDebtChange(uint _LUSDChange) int

555:    function _requireNotInRecoveryMode(
556:        address _collateral,
557:        uint _price,
558:        uint256 _CCR,
559:        uint256 _collateralDecimals
560:    ) internal view {

567:    function _requireNoCollWithdrawal(uint _collWithdrawal)

571:    function _requireValidAdjustmentInCurrentMode
572:    (
573:        bool _isRecoveryMode,
574:        address _collateral,
575:        uint _collWithdrawal,
576:        bool _isDebtIncrease,
577:        LocalVariables_adjustTrove memory _vars
578:    )
579:        internal
580:        view
581:    {

624:    function _requireNewICRisAboveOldICR(uint _newICR, uint

636:    function _requireValidLUSDRepayment(uint _currentDebt, u

640:    function _requireCallerIsStabilityPool() internal view {

661:    function _getNewNominalICRFromTroveChange

674:    {
```

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/TroveManager.sol#L478-L489

```
File: /Ethos-Core/contracts/TroveManager.sol
```

```
478:    function _getCappedOffsetVals

582:    function _getTotalsFromLiquidateTrovesSequence_RecoveryM

671:    function _getTotalsFromLiquidateTrovesSequence_NormalMoc

785:    function _getTotalFromBatchLiquidate_RecoveryMode

864:    function _getTotalsFromBatchLiquidate_NormalMode

1213:    function _computeNewStake(address _collateral, uint _cc

1321:    function _addTroveOwnerToArray(address _borrower, addre

1538:    function _requireMoreThanOneTroveInSystem(uint TroveOwr
```

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/ActivePool.sol#L320

```
File: /Ethos-Core/contracts/ActivePool.sol
320:    function _requireCallerIsBorrowerOperationsOrDefaultPool
```

Multiple accesses of a mapping/array should use a local variable cache

Caching a mapping's value in a local storage or calldata variable when the value is accessed multiple times saves ~42 gas per access due to not having to perform the same offset calculation every time.

Help the Optimizer by saving a storage variable's reference instead of repeatedly fetching it

To help the optimizer,declare a storage type variable and use it instead of repeatedly fetching the reference in a map or an array.

As an example, instead of repeatedly calling `someMap[someIndex]` , save its reference like this: `SomeStruct storage someStruct = someMap[someIndex]` and use it.

TroveManager.sol.updateDebtAndCollAndStakesPostRedemption(): Troves[_borrower][_collateral] should be cached in local storage

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/TroveManager.sol#L962-L980

```
File: /Ethos-Core/contracts/TroveManager.sol
962:    function updateDebtAndCollAndStakesPostRedemption(

967:    ) external override {
968:        _requireCallerIsRedemptionHelper();
969:        Troves[_borrower][_collateral].debt = _newDebt;//@au
970:        Troves[_borrower][_collateral].coll = _newColl;//@au
971:        _updateStakeAndTotalStakes(_borrower, _collateral);

973:        emit TroveUpdated(

977:            Troves[_borrower][_collateral].stake,//@audit:
978:            TroveManagerOperation.redeemCollateral
979:        );
980:    }
```

TroveManager.sol._getCurrentTroveAmounts(): Troves[_borrower][_collateral] should be cached in local storage

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/TroveManager.sol#L1066-L1074

```
File: /Ethos-Core/contracts/TroveManager.sol
1066:    function _getCurrentTroveAmounts(address _borrower, ad

1070:    uint currentCollateral = Troves[_borrower][_collateral]
1071:        uint currentLUSDDebt = Troves[_borrower][_collatera
```

TroveManager.sol._applyPendingRewards(): Troves[_borrower][_collateral] should be cached in local storage

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-

[Core/contracts/TroveManager.sol#L1082-L1108]()

```
File: /Ethos-Core/contracts/TroveManager.sol
1082:    function _applyPendingRewards(IActivePool _activePool,

1091:                Troves[_borrower][_collateral].coll = Troves[_b
1092:                Troves[_borrower][_collateral].debt = Troves[_b


1099:                emit TroveUpdated(
1100:                    _borrower,
1101:                    _collateral,
1102:                    Troves[_borrower][_collateral].debt,//@audi
1103:                    Troves[_borrower][_collateral].coll,//@audi
1104:                    Troves[_borrower][_collateral].stake,//@auc
1105:                    TroveManagerOperation.applyPendingRewards
1106:                );
1107:            }
1108:    }
```

TroveManager.sol._updateTroveRewardSnapshots(): rewardSnapshots[_borrower][_collateral], L*Collateral*[\collateral] and L*LUSDDebt*[\collateral] should be cached

[https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/TroveManager.sol#L1116-L1120]()

```
File: /Ethos-Core/contracts/TroveManager.sol
1116:    function _updateTroveRewardSnapshots(address _borrower,
1117:        rewardSnapshots[_borrower][_collateral].collAmount
1118:        rewardSnapshots[_borrower][_collateral].LUSDDebt =
1119:        emit TroveSnapshotsUpdated(_collateral, L_Collatera
1120:    }
```

TroveManager.sol.getPendingCollateralReward(): Troves[_borrower][_collateral] should be cached in local storage

[https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/TroveManager.sol#L1123-L1135]()

```
File: /Ethos-Core/contracts/TroveManager.sol
1123:    function getPendingCollateralReward(address _borrower,

1127:        if ( rewardPerUnitStaked == 0 || Troves[_borrower]|

1129:        uint stake = Troves[_borrower][_collateral].stake;/
```

TroveManager.sol.getPendingLUSDDebtReward(): Troves[_borrower][_collateral] should be cached in local storage

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/TroveManager.sol#L1138-L1150

```
File: /Ethos-Core/contracts/TroveManager.sol
1138:    function getPendingLUSDDebtReward(address _borrower, ac

1142:        if ( rewardPerUnitStaked == 0 || Troves[_borrower]|

1144:        uint stake = Troves[_borrower][_collateral].stake;/
```

TroveManager.sol.getEntireDebtAndColl(): Troves[_borrower][_collateral] should be cached in local storage

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/TroveManager.sol#L1164-L1181

```
File: /Ethos-Core/contracts/TroveManager.sol
1164:    function getEntireDebtAndColl(

1173:        debt = Troves[_borrower][_collateral].debt;//@audit
1174:       coll = Troves[_borrower][_collateral].coll;//@audit:
```

TroveManager.sol._removeStake(): Troves[_borrower][_collateral] should be cached in local storage

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-

Core/contracts/TroveManager.sol#L1189-L1193

```
File: /Ethos-Core/contracts/TroveManager.sol
1189:     function _removeStake(address _borrower, address _colla
1190:         uint stake = Troves[_borrower][_collateral].stake;
1191:         totalStakes[_collateral] = totalStakes[_collateral]
1192:         Troves[_borrower][_collateral].stake = 0;
1193:     }
```

Cache Troves[_borrower][_collateral] and totalStakes[_collateral] in local storage

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/TroveManager.sol#L1201-L1210

```
File: /Ethos-Core/contracts/TroveManager.sol
1201:     function _updateStakeAndTotalStakes(address _borrower,
1202:         uint newStake = _computeNewStake(_collateral, Trove
1203:         uint oldStake = Troves[_borrower][_collateral].stak
1204:         Troves[_borrower][_collateral].stake = newStake;

1206:         totalStakes[_collateral] = totalStakes[_collateral]
1207:         emit TotalStakesUpdated(_collateral, totalStakes[_c
```

ActivePool.sol.sendCollateral(): collAmount[_collateral] should be cached in local storage

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/ActivePool.sol#L171-L188

```
File: /Ethos-Core/contracts/ActivePool.sol
171:    function sendCollateral(address _collateral, address _ac

175:        collAmount[_collateral] = collAmount[_collateral].su
176:        emit ActivePoolCollateralBalanceUpdated(_collateral,
```

ActivePool.sol.increaseLUSDDebt(): LUSDDebt[_collateral] should be cached in local storage

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/ActivePool.sol#L190-L195

```
File: /Ethos-Core/contracts/ActivePool.sol
190:    function increaseLUSDDebt(address _collateral, uint _amo

193:        LUSDDebt[_collateral] = LUSDDebt[_collateral].add(_a
194:        ActivePoolLUSDDebtUpdated(_collateral, LUSDDebt[_col
195:    }
```

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/ActivePool.sol#L197-L202

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/ActivePool.sol#L204-L212

ReaperVaultV2.sol.availableCapital(): strategies[stratAddr] should be cached in local storage

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperVaultV2.sol#L225-L252

```
File: /Ethos-Vault/contracts/ReaperVaultV2.sol
225:    function availableCapital() public view returns (int256)
226:        address stratAddr = msg.sender;
227:        if (totalAllocBPS == 0 || emergencyShutdown) {
228:            return -int256(strategies[stratAddr].allocated);
229:        }

231:        uint256 stratMaxAllocation = (strategies[stratAddr].
232:        uint256 stratCurrentAllocation = strategies[stratAdd
```

Multiple address mappings can be combined into a single mapping of an address to a struct, where appropriate

Saves a storage slot for the mapping. Depending on the circumstances and sizes of types, can avoid a Gsset (20000 gas) per mapping combined. Reads and subsequent writes can also be cheaper when a function requires both values and they both fit in the same storage slot. Finally, if both fields are accessed in the same function, can save ~42 gas per access due to not having to recalculate the key’s keccak256 hash (Gkeccak256 - 30 gas) and that calculation’s associated stack operations.

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/TroveManager.sol#L88-L94

```
File: /Ethos-Core/contracts/TroveManager.sol
88:    mapping (address => uint) public totalStakes;

91:    mapping (address => uint) public totalStakesSnapshot;

94:    mapping (address => uint) public totalCollateralSnapshot;
```

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/TroveManager.sol#L104-L105

```
File: /Ethos-Core/contracts/TroveManager.sol
104:    mapping (address => uint) public L_Collateral;
105:    mapping (address => uint) public L_LUSDDebt;
```

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/TroveManager.sol#L119-L120

```
File: /Ethos-Core/contracts/TroveManager.sol
119:    mapping (address => uint) public lastCollateralError_Rec
120:    mapping (address => uint) public lastLUSDDebtError_Redis
```

Using storage instead of memory for structs/arrays saves gas

When fetching data from a storage location, assigning the data to a memory variable causes all fields of the struct/array to be read from storage, which incurs a Gcoldsload (2100 gas) for each field of the struct/array. If the fields are read from the new memory variable, they incur an additional MLOAD rather than a cheap stack read. Instead of declearing the variable with the memory keyword, declaring the variable with the storage keyword and caching any fields that need to be re-read in stack variables, will be much cheaper, only incuring the Gcoldsload for the fields actually read. The only time it makes sense to read the whole struct/array into a memory variable, is if the full struct/array is being returned by the function, is being passed to a function that requires memory, or if the array/struct is being read from another memory array/struct

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/StabilityPool.sol#L687

```
File: /Ethos-Core/contracts/StabilityPool.sol
687:        Snapshots memory snapshots = depositSnapshots[_depos
```

```
diff --git a/Ethos-Core/contracts/StabilityPool.sol b/Ethos-Core
index db163b7..5f2a340 100644
--- a/Ethos-Core/contracts/StabilityPool.sol
+++ b/Ethos-Core/contracts/StabilityPool.sol
@@ -684,7 +684,7 @@ contract StabilityPool is LiquityBase, Ownak
         uint initialDeposit = deposits[_depositor].initialValue
         if (initialDeposit == 0) {return 0;}

-        Snapshots memory snapshots = depositSnapshots[_deposito
+        Snapshots storage snapshots = depositSnapshots[_deposit
         uint LQTYGain = _getLQTYGainFromSnapshots(initialDeposi
```

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/StabilityPool.sol#L722

```
File: /Ethos-Core/contracts/StabilityPool.sol
```

```
722:        Snapshots memory snapshots = depositSnapshots[_depos
```

Avoid contract existence checks by using solidity version 0.8.10 or later

Prior to 0.8.10 the compiler inserted extra code, including EXTCODESIZE (100 gas), to check for contract existence for external calls. In more recent solidity versions, the compiler will not insert these checks if the external call has a return value.

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/BorrowerOperations.sol#L178-L181

```
File: /Ethos-Core/contracts/BorrowerOperations.sol

//@audit: getCollateralCCR(_collateral)
178:        vars.collCCR = collateralConfig.getCollateralCCR(_co

//@audit: getCollateralMCR(_collateral)
179:        vars.collMCR = collateralConfig.getCollateralMCR(_co

//@audit: getCollateralDecimals(_collateral)
180:        vars.collDecimals = collateralConfig.getCollateralDe

//@audit: fetchPrice(_collateral)
181:        vars.price = priceFeed.fetchPrice(_collateral);

//@audit: getCollateralCCR(_collateral);
286:        vars.collCCR = collateralConfig.getCollateralCCR(_co

//@audit: getCollateralMCR(_collateral)
287:        vars.collMCR = collateralConfig.getCollateralMCR(_co

//@audit: getCollateralDecimals(_collateral)
288:        vars.collDecimals = collateralConfig.getCollateralDe

//@audit: fetchPrice(_collateral)
289:        vars.price = priceFeed.fetchPrice(_collateral);

//@audit: getCollateralCCR(_collateral)
381:        uint256 collCCR = collateralConfig.getCollateralCCR(
//@audit: getCollateralDecimals(_collateral)
382:         uint256 collDecimals = collateralConfig.getCollatera

//@audit: fetchPrice(_collateral)
383:         uint price = priceFeed.fetchPrice(_collateral);

//@audit: getTroveColl(msg.sender, _collateral)
388:         uint coll = troveManagerCached.getTroveColl(msg.senc

//@audit: getTroveDebt(msg.sender, _collateral)
389:         uint debt = troveManagerCached.getTroveDebt(msg.senc

//@audit: getBorrowingFee(_LUSDAmount)
421:         uint LUSDFee = _troveManager.getBorrowingFee(_LUSDAm

//@audit: increaseTroveColl
468:         uint newColl = (_isCollIncrease) ? _troveManager.inc
469: _troveManager.decreaseTroveColl(_borrower, _collateral, _cc
470:         uint newDebt = (_isDebtIncrease) ? _troveManager.inc

//@audit: getTroveStatus(_borrower, _collateral)
542:         uint status = _troveManager.getTroveStatus(_borrowe

//@audit: getTroveStatus(_borrower, _collateral);
547:         uint status = _troveManager.getTroveStatus(_borrowe
```

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/TroveManager.sol#L420

```
File: /Ethos-Core/contracts/TroveManager.sol

//@audit: getCollateralDecimals(_collateral)
420:             uint collDecimals = collateralConfig.getCollater

//@audit: getCollateralCCR(_collateral)
521:         vars.collCCR = collateralConfig.getCollateralCCR(_cc

//@audit: getCollateralDecimals(_collateral)
522:         vars.collDecimals = collateralConfig.getCollateralDe

//@audit: getCollateralMCR(_collateral)
523:         vars.collMCR = collateralConfig.getCollateralMCR(_cc
//@audit: fetchPrice(_collateral)
524:        vars.price = priceFeed.fetchPrice(_collateral);

//@audit: getTotalLUSDDeposits();
525:        vars.LUSDInStabPool = stabilityPoolCached.getTotalLU

//@audit: getCollateralDecimals(_collateral)
596:        vars.collDecimals = collateralConfig.getCollateralDe

//@audit: getCollateralCCR(_collateral)
597:        vars.collCCR = collateralConfig.getCollateralCCR(_co

//@audit: getCollateralMCR(_collateral)
598:        vars.collMCR = collateralConfig.getCollateralMCR(_co

//@audit: getLast(_collateral)
606:        vars.user = _sortedTroves.getLast(_collateral);

//@audit: getFirst(_collateral)
607:        address firstUser = _sortedTroves.getFirst(_collater

//@audit: getPrev(_collateral, vars.user)
610:            address nextUser = _sortedTroves.getPrev(_collat

//@audit: getLast(_collateral)
691:            vars.user = sortedTrovesCached.getLast(_collater

//@audit: getCollateralDecimals(_collateral)
725:        vars.collDecimals = collateralConfig.getCollateralDe

//@audit: getCollateralCCR(_collateral)
726:        vars.collCCR = collateralConfig.getCollateralCCR(_co

//@audit: getCollateralMCR(_collateral)
727:        vars.collMCR = collateralConfig.getCollateralMCR(_co

//@audit: fetchPrice(_collateral)
728:        vars.price = priceFeed.fetchPrice(_collateral);

//@audit: getTotalLUSDDeposits()
729:        vars.LUSDInStabPool = stabilityPoolCached.getTotalLU

//@audit: getCollateralDecimals(_collateral)
798:        vars.collDecimals = collateralConfig.getCollateralDe
```

```
//@audit: getCollateralCCR(_collateral)
799:        vars.collCCR = collateralConfig.getCollateralCCR(_cc

//@audit: getCollateralMCR(_collateral)
800:        vars.collMCR = collateralConfig.getCollateralMCR(_cc

//@audit: getCollateralDecimals(_collateral)
1048:        uint256 collDecimals = collateralConfig.getCollater

//@audit: getCollateralDecimals(_collateral)
1061:        uint256 collDecimals = collateralConfig.getCollater

//@audit: getCollateralDecimals(_collateral)
1131:        uint256 collDecimals = collateralConfig.getCollater

//@audit: getCollateralDecimals(_collateral)
1146:        uint256 collDecimals = collateralConfig.getCollater

//@audit: getCollateral(_collateral)
1308:        uint activeColl = _activePool.getCollateral(_collat

//@audit: getCollateral(_collateral);
1309:        uint liquidatedColl = defaultPool.getCollateral(_cc

//@audit: getCollateralDecimals(_collateral)
1362:        uint256 collDecimals = collateralConfig.getCollater

//@audit: getCollateralCCR(_collateral);
1367:        uint256 collCCR = collateralConfig.getCollateralCCR

//@audit: getCollateralDecimals(_collateral);
1368:        uint256 collDecimals = collateralConfig.getCollater
```

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/ActivePool.sol#L105

```
File: /Ethos-Core/contracts/ActivePool.sol

//@audit: getAllowedCollaterals()
105:        address[] memory collaterals = ICollateralConfig(col

//@audit: asset()
```

```
    111:                require(IERC4626(vault).asset() == collateral, '

    //@audit: isCollateralAllowed(_collateral)
    315:            ICollateralConfig(collateralConfigAddress).isColla

    //@audit: redemptionHelper()
    328:          address redemptionHelper = address(ITroveManager(trc
```

x += y costs more gas than x = x + y for state variables

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperVaultV2.sol#L168

```
File: /Ethos-Vault/contracts/ReaperVaultV2.sol
168:          totalAllocBPS += _allocBPS;

194:          totalAllocBPS -= strategies[_strategy].allocBPS;

196:          totalAllocBPS += _allocBPS;

214:          totalAllocBPS -= strategies[_strategy].allocBPS;

396:          totalAllocated -= actualWithdrawn;

445:          totalAllocBPS -= bpsChange;

452:          totalAllocated -= loss;

515:          totalAllocated -= vars.debtPayment;

521:          totalAllocated += vars.credit;
```

Using unchecked blocks to save gas

Solidity version 0.8+ comes with implicit overflow and underflow checks on unsigned integers. When an overflow or an underflow isn't possible (as an example, when a comparison is made before the arithmetic operation), some gas can be saved by using an unchecked block.
[see resource]()

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperVaultV2.sol#L235

```
File: /Ethos-Vault/contracts/ReaperVaultV2.sol
235:                return -int256(stratCurrentAllocation - stratMax
```

The operation `stratCurrentAllocation - stratMaxAllocation` cannot underflow due to the check on Line 234 that ensures that the operation would only be performed if `stratCurrentAllocation` is greater than `stratMaxAllocation`.

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperVaultV2.sol#L244

```
File: /Ethos-Vault/contracts/ReaperVaultV2.sol
244:                uint256 available = stratMaxAllocation - stratCu
```

The operation `stratMaxAllocation - stratCurrentAllocation` cannot underflow due to the check on Line 236 that ensures that `stratMaxAllocation` is greater than `stratCurrentAllocation` before performing the arithmetic operation.

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperVaultV2.sol#L245

```
File: /Ethos-Vault/contracts/ReaperVaultV2.sol
245:                available = Math.min(available, vaultMaxAllocati
```

The operation `vaultMaxAllocation - vaultCurrentAllocation` cannot underflow due to the check on Line 240 that ensures that `vaultMaxAllocation` is greater than `vaultCurrentAllocation` before performing our arithmetic operation.

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperVaultV2.sol#L384

```
File: /Ethos-Vault/contracts/ReaperVaultV2.sol
384:                    uint256 remaining = value - vaultBalance;
```

The operation `value - vaultBalance` cannot underflow due to the check on Line 374 that ensures that if `value` is less than `vaultBalance`, the loop would break out and our operation would never be performed.

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperStrategyGranarySupplyOnly.sol#L81

```
File: /Ethos-Vault/contracts/ReaperStrategyGranarySupplyOnly.sol
81:               uint256 toReinvest = wantBalance - _debt;
```

The operation `wantBalance - _debt` cannot underflow due to the check on Line 80 that ensures that `wantBalance` is greater than `_debt` before performing the arithmetic operation.

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperStrategyGranarySupplyOnly.sol#L93

```
File: /Ethos-Vault/contracts/ReaperStrategyGranarySupplyOnly.sol
93:               _withdraw(_amountNeeded - wantBal);
```

The operation `_amountNeeded - wantBal` cannot underflow due to the check on Line 92 that ensures that `_amountNeeded` is greater than `wantBal` before performing the arithmetic operation.

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-

Vault/contracts/ReaperStrategyGranarySupplyOnly.sol#L100

```
File: /Ethos-Vault/contracts/ReaperStrategyGranarySupplyOnly.sol
100:                loss = _amountNeeded - liquidatedAmount;
```

The operation `_amountNeeded - liquidatedAmount` cannot underflow due to the check on Line 99 that ensures that `_amountNeeded` is greater than `liquidatedAmount` before performing the arithmetic operation.

Splitting require() statements that use && saves gas - (saves 8 gas per &&)

Instead of using the && operator in a single require statement to check multiple conditions,using multiple require statements with 1 condition per require statement will save 8 GAS per `&&`.

The gas difference would only be realized if the revert condition is realized(met).

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/BorrowerOperations.sol#L653-L654

```
File: /Ethos-Core/contracts/BorrowerOperations.sol
653:                require(_maxFeePercentage >= BORROWING_FEE_FLOOR
654:                    "Max fee percentage must be between 0.5% and
```

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/TroveManager.sol#L1539

```
File: /Ethos-Core/contracts/TroveManager.sol
1539:        require (TroveOwnersArrayLength > 1 && sortedTroves
```

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-

Core/contracts/LUSDToken.sol#L347-L351

```
File: /Ethos-Core/contracts/LUSDToken.sol
347:        require(
348:            _recipient != address(0) &&
349:            _recipient != address(this),
350:            "LUSD: Cannot transfer tokens directly to the LU
351:        );


352:        require(
353:            !stabilityPools[_recipient] &&
354:            !troveManagers[_recipient] &&
355:            !borrowerOperations[_recipient],
356:            "LUSD: Cannot transfer tokens directly to the St
357:        );
```

Use the cached value in the following to save gas.

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/ActivePool.sol#L171-L188

```
File: /Ethos-Core/contracts/ActivePool.sol
171:    function sendCollateral(address _collateral, address _ac

179:        if (_account == defaultPoolAddress) {
180:            IERC20(_collateral).safeIncreaseAllowance(defaul
181:    IDefaultPool(defaultPoolAddress).pullCollateralFromActiv
182:        } else if (_account == collSurplusPoolAddress) {
183:            IERC20(_collateral).safeIncreaseAllowance(collSu
185:        } else {
186:            IERC20(_collateral).safeTransfer(_account, _amou
187:        }
188:    }
```

In the above function, The first if condition checks that `_account == defaultPoolAddress` meaning the operations would only be performed if the two are the same, as `defaultPoolAddress` is a storage variable(1 SLOAD = 100 gas) we could replace it's occurence inside the if blocks with the local `_account` variable (1

MLOAD = 3 gas) since the two are the same. The storage variable is only read in this block thus we can get away with using the local one.

Similar explanation to the above applies to the `collSurplusPoolAddress` variable

```
diff --git a/Ethos-Core/contracts/ActivePool.sol b/Ethos-Core/cc
index 753fcd0..735a0dd 100644
--- a/Ethos-Core/contracts/ActivePool.sol
+++ b/Ethos-Core/contracts/ActivePool.sol
@@ -177,11 +177,11 @@ contract ActivePool is Ownable, CheckContr
         emit CollateralSent(_collateral, _account, _amount);

         if (_account == defaultPoolAddress) {
-            IERC20(_collateral).safeIncreaseAllowance(defaultPc
-            IDefaultPool(defaultPoolAddress).pullCollateralFrom
+            IERC20(_collateral).safeIncreaseAllowance(_account,
+            IDefaultPool(_account).pullCollateralFromActivePool
         } else if (_account == collSurplusPoolAddress) {
-            IERC20(_collateral).safeIncreaseAllowance(collSurpl
-            ICollSurplusPool(collSurplusPoolAddress).pullCollat
+            IERC20(_collateral).safeIncreaseAllowance(_account,
+            ICollSurplusPool(_account).pullCollateralFromActive
         } else {
             IERC20(_collateral).safeTransfer(_account, _amount)
         }
```

Caching global variables is expensive than using the variable itself

Don’t cache msg.sender in the following

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperVaultV2.sol#L225-L252

```
File: /Ethos-Vault/contracts/ReaperVaultV2.sol
225:    function availableCapital() public view returns (int256)
226:        address stratAddr = msg.sender;
227:        if (totalAllocBPS == 0 || emergencyShutdown) {
228:            return -int256(strategies[stratAddr].allocated);
```

```
        }

        uint256 stratMaxAllocation = (strategies[stratAddr].
        uint256 stratCurrentAllocation = strategies[stratAdc
```

```
diff --git a/Ethos-Vault/contracts/ReaperVaultV2.sol b/Ethos-Vau
index b5f2a58..d6f5d3e 100644
--- a/Ethos-Vault/contracts/ReaperVaultV2.sol
+++ b/Ethos-Vault/contracts/ReaperVaultV2.sol
@@ -223,13 +223,12 @@ contract ReaperVaultV2 is ReaperAccessCont
       * the vault.
       */
      function availableCapital() public view returns (int256) {
-         address stratAddr = msg.sender;
          if (totalAllocBPS == 0 || emergencyShutdown) {
-             return -int256(strategies[stratAddr].allocated);
+             return -int256(strategies[msg.sender].allocated);
          }

-         uint256 stratMaxAllocation = (strategies[stratAddr].all
-         uint256 stratCurrentAllocation = strategies[stratAddr].
+         uint256 stratMaxAllocation = (strategies[msg.sender].al
+         uint256 stratCurrentAllocation = strategies[msg.sender]
```

Don’t cache msg.sender in the following

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Vault/contracts/ReaperVaultV2.sol#L493-L560

```
File: /Ethos-Vault/contracts/ReaperVaultV2.sol
    function report(int256 _roi, uint256 _repayment) external re
        LocalVariables_report memory vars;
        vars.stratAddr = msg.sender;
        StrategyParams storage strategy = strategies[vars.stratA
        require(strategy.activation != 0, "Unauthorized strategy


        if (_roi < 0) {
            vars.loss = uint256(-_roi);
            _reportLoss(vars.stratAddr, vars.loss);
        } else if (_roi > 0) {
            vars.gain = uint256(_roi);
            vars.fees = _chargeFees(vars.stratAddr, vars.gain);
            strategy.gains += vars.gain;
        }


        if (vars.credit > vars.freeWantInStrat) {
            token.safeTransfer(vars.stratAddr, vars.credit - vai
        } else if (vars.credit < vars.freeWantInStrat) {
            token.safeTransferFrom(vars.stratAddr, address(this)
        }

        emit StrategyReported(
            vars.stratAddr,

        );


        if (strategy.allocBPS == 0 || emergencyShutdown) {
            return IStrategy(vars.stratAddr).balanceOf();
        }
```

Use a more recent version of solidity

Use a solidity version of at least 0.8.2 to get simple compiler automatic inlining Use a solidity version of at least 0.8.3 to get better struct packing and cheaper multiple storage reads Use a solidity version of at least 0.8.4 to get custom errors, which are cheaper at deployment than revert()/require() strings Use a solidity version of at least 0.8.10 to have external calls skip contract existence checks if the external call has a return value.

Most of the in scope code seems to be running versions lower than 0.8

https://github.com/code-423n4/2023-02-ethos/blob/73687f32b934c9d697b97745356cdf8a1f264955/Ethos-Core/contracts/TroveManager.sol#L3

```
File: /Ethos-Core/contracts/TroveManager.sol
3:pragma solidity 0.6.11;
```

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