Venus Protocol Isolated Pools Findings & Analysis Report

2023-08-09

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 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 outlined in this document, C4 conducted an analysis of the Venus Protocol Isolated Pools smart contract system written in Solidity. The audit took place between May 8 - May 15 2023.

Wardens

113 Wardens contributed reports to the Venus Protocol Isolated Pools:

1. 0x73696d616f
2. 0x8chars
3. 0xAce (Coollaitar and 0xaditya)
4. 0xSmartContract
5. 0xStalin
6. 0xWaitress
7. 0xadrii
8. 0xbepresent
9. 0xcm

10. 0xkazim
11. 0xnev
12. Audit_Avengers_2 (JP_Courses, ravikiranweb3, zzebra83 and 0xMosh)
13. Aymen0909
14. BGSecurity (anonresercher and martin)
15. BPZ (Bitcoinfever244, PrasadLak and zinc42)
16. Bauchibred
17. BoltzmannBrain
18. Brenzee
19. BugBusters (nirlin and 0xepley)
20. Cayo
21. ChrisTina
22. CoOnan
23. Cryptor
24. DeliChainSec (deliriusz and 0xffchain)
25. Emmanuel
26. Franfran
27. IceBear
28. Infect3d
29. J4de
30. JCN
31. Josiah
32. K42
33. Kose
34. Lilyjjo
35. LokiThe5th
36. MohammedRizwan
37. Norah
38. PNS

39. Parad0x
40. QiuhaoLi
41. Rageur
42. Raihan
43. RaymondFam
44. ReyAdmirado
45. SAAJ
46. SM3_SS
47. SaeedAlipoor01988
48. Sathish9098
49. Team_Rocket (Shame and AlexCzm)
50. Udsen
51. YakuzaKiawe
52. Yardi256
53. YoungWolves (lopotras and Bloqarl)
54. YungChaza
55. ast3ros
56. berlin-101
57. bin2chen
58. brgltd
59. btk
60. c3phas
61. carlitox477
62. chaieth
63. codeslide
64. dacian
65. descharre
66. fatherOfBlocks
67. frazerch

68. fs0c
69. hunter_w3b
70. j4ld1na
71. jasonxiale
72. joestakey
73. kodyvim
74. koxuan
75. lanrebayode77
76. lfzkoala
77. lllu_23
78. lukris02
79. matrix_0wl
80. mussucal
81. nadin
82. naman1778
83. pavankv
84. peanuts
85. peritoflores
86. petrichor
87. pontifex
88. qpzm
89. rapha
90. rvierdiiev
91. sashik_eth
92. sces60107
93. souilos
94. thekmj
95. tnevler
96. volodya

97. wahedtalash77
98. wonjun
99. xuwinnie
100. yjrwkk
101. yongskiws
102. zzykxx

This audit was judged by Oxean.

Final report assembled by thebrittfactor.

Summary

The C4 analysis yielded an aggregated total of 17 unique vulnerabilities. Of these vulnerabilities, 1 received a risk rating in the category of HIGH severity and 16 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 42 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 27 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 Venus Protocol Isolated Pools repository, and is composed of 28 smart contracts written in the Solidity programming language and includes 3549 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 (1)

[H-01] Incorrect `blocksPerYear` constant in `WhitepaperInterestRateModel`

Submitted by Team_Rocket, *also found by* thekmj, MohammedRizwan, peritoflores, DeliChainSec, 0xkazim, sces60107, 0xkazim, ast3ros, BPZ, carlitox477, sashik_eth, Yardi256, berlin-101, zzykxx, Brenzee, fs0c, Franfran, Bauchibred, BoltzmannBrain, SaeedAlipoor01988, Lilyjjo *and* volodya.

https://github.com/code-423n4/2023-05-venus/blob/8be784ed9752b80e6f1b8b781e2e6251748d0d7e/contracts/WhitePaperInterestRateModel.sol#L17

The interest rate per block is **5x** greater than it's intended to be for markets that use the Whitepaper interest rate model.

Proof of Concept

The `WhitePaperInterestRateModel` contract is forked from Compound Finance, which was designed to be deployed on Ethereum Mainnet. The `blocksPerYear` constant inside the contract is used to calculate the interest rate of the market on a per-block basis and is set to **2102400**, which assumes that there are 365 days a year and that the block-time is **15 seconds**.

However, Venus Protocol is deployed on the BNB chain, which has a block-time of only **3 seconds**. This results in the interest rate per block on the BNB chain to be **5x** greater than intended.

Both `baseRatePerBlock` and `multiplierPerBlock` are affected and are **5x** the value they should be. This also implies that the pool's interest rate is also 5 times more sensitive to utilization rate changes than intended. It is impossible for the market to arbitrage and adjust the interest rate back to the intended rate as seen in the PoC graph below. It's likely that arbitrageurs will deposit as much collateral as possible to take advantage of the high supply rate, leading to a utilization ratio close to 0.

The following Python script plots the `WhitePaperInterestRateModel` curves for a 15 second and a 3 second block time.

```
import matplotlib.pyplot as plt

# Constants
BASE = 1e18

# Solidity functions converted to Python functions
def utilization_rate(cash, borrows, reserves):
    if borrows == 0:
        return 0
    return (borrows * BASE) / (cash + borrows - reserves)

def get_borrow_rate(ur, base_rate_per_block, multiplier_per_bloc
    return ((ur * multiplier_per_block) / BASE) + base_rate_per_

def generate_data_points(base_rate_per_year, multiplier_per_yea
    base_rate_per_block = base_rate_per_year / blocks_per_year
    multiplier_per_block = multiplier_per_year / blocks_per_yea

    utilization_rates = [i / 100 for i in range(101)]
    borrow_rates = [get_borrow_rate(ur * BASE, base_rate_per_blo

    return utilization_rates, borrow_rates

# User inputs
base_rate_per_year = 5e16 # 5%
multiplier_per_year = 1e16 # 1%
blocks_per_year1 = 2102400 # 15 second block-time
blocks_per_year2 = 10512000 # 3 second block-time

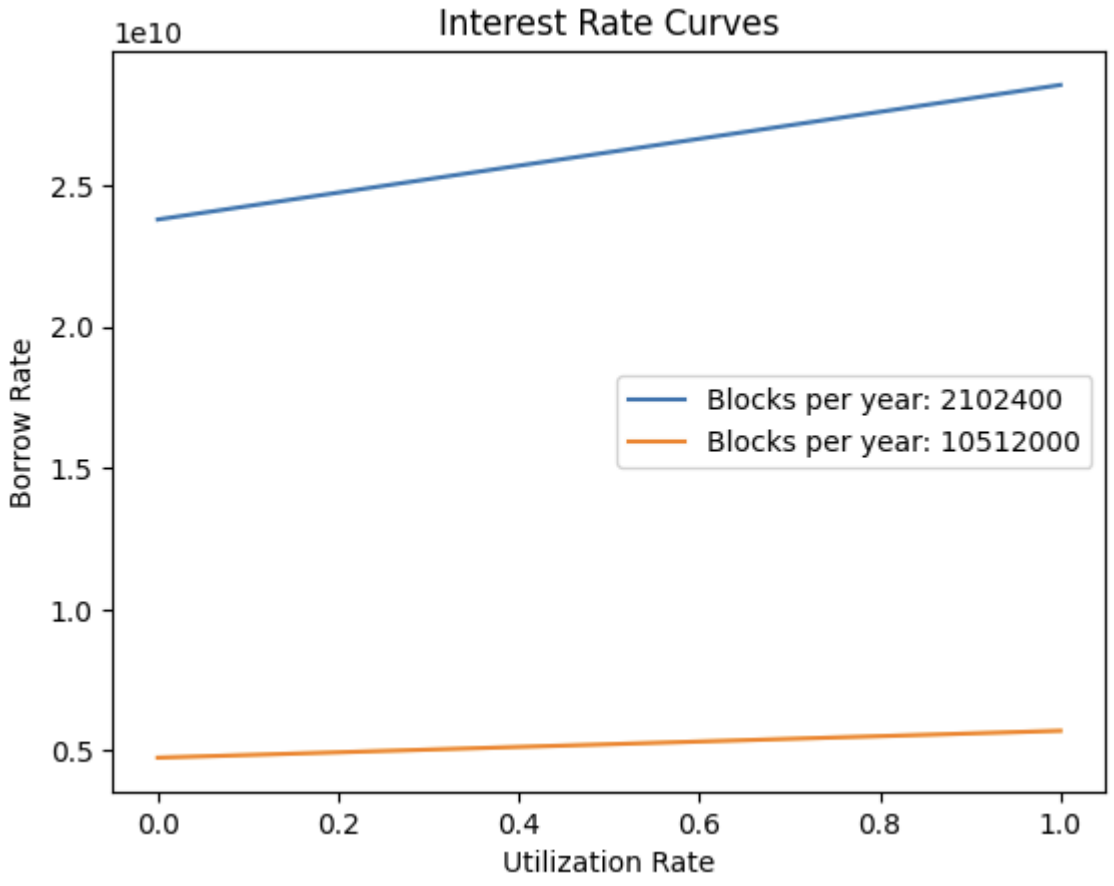
# Example values for cash, borrows, and reserves
cash = 1e18
borrows = 5e18
```

```
reserves = 0.1e18

# Generate data points for both curves
utilization_rates1, borrow_rates1 = generate_data_points(base_ra
utilization_rates2, borrow_rates2 = generate_data_points(base_ra

# Plot both curves on the same plot with a key
plt.plot(utilization_rates1, borrow_rates1, label=f"Blocks per y
plt.plot(utilization_rates2, borrow_rates2, label=f"Blocks per y
plt.xlabel("Utilization Rate")
plt.ylabel("Borrow Rate")
plt.title("Interest Rate Curves")
plt.legend()
plt.show()
```

Result:



As seen above, the borrow rate curves are different and do not intersect. Hence, it's impossible via arbitrage for market participants to adjust the rate back to its intended value.

Recommended Mitigation Steps

Fix the `blocksPerYear` constant so that it accurately describes the number of blocks a year on the BNB chain, which has a block-time of 15 seconds. The correct value is **10512000**.

```
\begin{aligned}
\text{blocksPerYear} &= \frac{\text{secondsInAYear}}{\text{block
&= \frac{365 \times 24 \times 60 \times 60}{3} \\
&= 10{,}512{,}000
\end{aligned}
```

```
@@ -14,7 +14,7 @@ contract WhitePaperInterestRateModel is Intere
     /**
      * @notice The approximate number of blocks per year that i
      */
-    uint256 public constant blocksPerYear = 2102400;
+    uint256 public constant blocksPerYear = 10512000;

     /**
      * @notice The multiplier of utilization rate that gives th
```

Oxean (judge) increased severity to High

chechu (Venus) confirmed via duplicate issue #559

Medium Risk Findings (16)

[M-01] Malicious actor can win an auction unfavorably to the protocol by block stuffing

Submitted by DeliChainSec

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Shortfall/Shortfall.sol#L158-L202
https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Shortfall/Shortfall.sol#L467-L470

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Shortfall/Shortfall.sol#L213

When the protocol bad debt is auctioned off with 10% incentive at the beginning, a user who gives the best bid wins. The auction ends when at least one account placed a bid, and the current block number is bigger than `nextBidderBlockLimit`:

```
function closeAuction(address comptroller) external nonReentrant
        Auction storage auction = auctions[comptroller];

        require(_isStarted(auction), "no on-going auction");
        require(
            block.number > auction.highestBidBlock + nextBidderB
            "waiting for next bidder. cannot close auction"
        );
```

`nextBidderBlockLimit` is set to 10 in the initializer, which means that other users have only 30 seconds to place a better bid. This is a serious problem because stuffing a whole block with dummy transactions is very cheap on Binance Smart Chain. According to https://www.cryptoneur.xyz/en/gas-fees-calculator, 15M gas - whole block - costs `$` 14~ `$` 15 on BSC. This makes a malicious user occasionally cheaply prohibit other users to overbid them, winning the auction at the least favorable price for the protocol. Because BSC is a centralized blockchain, there are no private mempools and bribes directly to the miners (like in FlashBots); hence, other users are very limited concerning the prohibitive actions.

Impact

The protocol overpays for bad debt, losing value.

Proof of Concept

1. Pool gathered `$` 100,000 in bad debt and it's eligible for auction.
2. A malicious user frontruns others and places the first bid with the least possible amount (bad debt + 10% incentive).
3. The user sends dozens of dummy transactions with increased gas prices, only to fill up whole block space for 11 blocks.

4. At the end, the user sends a transaction to close the auction, getting the bad debt + 10% incentive.

Recommended Mitigation Steps

There are at least three options to resolve this issue:

1. Make the bidding window much higher at the beginning; like 1000 blocks.
2. Make the bidding window very high at the beginning and decrease it; the more attractive the new bid is.
3. Make the bidding window dependent on the money at stake, to disincentivize block stuffing.

chechu (Venus) confirmed

[M-02] It's possible to borrow, redeem, transfer tokens and exit markets with outdated collateral prices and borrow interest

Submitted by 0x73696d616f, *also found by* pontifex, 0xbepresent, J4de, J4de, 0xkazim, peanuts, 0xadrii, rvierdiiev, rvierdiiev, rvierdiiev, volodya, rvierdiiev *and* rvierdiiev.

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Comptroller.sol#L199
https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Comptroller.sol#L299
https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Comptroller.sol#L324
https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Comptroller.sol#L553
https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Comptroller.sol#L1240
https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Comptroller.sol#L1255
https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Comptroller.sol#L578

Incorrect `borrowBalance` and token collateral values. This could lead to many different exploits, such as:

- Users with a collateral token that fell substantially in price can borrow another underlying token, whose price has not been updated, and earn profit.
- Users can borrow/redeem/transfer more if the interest/price was not updated.

Proof of Concept

In the `Comptroller`, the total collateral balance and borrow balance are calculated at `_getHypotheticalLiquiditySnapshot(...)`. This function calculates these balances in the following loop:

```
    for (uint256 i; i < assetsCount; ++i) {
        VToken asset = assets[i];

        // Read the balances and exchange rate from the vToken
        (uint256 vTokenBalance, uint256 borrowBalance, uint256 excha
            asset,
            account
        );

        // Get the normalized price of the asset
        Exp memory oraclePrice = Exp({ mantissa: _safeGetUnderlyingP

        // Pre-compute conversion factors from vTokens -> usd
        Exp memory vTokenPrice = mul_(Exp({ mantissa: exchangeRateMa
        Exp memory weightedVTokenPrice = mul_(weight(asset), vTokenP

        // weightedCollateral += weightedVTokenPrice * vTokenBalance
        snapshot.weightedCollateral = mul_ScalarTruncateAddUInt(
            weightedVTokenPrice,
            vTokenBalance,
            snapshot.weightedCollateral
        );

        // totalCollateral += vTokenPrice * vTokenBalance
        snapshot.totalCollateral = mul_ScalarTruncateAddUInt(vTokenP

        // borrows += oraclePrice * borrowBalance
        snapshot.borrows = mul_ScalarTruncateAddUInt(oraclePrice, bo

        // Calculate effects of interacting with vTokenModify
```

```
        if (asset == vTokenModify) {
            // redeem effect
            // effects += tokensToDenom * redeemTokens
            snapshot.effects = mul_ScalarTruncateAddUInt(weightedVTo

            // borrow effect
            // effects += oraclePrice * borrowAmount
            snapshot.effects = mul_ScalarTruncateAddUInt(oraclePrice
        }
    }
```

As seen, the oracle price is not updated via calling `updatePrice(...)`, nor is the borrow interest updated by calling `AccrueInterest(...)`. Only the corresponding `VToken` that called the `borrow(...)`, `transfer(...)` or `redeem(...)` has an updated price and interest, which could lead to critical inaccuracies for accounts with several VTokens.

Tools Used

Vscode, Hardhat

Recommended Mitigation Steps

Update the price and interest of every collateral, except the VToken that triggered the hook which has already been updated. Similarly to what is being done on `healAccount(...)`:

```
    for (uint256 i; i < userAssetsCount; ++i) {
       userAssets[i].accrueInterest();
       oracle.updatePrice(address(userAssets[i]));
    }
```

Assessed type

Oracle

chechu (Venus) acknowledged via duplicate issue #88 and commented:

> Oracle price of every market is updated on every action involving directly that market. We didn't update the prices of every market (secondarily) used in some operations to save gas, assuming the price would be valid (there are mechanisms in the Oracles to avoid the use of old prices). We'll monitor this topic to decide if our approach is enough or if we have to update the price of every market every time we invoke `_getHypotheticalLiquiditySnapshot`.

Oxean (judge) decreased severity to Medium

chechu (Venus) commented:

> To share more info related to this topic:
>
> - As we said in **#88**, we assume the prices in the rest of the markets will be updated frequently because they are updated every time other users interact directly with these other markets
> - > Moreover, "update" in our context only affects the TWAP oracle. As you can see **here**, our oracles system uses Chainlink, Binance Oracle, Pyth, and TWAP sources.
>
>   - Chainlink and Binance prices are not updated by Venus anyway. Chainlink and Binance update these price feeds following the classical rules of heartbeat and deviation (example **here**). In the Oracle system, we check the last time these prices were updated and discard them (reverting the TX) if they are staled.
>   - TWAP needs a proactive update, executed by Venus when the mentioned `oracle.updatePrice` is invoked. If no one invokes this update function for an asset later used indirectly by another user, and the difference between the price offered by the TWAP oracle and the rest of the oracles (i.e. Chainlink) is greater than a threshold configure in our Oracle system, the TX will be also reverted. Could this generate a DoS? I suppose that potentially it can do it, but as soon as a user interacts with the "staled" asset in Venus the block will disappear.

Oxean (judge) commented:

> @chechu - I believe I should mark #104 as a dupe of this as well. Since this issue talks about both prices updated and accruing interests.

> we assume the prices in the rest of the markets will be updated frequently because they are updated every time other users interact directly with these other markets

> This assumption I think has risks in which the wardens are calling out. It's hard to say how real these risks are apriori without making assumptions about the usage of all the markets.

> I think these issues should be batched together over concerns around `accrueInterest` and price updates into a single M issue.

chechu (Venus) commented:

> This assumption I think has risks in which the wardens are calling out. It's hard to say how real these risks are apriori without making assumptions about the usage of all the markets.

> @0xean - In my honest opinion, I think the risk is low, but I could understand the concern and the lack of a guarantee in the code.

[M-03] liquidateAccount will fail if the transaction is not included in the current block

Submitted by **xuwinnie**, *also found by* **Udsen**, **mussucal** *and* **BoltzmannBrain**.

Functon `liquidateAccount` will fail if the transaction is not included in the current block because interest accrues per block, and `repayAmount` and `borrowBalance` need to match precisely.

Proof of Concept

At the end of the function `liquidateAccount` , a check is performed to ensure that the `borrowBalance` is zero:

```
    for (uint256 i; i < marketsCount; ++i) {
            (, uint256 borrowBalance, ) = _safeGetAccountSnapshot(bo
            require(borrowBalance == 0, "Nonzero borrow balance afte
    }
```

This means that `repayAmount` specified in calldata must exactly match the `borrowBalance` call. (If `repayAmount` is greater than `borrowBalance`, `Comptroller.preLiquidateHook` will revert with the error `TooMuchRepay`.) However, the `borrowBalance` updates every block due to interest accrual. The liquidator cannot be certain that their transaction will be included in the current block or in a future block. This uncertainty significantly increases the likelihood of liquidation failure.

Recommended Mitigation Steps

Use a looser check:

```
snapshot = _getCurrentLiquiditySnapshot(borrower, _getLiquidatic
require (snapshot.shortfall == 0);
```

to replace:

```
for (uint256 i; i < marketsCount; ++i) {
        (, uint256 borrowBalance, ) = _safeGetAccountSnapshot(bc
        require(borrowBalance == 0, "Nonzero borrow balance afte
}
```

Assessed type

Invalid Validation

chechu (Venus) confirmed via duplicate issue #545

chechu (Venus) disagreed with severity and commented:

> Suggestion: QA (no risk for funds, no risk of DoS).

> The liquidator has to take into account the pending interests to be accrued before invoking `liquidateAccount`. It's technically feasible and if the TX fails, they can retry it, so finally the position will be liquidated. The amount to be liquidated will be very low, so we don't see any risk of front running.

Oxean (judge) decreased severity to Medium and commented:

> They would have to know which block specifically their transaction would get mined in to be able to precompute this.
>
> While they could retry the transaction, I do think this will have an impact of the protocol's availability under normal conditions and therefore does meet the criteria for Medium severity.

chechu (Venus) commented:

> They would have to know which block specifically their transaction would get mined in to be able to precompute this.
>
> While they could retry the transaction, I do think this will have an impact of the protocol's availability under normal conditions and therefore does meet the criteria for Medium severity.

> @0xean - The liquidator can calculate the exact needed amount in a contract; for example, to guarantee that the amount is valid in the same block where the transaction will be minted.

Oxean (judge) commented:

> @chechu - I agree, that is possible, but how does one do this from an EOA?

chechu (Venus) commented:

> @chechu - I agree, that is possible, but how does one do this from an EOA?

> @0xean - with an EOA I would do a multicall, first statically invoking the functions to accrue interest, and then invoking the function to get the precise borrow amounts. You could use https://www.npmjs.com/package/ethereum-multicall to do this. We do something similar in our frontend here.
>
> By doing this, it's true that your TX may not be included in the current block and it won't be valid in the next one.

Oxean (judge) commented:

> Yeah, this seems like a workaround to the problem, in my opinion. Why would you be opposed to simply updating the function to make it more tolerant to the specific block it's called in?

[M-04] `_ensureMaxLoops` causes liquidateAccount to fail in certain conditions

Submitted by xuwinnie*, also found by* 0x8chars.

The function `_ensureMaxLoops` reverts if the iteration count exceeds the `maxLoopsLimit`. However, the limitation imposed by `maxLoopsLimit` hinders the functioning of `liquidateAccount` under certain conditions, as `orderCount` needs to reach twice the market count (which is also constrained by the `maxLoopsLimit`) in extreme cases.

Proof of Concept

Suppose `maxLoopsLimit` is set to **16** and currently **12** markets have been added, which is allowed by `_ensureMaxLoops` in function `_addMarket`:

```
allMarkets.push(VToken(vToken));
marketsCount = allMarkets.length;
_ensureMaxLoops(marketsCount);
```

Then, Alice enters all **12** markets by depositing and borrowing simultaneously, which is also allowed by `_ensureMaxLoops` in function `enterMarkets`:

```
uint256 len = vTokens.length;
uint256 accountAssetsLen = accountAssets[msg.sender].length;
_ensureMaxLoops(accountAssetsLen + len);
```

To illustrate, assume these **12** coins are all stablecoin with an equal value. Let's call them USDA - USDL. Alice deposits 20 USDA, 1.1 USDB - 1.1 USDL, worth 32.1 USD in total. Then she borrows 2 USDA - 2 USDL, worth 24 USD in total. Unluckily, USDA de-pegs to 0.6 USD and Alice's deposit value drops to 24.1 USD, which is below the

liquidation threshold (also below the `minLiquidatableCollateral`). However, nobody can liquidate Alice's account by calling `liquidateAccount`, because the least possible `orderCount` is **23**, which exceeds `maxLoopsLimit`.

Let's take a closer look at `LiquidationOrder`:

```
struct LiquidationOrder {
        VToken vTokenCollateral;
        VToken vTokenBorrowed;
        uint256 repayAmount;
}
```

In this case, the liquidator cannot perfectly match `vTokenCollateral` with `vTokenBorrowed` one-to-one. Because the value of collateral and debt is not equal, more than one order is needed to liquidate each asset. To generalize, if the asset count is **n**, in the worst case, **2n-1** orders are needed for a complete liquidation (not hard to prove).

Recommended Mitigation Steps

```
_ensureMaxLoops(ordersCount / 2);
```

Assessed type

Loop

[0xean (judge) commented](#):

> more than one order is needed to liquidate each asset

> I am not sure I am tracking this assertion in the above report.

[chechu (Venus) disputed and commented](#):

> We can resolve it by just increasing the limit accepted by `_ensureMaxLoops`, with a VIP. By the way, the suggestion is not valid, because the number of iterations will

> be `ordersCount`, not `ordersCount/2`.

Oxean (judge) commented:

> We can resolve it by just increasing the limit accepted by `_ensureMaxLoops`, with a VIP.

> I don't think this is a valid mitigation and would still cause a disruption to the protocol that warrants the Medium severity.

> Per the C4 docs

> 2 Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.

Oxean (judge) commented:

> @chechu - the validity of this issue comes down to the assumption of there being more than 1 order per asset, let's discuss that point specifically, because otherwise, this issue doesn't seem to be valid.

chechu (Venus) confirmed and commented:

> Sorry for the delay in my response. Reviewing it more carefully, the finding seems valid. It's true that in `Comptroller.liquidateAccount` several orders per borrowed asset could be needed. Example:

> Params:

- liquidationIncentiveMantissa: 1.1
- minLiquidatableCollateral: `$` 100

> Borrower position:

- > Collateral:
  - USDC: `$` 20 (liquidation threshold: 0.8)
  - USDT: `$` 20 (liquidation threshold: 0.8)

- Borrow CAKE: $35

> Every condition is satisfied to allow the execution of `liquidationAccount`:

- Total collateral ($40) < minLiquidatableCollateral ($100)
- `collateralToSeize` ($35 * 1.1 = $38.5) < total collateral ($40)
- Shortfall ($35 - ($20 * 0.8 + $20 * 0.8) = $3) > 0

> But the liquidator cannot repay $35 of the borrowed asset and get enough tokens of just one of the collaterals. So, the liquidator will need two orders:

- > order 1:
  - collateral USDC
  - borrowed asset: CAKE
  - repay amount: $17.5 (getting $19.25 of USDC)
- > order 2:
  - collateral USDT
  - borrowed asset: CAKE
  - repay amount: $17.5 (getting $19.25 of USDT)

> This way, the final position of the borrower will be:

- > Collateral:
  - USDC: $0.75
  - USDT: $0.75
- Borrow CAKE: $0 <— that is required at the end of the `Comptroller.liquidateAccount`, and it would be impossible to get only using one of the available collaterals

> We will apply the mitigation suggested by the warden.

[M-05] Bad Debt in PoolLens.sol#getPoolBadDebt() is not calculated correctly in USD

Submitted by peanuts*, also found by* jasonxiale*,* 0xStalin *and* volodya*.*

Proof of Concept

In `PoolLens.sol#getPoolBadDebt()`, bad debt is calculated as such:

```
            badDebt.badDebtUsd =
                VToken(address(markets[i])).badDebt() *
                priceOracle.getUnderlyingPrice(address(markets[i
            badDebtSummary.badDebts[i] = badDebt;
            totalBadDebtUsd = totalBadDebtUsd + badDebt.badDebtU
```

In `Shortfall.sol#\_startAuction()`, bad debt is calculated as such:

```
        uint256[] memory marketsDebt = new uint256[](marketsCour
        auction.markets = new VToken[](marketsCount);

        for (uint256 i; i < marketsCount; ++i) {
            uint256 marketBadDebt = vTokens[i].badDebt();

            priceOracle.updatePrice(address(vTokens[i]));
            uint256 usdValue = (priceOracle.getUnderlyingPrice(a

            poolBadDebt = poolBadDebt + usdValue;
```

Focus on the line with the `priceOracle.getUnderlyingPrice`. In `PoolLens.sol#getPoolBadDebt`, `badDebt` in USD is calculated by multiplying the bad debt of the VToken market by the underlying price. However, in `Shortfall`, `badDebt` in USD is calculated by the bad debt of the VToken market by the underlying price and divided by 1e18.

The `PoolLens#getPoolBadDebt()` function doesn't divide the debt in USD by 1e18.

This is what the function is actually counting:

Let's say that the VToken market has a `badDebt` of 1.3 ETH (1e18 ETH). The pool intends to calculate 1.3 ETH in terms of USD, so it calls the oracle to determine the price of ETH. Let's say the price of ETH is 1500 USD. The total pool debt should be 1.3 * 1500 = 1950 USD. In decimal calculation, the pool debt should be 1.3e18 * 1500e18 (if oracle returns in 18 decimal places) / 1e18 = 1950e18.

The badDebt in USD in `PoolLens.sol#getPoolBadDebt()` will be massively inflated.

Tools Used

VSCode

Recommended Mitigation Steps

Normalize the decimals of the bad debt calculation in `getPoolBadDebt()`.

```
                badDebt.badDebtUsd =
                    VToken(address(markets[i])).badDebt() *
+                   priceOracle.getUnderlyingPrice(address(markets[i
                badDebtSummary.badDebts[i] = badDebt;
                totalBadDebtUsd = totalBadDebtUsd + badDebt.badDebtU
```

Assessed type

Decimal

chechu (Venus) confirmed

[M-06] Potential Unjust Liquidation After Exiting Market

Submitted by **0xcm**, *also found by* **thekmj** *and* **bin2chen**.

Users might face unjust liquidation of their assets even after exiting a particular market. This could lead to potential financial losses for users, and it might undermine the trust and reputation of the platform.

Proof of Concept

Consider a user with the following financial status:

- Collateral: 1 Bitcoin (BTC), worth `$` 30,000, and 10,000 USDT, worth `$` 10,000.
- Outstanding loan: 1 Ethereum (ETH), worth `$` 3,000.
- The user decides to remove their risk from BTC volatility and exits the BTC market. As per the protocol's rules, exiting the market should remove BTC from their collateral.
- Following the user's exit from the BTC market, a sharp rise in the ETH price occurs, and it surpasses `$` 10,000.
- Due to the increase in ETH price, the system identifies that the user's collateral (now only 10,000 USDT) is insufficient to cover their loan, leading to an insufficient collateralization rate.
- Despite the user's exit from the BTC market, the system still triggers a liquidation process to liquidate BTC as collateral.

In reality, if the BTC was still part of the user's collateral, the total collateral value would have been `$` 40,000 (`$` 30,000 from BTC and `$` 10,000 from USDT). This total value would be sufficient to cover the ETH loan even with the price surge of ETH. Therefore, the user should not have faced liquidation.

This can be traced back to the missing membership check in `preLiquidateHook` function which does not consider if the user has exited the market or not.

https://github.com/code-423n4/2023-05-venus/blob/8be784ed9752b80e6f1b8b781e2e6251748d0d7e/contracts/Comptroller.sol#L424-L526

```
    function preLiquidateHook(
            address vTokenBorrowed,
            address vTokenCollateral,
            address borrower,
            uint256 repayAmount,
            bool skipLiquidityCheck
        ) external override {
            // Pause Action.LIQUIDATE on BORROWED TOKEN to prevent ]
            // If we want to pause liquidating to vTokenCollateral,
            // Action.SEIZE on it
```

```
    _checkActionPauseState(vTokenBorrowed, Action.LIQUIDATE)

    oracle.updatePrice(vTokenBorrowed);
    oracle.updatePrice(vTokenCollateral);

    if (!markets[vTokenBorrowed].isListed) {
        revert MarketNotListed(address(vTokenBorrowed));
    }
    if (!markets[vTokenCollateral].isListed) {
        revert MarketNotListed(address(vTokenCollateral));
    }

    uint256 borrowBalance = VToken(vTokenBorrowed).borrowBal

    /* Allow accounts to be liquidated if the market is depr
    if (skipLiquidityCheck || isDeprecated(VToken(vTokenBorr
        if (repayAmount > borrowBalance) {
            revert TooMuchRepay();
        }
        return;
    }

    /* The borrower must have shortfall and collateral > thr
    AccountLiquiditySnapshot memory snapshot = _getCurrentLi

    if (snapshot.totalCollateral <= minLiquidatableCollatera
        /* The liquidator should use either liquidateAccount
        revert MinimalCollateralViolated(minLiquidatableColl
    }

    if (snapshot.shortfall == 0) {
        revert InsufficientShortfall();
    }

    /* The liquidator may not repay more than what is allowe
    uint256 maxClose = mul_ScalarTruncate(Exp({ mantissa: cl
    if (repayAmount > maxClose) {
        revert TooMuchRepay();
    }
}

/**
 * @notice Checks if the seizing of assets should be allowed
 * @param vTokenCollateral Asset which was used as collatera
 * @param seizerContract Contract that tries to seize the as
 * @param liquidator The address repaying the borrow and sei
```

```
     * @param borrower The address of the borrower
     * @custom:error ActionPaused error is thrown if seizing thi
     * @custom:error MarketNotListed error is thrown if either c
     * @custom:error ComptrollerMismatch error is when seizer cc
     * @custom:access Not restricted
     */
    function preSeizeHook(
        address vTokenCollateral,
        address seizerContract,
        address liquidator,
        address borrower
    ) external override {
        // Pause Action.SEIZE on COLLATERAL to prevent seizing i
        // If we want to pause liquidating vTokenBorrowed, we sh
        // Action.LIQUIDATE on it
        _checkActionPauseState(vTokenCollateral, Action.SEIZE);

        if (!markets[vTokenCollateral].isListed) {
            revert MarketNotListed(vTokenCollateral);
        }

        if (seizerContract == address(this)) {
            // If Comptroller is the seizer, just check if colla
            // is equal to the current address
            if (address(VToken(vTokenCollateral).comptroller())
                revert ComptrollerMismatch();
            }
        } else {
            // If the seizer is not the Comptroller, check that
            // listed market, and that the markets' comptrollers
            if (!markets[seizerContract].isListed) {
                revert MarketNotListed(seizerContract);
            }
            if (VToken(vTokenCollateral).comptroller() != VToker
                revert ComptrollerMismatch();
            }
        }

        // Keep the flywheel moving
        uint256 rewardDistributorsCount = rewardsDistributors.le

        for (uint256 i; i < rewardDistributorsCount; ++i) {
            rewardsDistributors[i].updateRewardTokenSupplyIndex(
            rewardsDistributors[i].distributeSupplierRewardToker
            rewardsDistributors[i].distributeSupplierRewardToker
        }
```

```
        }
```

In essence, the user is punished for market volatility, even after they have taken steps to protect themselves (by exiting the BTC market).

Recommended Mitigation Steps

Update the `preLiquidateHook` function to check if a user has exited a market before proceeding with liquidation.

Assessed type

Invalid Validation

chechu (Venus) disputed and commented:

> This is the desired behaviour. Even if user exits a market, we expect user to maintain healthy position to protect the protocol.

Oxean (judge) commented:

> @chechu - By exiting the market, the user is no longer expecting those assets to be used as collateral.

```
    * @notice Removes asset from sender's account liquidity calcula
```

> So while the user is still expected to maintain a healthy position with the assets they have marked as being part of their collateral, do you agree that assets that are not being used as part of their collateral should not be seized?

chechu (Venus) confirmed and commented:

> We have been reviewing this topic internally. Compound allows the seizing of tokens from markets not enabled as collateral. And our code does the same.

> But, we added the sentence `disabling them as collateral` in the `Comptroller.exitMarket` function, so we think it's fair to forbid seizing if the

> borrower didn't enable the market as collateral.

> So, I would say the issue is valid, and we'll work to mitigate it (we'll add the check in the `preSeizeHook`, which is also used in the `healAccount` flow).

[M-07] DOS attack prevents refunding previous bid in Shortfall.sol and malicious bidder always wins the auction

Submitted by berlin-101, *also found by* Emmanuel, YungChaza, Emmanuel, sashik_eth, Team_Rocket, fs0c, fs0c, Audit_Avengers_2, 0xadrii *and* bin2chen.

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Shortfall/Shortfall.sol#L183
https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Shortfall/Shortfall.sol#L190

The auction logic in `Shortfall.sol` refunds the previously accepted (highest) bid when a new acceptable bid is placed via the `placeBid` function.

It is important that this refund succeeds as otherwise a new acceptable (higher) bid is not possible and the auction is disrupted which consequently makes the current highest bidder the auction winner and causes a loss for the Venus project and its users.

When refunding the `safeTransfer` of OpenZeppelin `SafeERC20Upgradeable` (inheriting from `SafeERC20`) is used which deals with the multiple ways in which different ERC-20 (BEP-20) tokens indicate the success/failure of a token transfer.

For details see: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/utils/SafeERC20.sol#L12

Nevertheless, there are additional scenarios that may still disrupt the auction and put it into a state of DOS (Denial of Service). Specifically, 2 scenarios were identified:

1. DOS with the underlying token implementing a blacklist.
2. DOS with the underlying token being an ERC20-compatible ERC777 token.

Proof of Concept

1. DOS with the underlying token implementing a blacklist

In this scenario, the underlying token is implemented with a blacklist (also known as blocklist).

Because this is common for tokens on the Ethereum network (e.g. USDC/USDT implementing blacklist/blocklist; See: https://github.com/d-xo/weird-erc20) this is a scenario also possible for tokens on the Binance Chain.

And since it is not specifically stated that such tokens are excluded from the Venus project, while the fee on transfer/deflationary/rebase tokens are specifically mentioned to be excluded, this is assumed to be a potential issue.

The following steps describe the issue:

1. Bidder 1 makes a bid while he is not on the token blacklist.
2. After the bid, he is put on the token blacklist.
3. Bidder 2 makes a higher bid and the refund to bidder 1 is attempted.
4. The refund reverts due to bidder 1 being blacklisted which blocks the token transfer back.
5. Bidder 1 remains the highest bidder and wins the auction.

2. DOS with the underlying token being an ERC20-compatible ERC777 token.

In this scenario, the underlying token is an ER777 token instead of an ERC20. Since the audit does not specifically state that ERC777 tokens (which are ERC-20 compatible) are out of scope, this is assumed to be a potential issue.

See https://docs.openzeppelin.com/contracts/2.x/api/token/erc777 for details on ERC777 tokens.

The following steps describe the issue:

1. Bidder 1 implements a contract that acts as an “ERC777 recipient” which can either accept/reject tokens that are transferred to it.
2. Bidder 1 makes a bid not with an EOA (externally owned account) but uses his smart contract to make the bid.

3. After the bid was accepted, he activates his smart contract and rejects any tokens transferred to it.
4. Bidder 2 makes a higher bid and the refund to the smart contract of bidder 1 is attempted.
5. The refund fails due to the smart contract of bidder 1 rejecting the token transfer (ERC777 token calls `tokensReceived` function of receiving a smart contract to finalize the token transfer which reverts).
6. Bidder 1 remains the highest bidder and wins the auction.

Coded POC

To prove both aforementioned scenarios of putting the auction into a state of DOS, the `Shortfall.ts` test was modified and 1 test case for each scenario was added. Code for additional required mock tokens etc. (`MockTokenERC20Blacklistable.sol`, `MockTokenERC777.sol`, `ERC777Recipient.sol`) are included.

Note: see **DOS attack prevents refunding previous bid in Shortfall.sol and malicious bidder always wins the auction** *for coding details.*

Recommended Mitigation Steps

Use a withdrawal pattern ("pull over push") instead of directly refunding the highest bidder during the bid. See: **https://fravoll.github.io/solidity-patterns/pull_over_push.html** for details. This way, the auction will not get into a state of DOS.

Assessed type

DoS

chechu (Venus) confirmed and commented via duplicate issue #376:

> We won't accept ERC777 tokens as underlying tokens. But we have upgradable ERC20 tokens that can include a similar behavior, so the risk exists and we'll try to mitigate it by applying some changes.

[M-08] Borrower can cause a DoS by frontrunning a liquidation and repaying as low as 1 wei of the current debt

Submitted by OxStalin*, also found by* J4de *and* rvierdiiev.

Borrowers can cause DoS when the liquidator attempts to liquidate 100% of the borrower's position. The borrower needs to frontrun the liquidation tx and repay a slight portion of the debt, paying as low as 1 wei will make the **borrowBalance** to be less than what it was when the liquidator sent the tx to liquidate the position.

Proof of Concept

If a liquidator intends to liquidate the entire position, but the borrower frontruns the liquidator's transaction and repays an insignificant amount of the total debt, will cause the **borrowBalance** to be less than it was when the liquidator sent its transaction; thus, will cause the value of the **maxClose** variable to be less than the `repayAmount` that the liquidator set to liquidate the whole position, which will end up causing the tx to be reverted because of **this validation**

Example:

- There is a position of 100 WBNB to be liquidated, the liquidator sends the `repayAmount` as whatever the `maxClose` was at that point, the borrower realizes that 100% of its position will be liquidated and then frontruns the liquidation transaction by repaying an insignificant amount of the total borrow.
- When the liquidation transaction is executed, the `maxClose` will be calculated based on the new `borrowBalance`, which will cause the calculation of `maxClose` to be less than the total `repayAmount` that was sent, and the transaction will be reverted even though the position is still in a liquidation state

Recommended Mitigation Steps

Instead of **reverting the tx if the repayAmount is greater than maxClose**, recalculate the final `repayAmount` to be paid during the execution of the liquidation and return this calculated value back to the function that called the `preLiquidateHook()`.

```
    function preLiquidateHook(
        ...
```

```
    uint256 repayAmount,
    ...
+   ) external override returns(uint256 repayAmountFinal) {
    ...
    ...
-   if (repayAmount > maxClose) {
-      revert TooMuchRepay();
-   }
+   repayAmountFinal = repayAmount > maxClose ? maxClose : repay
```

Assessed type

DoS

chechu (Venus) disagreed with severity and commented:

> Suggestion: Med

> Front running attacks can be easily avoided by the liquidator, connecting to the right nodes and using private mempools. Moreover, the borrower will need to spend gas to defend their position against several liquidators.

Oxean (judge) decreased severity to Medium and commented:

> I think Med makes sense here, but will be open to warden comments during QA process.

[M-09] ShortFall contract might transfer an incorrect amount of tokens to the highest bidder.

Submitted by fs0c, *also found by* yongskiws, BPZ, rvierdiiev, BPZ, rvierdiiev, J4de, Team_Rocket, rvierdiiev, rvierdiiev, peanuts, Brenzee *and* Oxnev.

There might be an incorrect amount of transfer possible if `convertibleBaseAsset` is not a token which is pegged to USD.

There is not much information on what `convertibleBaseAsset` is supposed to be. If it is a token which is not pegged to USD, then the auction process might transfer

wrong amount of tokens or entirely wrong tokens.

Let's take an example of `LARGE_RISK_FUND` type of auction for simplicity. Assuming the `convertibleBaseAsset` is not a token pegged to USD (let's take it as BNB, for this case).

Now the calculation of `poolBadDebt` is calculated by converting the `badDebt` to usd terms in the code below:

```
    for (uint256 i; i < marketsCount; ++i) {
                uint256 marketBadDebt = vTokens[i].badDebt();

                priceOracle.updatePrice(address(vTokens[i]));
                uint256 usdValue = (priceOracle.getUnderlyingPrice(a

                poolBadDebt = poolBadDebt + usdValue;
                auction.markets[i] = vTokens[i];
                auction.marketDebt[vTokens[i]] = marketBadDebt;
                marketsDebt[i] = marketBadDebt;
            }
```

In this case, the auction properties would be as follows:

```
    auction.seizedRiskFund = incentivizedRiskFundBalance;
    auction.startBlock = block.number;
    auction.status = AuctionStatus.STARTED;
    auction.highestBidder = address(0);
```

Where `incentivizedRiskFundBalance = poolBadDebt + ((poolBadDebt * incentiveBps) / MAX_BPS);`

```
    function closeAuction(address comptroller) external nonReentrant
            Auction storage auction = auctions[comptroller];
                                    // ...
            if (auction.auctionType == AuctionType.LARGE_POOL_DEBT)
                riskFundBidAmount = auction.seizedRiskFund;
            } else {
                riskFundBidAmount = (auction.seizedRiskFund * auctic
```

```
            }

            uint256 transferredAmount = riskFund.transferReserveFor
            IERC20Upgradeable(convertibleBaseAsset).safeTransfer(auc
        }
```

When the `closeAuction` is called, the contract will transfer the `riskFundBidAmount` of `convertibleBaseAsset` to the highest bidder. Here, if the `convertibleBaseAsset` token is not a token pegged to USD, it will transfer those tokens to the highest bidder, where it should have transferred the tokens that amount to that value.

Example:

- `convertibleBaseAsset` = TokenA (the price of this token is `$` 100 per 1e18 tokens).
- `PoolBadDebt` = 200 * ie18, which should be equal to `$` 200 as `poolbaddebt` is calculated in USD.
- `seizedRiskFund` = 220 * 1e18.

Assume the auction type is `LARGE_RISK_FUND` and `highestBidBps` = 10000.

At the auction complete the tokens transferred to the highestbidder would be:

- `riskFundBidAmount` = 220 * 1e18 * 10000/10000 = 220 * 1e18
- Actual price of tokens transferred to the highestbidder = 220 * 100 = `$` 22000
- Token amount that should be transferred = `$` 220.

Here, the tokens are directly transferred before converting them into the terms of `convertibleBaseAsset` which causes the main issue.

Recommendation

Before transferring the amount to the highest bidder, ping the Oracle for the correct price of `convertibleBaseAsset` and then convert the `riskFundBidAmount` in the terms of `convertibleBaseAsset` Tokens. Even if a token pegged to USD is used, the Oracle should be used to get the correct value AND the tokens should always be

converted in terms of `convertibleBaseAsset`, as sometimes the pegged tokens might also divert from their price or decimals might be different for different tokens.

Assessed type

Token-Transfer

chechu (Venus) confirmed

[M-10] Exchange Rate can be manipulated

Submitted by LokiThe5th, *also found by* thekmj, Parad0x, Josiah, J4de, 0x8chars, qpzm, RaymondFam, Cryptor, fs0c, fs0c, QiuhaoLi, Norah, CoOnan, xuwinnie, bin2chen *and* volodya.

https://github.com/code-423n4/2023-05-venus/blob/8be784ed9752b80e6f1b8b781e2e6251748d0d7e/contracts/VToken.sol#L1463
https://github.com/code-423n4/2023-05-venus/blob/8be784ed9752b80e6f1b8b781e2e6251748d0d7e/contracts/VToken.sol#L1421
https://github.com/code-423n4/2023-05-venus/blob/8be784ed9752b80e6f1b8b781e2e6251748d0d7e/contracts/VToken.sol#L756

A malicious user can manipulate the protocol to receive greater rewards from the `RewardsDistributor` than they should. To achieve this, the attacker manipulates the `exchangeRate`.

The attacker `mint`s into the `VToken` contract legitimately but also `transfer`s an amount of tokens directly to the `VToken` contract. This inflates the `exchangeRate` for all subsequent users who `mint` and has the following impact:

1. Allows the attacker to push their leverage past the market `collateralFactor`
2. Violates the internal accounting when `borrowing` and `repaying`, causing the `totalBorrows` and the sum of the individual account borrows to become out

of sync. I.e. the `totalBorrows` can become `0` while there are still some loans outstanding, leading to loss of earned interest.

3. The attacker can use the leverage to repeatedly `borrow` + `mint` into the `VToken` in order to inflate their share of the token rewards issued by the rewards distributor.

This means that all subsequent `minter`s receive less `VTokens` than they should.

The attack cost is the loss of the `transfer` into the `VToken` contract. But it must be noted that the attacker still receives around 65-75% of the (`attackTokens` + `mintTokens`) back.

The permanent side-effect of this exploit is that the minting of `VTokens` to any subsequent users remains stunted, as there is no direct mechanism to clear the excess underlying tokens from the contract. This can taint this Pool permanently.

This exploit becomes more profitable as block count accrues and more `REWARD_TOKENS` are issued, or, for example, if VENUS sets up greater rewards to incentivize supplying into a particular Pool; these increased rewards are a normal practice in DeFi and could be a prime target for this manipulation.

Proof of Concept

In this scenario an attacker:

1. Needs ~60 underlying tokens (supply 10 underlying, 20 direct transfer, 30 interest).
2. Gets ~5 times more rewards than other users.
3. Is still able to withdraw ~50 underlying tokens from the `VToken`.
4. ~10 tokens are now stuck in the contract, permanently tainting the exchange rate.

A detailed Proof of Concept illustrating the case can be found in this gist.

The gist simulates and walks through the attack using the repo's test suite as a base. The exploit is commented on throughout its various steps.

Tools Used

Manual Code Review. Hardhat + modified tests from repo.

Recommended Mitigation Steps

When contract calculations depend on calls to an `ERC20.balanceOf`, there is always a risk of a malicious user sending tokens directly to the contract to manipulate the calculations to their benefit.

The simplest solution would be to have a check that the expected amount of underlying is equal to the actual amount of underlying, and if not, have the `mint` function sweep these additional underlying tokens into the next `minter's` calculations, reducing the economic incentive and eliminating the exaggerated effect on the exchange rate.

Assessed type

Token-Transfer

chechu (Venus) disputed and commented via duplicate issue #314:

> The attack would indeed be feasible if we didn't require an initial supply.

0xean (judge) commented via duplicate issue #314:

> @chechu - can you point me to this in the codebase?

chechu (Venus) commented via duplicate issue #314:

> Our fault, we **allow** an initial supply, but we don't **require** it.
> https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Pool/PoolRegistry.sol#L321

> The origin of the confusion is that we'll provide, for sure, an initial supply on every market that we'll add to the `PoolRegistry`, and the process to add new markets is under the control of the Governance (so, the community will have to vote for it). For that reason, we really assumed that there will be an initial supply, but now we realized we are not requiring it in the code. We'll do it, just to avoid any confusion or potential error.

> We won't integrate the Oracles, but the initial idea is to provide at least `$ 10,000` as an initial supply on each new market. That "check" will be done externally, when the VIP is prepared to be proposed to the community.

chechu (Venus) commented:

1. Allows the attacker to push their leverage past the market collateralFactor

> This is wrong. If you mint (receiving X `vTokens`) and then you transfer underlying tokens to the market, your X `vTokens` will have a greater value because the exchange rate is greater after the donation. In the PoC:

1. The attacker mints 10 WBTC -> `vTokens` received: 10
2. The attacker donates 20 WBTC -> this change the exchange rate from 100000000000000000 to 200000000000000000, so, basically, the `vTokens` previously minted now can be redeemed receiving the double amount of WBTC (that is the expected effect of this donation).

> So, the value of the 10 `vTokens` of the attacker after the donation is 20 WBTC, not 10 WBTC. For that reason, the user can borrow 13 WBTC (13 < 20 * 0.7, where 0.7 is the collateral factor).

> The attacker could get a similar (actually better) effect minting more `vTokens` supplying the 20 WBTC tokens, instead of donating them. Moreover, donating is a benefit for every user with `vTokens` before the donation, while minting only benefits the minter.

> To demonstrate it, you can replace the following statement in the PoC:

```
await mockWBTC.connect(attacker).transfer(vWBTC.address, convert
```

> with this one:

```
await vWBTC.connect(attacker).mint(convertToUnit(20, 8));
```

> And the output of the PoC will be the same.

2. Violates the internal accounting when borrowing and repaying causing the `totalBorrows` and the sum of the individual account borrows to become out of sync. I.E. the `totalBorrows` can become 0 while their are still some loans outstanding, leading to loss of earned interest.

> This is not because of the transfer (donation), but because of the known rounding issues associated with the used math; which can generate small differences among the `totalBorrow` variable and the sum of the individual borrowed amounts.

3. The attacker can use the leverage to repeatedly borrow + mint into the `VToken` in order to inflate their share of the token rewards issued by the rewards distributor.

> That is true, but, again, it's independent of the donations. Users can use the leverage to increase their positions and therefore get more rewards. The main downside of leveraging is the cost (every X `WBTC` borrowed that are then supplied implies a cost for the user proportional to the reserve factor of the market). So, taking into account that the total rewards to distribute are fixed, the leverage can make sense depending on the total suppliers and borrowers.

> The permanent side-effect of this exploit is that the minting of `VTokens` to any subsequent users remains stunted as there is no direct mechanism to clear the excess underlying tokens from the contract. This can taint this `Pool` permanently.

> Donations to markets are supported, and there aren't known negative side effects on regular scenarios. If the liquidity of the market is very low, donations can facilitate issues related to rounding (like in the Hundred Finance attack, https://twitter.com/danielvf/status/1647329491788677121), but every market in Venus starts with a minimum liquidity that should reduce these risks.

> Finally, in the PoC, some redeems operations fail because those users still have some borrowed amount. Printing the error thrown you can see how the error is `InsufficientLiquidity`, thrown in the `Comptroller._checkRedeemAllowed` function. To repay 100% of the debt, the best option is to invoke the `repayBorrow` providing an big amount as parameter. The function will get only the borrowed amount (considering interest accrued until that block).

thebrittfactor (C4) commented:

> Sponsor requested additional feedback from the warden in regards to this submission after the Post-Judging QA period. C4 staff reached out to the warden directly with that request.

LokiThe5th (warden) commented:

> Thank you for the feedback. I don't have access to my original notes anymore, but will try to provide clarity where I can. To be clear, in retrospect, this submission seems to have conflated a few issues while trying to demonstrate the exchange rate issue.

> This is wrong. If you mint (receiving X `vTokens`) and then you transfer underlying tokens to the market, your X `vTokens` will have a greater value because the exchange rate is greater after the donation.

> Yes, you are correct. The exchange rate is manipulated (which is the issue). To be more specific, the attacker *appears* to be able to push past the collateral factor when *considering the amount of* `vToken` *held by the attacker*. The intention here is to demonstrate this exchange rate manipulation through borrowing past what the internal accounting would hold the attacker's safe collateral factor would be.

> This is not because of the transfer (donation), but because of the known rounding issues associated with the used math; which can generate small differences among the `totalBorrow` variable and the sum of the individual borrowed amounts.

> Indeed, rounding in Solidity is a known issue. It may well be that the direct transfer only served to exacerbate this issue when compared with the control scenario.

> That is true, but, again, it's independent of the donations. Users can use the leverage to increase their positions and therefore get more rewards. The main downside of leveraging is the cost (every X `WBTC` borrowed that are then supplied implies a cost for the user proportional to the reserve factor of the market). So, taking into account that the total rewards to distribute are fixed, the leverage can make sense depending on the total suppliers and borrowers.

> You are correct that this is independent of donations. Users using leverage in this way to increase their rewards is likely a separate issue.

> Donations to markets are supported, and there aren't known negative side effects on regular scenarios. If the liquidity of the market is very low, donations can facilitate issues related to rounding (like in the Hundred Finance attack, https://twitter.com/danielvf/status/1647329491788677121), but every market in Venus starts with a minimum liquidity that should reduce these risks.

> In the context of modular markets exchange rate manipulation can be damaging. It is good practice to explicitly handle (or not handle) donations in the accounting logic for the contract. For example, in the standard `UniswapV2Pair` contracts calculations are made using an internal tracking of `reserves` to avoid this issue.

> Finally, in the PoC, some redeems operations fail because those users still have some borrowed amount. Printing the error thrown you can see how the error is `InsufficientLiquidity`, thrown in the `Comptroller._checkRedeemAllowed` function. To repay 100% of the debt, the best option is to invoke the `repayBorrow` providing an big amount as parameter. The function will get only the borrowed amount (considering interest accrued until that block).

> You are correct. Some redeems fail because some users still have borrowed amounts. In the preceeding code these users tried repay their borrows using their exact `borrowBalance` from the call to `VToken.getAccountSnapshot(user)`. It would be acceptable for a user to assume that should they try to `repayBorrow` with this outstanding amount. If memory serves, this failure of repayment using the returned `borrowBalance` happened in exchange manipulation scenarios, but not others. But this may have been a mistaken assumption if that is not the case. If so, it would also be a separate issue.

chechu (Venus) commented:

> Hey @LokiThe5th - Thanks for your message.

> To be more specific, the attacker appears to be able to push past the collateral factor when considering the amount of `vToken` held by the attacker.

> I think that is not precise. The donation doesn't allow users to break the rule of the collateral factor. The donation is increasing the value of the `vTokens`, so any user with `vTokens` before the donation will be able to borrow more tokens. That is correct, expected, and doesn't generate any issue.

> You call it manipulation, and I can see your point because with a donation the user is able to change the value of the exchange rate. Personally, I don't think this is a manipulation, because the user doesn't get any benefit by doing it. As I said, if the attacker mints instead of donating the same amount, he would be able to borrow more tokens (in a regular scenario, not being the first and only supplier).

> Example:

- Initial exchange rate: 1 (1 underlying token == 1 `vToken`)
- User 1 mints 1,000 tokens, receiving 1,000 `vTokens` . Exchange rate is not affected, so, it's 1
- Attacker 1 mints 1,000 tokens, receiving 1,000 `vTokens` . Exchange rate is not affected, so, it's 1
- Attacker 1 donates 2,000 tokens, not receiving anything, but changing the exchange rate, that now will be 2 (total cash / total `vTokens` minted)

> So, now the 1,000 `vTokens` have more value (the attacker would be able to redeem 1,000 `vTokens` and receive 2,000 tokens, instead of the original 1,000 tokens they minted). And therefore, the "borrowing power" of the attacker is greater. The attacker can borrow more assets from another market, because now his 1,000 `vTokens` has more value.

> But, that is a bad strategy by the attacker, because by doing the donation User 1 also received a benefit. Now, User 1 can redeem their 1,000 `vTokens` , receiving 2,000 tokens. Not only their original 1,000 tokens.

> A better strategy by the attacker would be to mint 2,000 tokens, instead of donating them. This way, the exchange rate doesn't change (so User 1 doesn't receive any benefit) and the "borrowing power" of the attacker is even higher (3,000 tokens, instead of 2,000 tokens achieved via the donation).

> So, yes, with a donation you are able to update the exchange rate, but you won't get any benefit, and you will lose resources.

> Indeed, rounding in Solidity is a known issue. It may well be that the direct transfer only served to exacerbate this issue when compared with the control scenario.

> If you mint instead of donating, the rounding issue appears too. So, I don't think the donation exacerbates the rounding issue.

> In the context of modular markets exchange rate manipulation can be damaging. It is good practice to explicitly handle (or not handle) donations in the accounting logic for the contract. For example, in the standard `UniswapV2Pair` contracts calculations are made using an internal tracking of reserves to avoid this issue.

> In the Venus protocol, I think donations benefit every `vToken` holder and don't affect future holders, because for a user getting `vTokens` , the relevant events happen from the `vTokens` are minted until they are redeemed. It doesn't matter what happened before. Moreover, the exchange rate is never decreasing. So, IMO, we can avoid the internal tracking of cash in the markets.

> this failure of repayment using the returned `borrowBalance` happened in exchange manipulation scenarios, but not others.

> I think the failures of repayments are associated with the rounding issues, not with donations. I modified the provided PoC, transforming the donation into a mint, and this issue is still there. I think the impact is low because users are not able to repay 100% of their debt only in edge cases, with 1 or 2 borrowers in the market and after several blocks. With a regular number of borrowers, users shouldn't have any problem repaying their debt, and therefore redeeming their `vTokens`

> Thank you again for your time reviewing the code. We really appreciate it. Your comments push us to improve the code (and to understand it better, tbh). We are totally open to trying to clarify any doubts.

[M-11] `RiskFund.swapPoolsAsset` does not allow the user to supply deadline, which may cause swap revert

Submitted by Oxnev*, also found by* OxStalin*,* BugBusters *and* chaieth.

Not allowing users to supply their own deadline could potentially expose them to sandwich attacks.

Proof of Concept

RiskFund.sol#L174

```
function swapPoolsAssets(
    address[] calldata markets,
    uint256[] calldata amountsOutMin,
    address[][] calldata paths
) external override returns (uint256) {
    _checkAccessAllowed("swapPoolsAssets(address[],uint256[],adc
    require(poolRegistry != address(0), "Risk fund: Invalid pool
    require(markets.length == amountsOutMin.length, "Risk fund:
    require(markets.length == paths.length, "Risk fund: markets

    uint256 totalAmount;
    uint256 marketsCount = markets.length;

    _ensureMaxLoops(marketsCount);

    for (uint256 i; i < marketsCount; ++i) {
        VToken vToken = VToken(markets[i]);
        address comptroller = address(vToken.comptroller());

        PoolRegistry.VenusPool memory pool = PoolRegistry(poolRe
        require(pool.comptroller == comptroller, "comptroller dc
        require(Comptroller(comptroller).isMarketListed(vToken),

        uint256 swappedTokens = _swapAsset(vToken, comptroller,
        poolReserves[comptroller] = poolReserves[comptroller] +
        totalAmount = totalAmount + swappedTokens;
    }

    emit SwappedPoolsAssets(markets, amountsOutMin, totalAmount)

    return totalAmount;
}
```

In `RiskFund.swapPoolsAsset`, there is a parameter to allow users to supply slippage through `amountOutMin`, but does not allow the user to include a deadline check when swapping pool assets into base assets, in the event that pool assets are not equal to `convertibleBaseAsset`.

```
uint256 swappedTokens = _swapAsset(vToken, comptroller, amountsC
```

In `RiskFund._swapAsset`, there is a call to `IPancakeswapV2Router(pancakeSwapRouter).swapExactTokensForTokens()`, but the `deadline` parameter is simply passed in as the current `block.timestamp`, in which the transaction occurs. This effectively means that the transaction has no deadline, which means that swap transactions may be included anytime by validators and remain pending in mempool, potentially exposing users to sandwich attacks by attackers or MEV bots.

RiskFund.sol#L265

```
    function _swapAsset(
        VToken vToken,
        address comptroller,
        uint256 amountOutMin,
        address[] calldata path
    ) internal returns (uint256)
                ...
                ...
                if (underlyingAsset != convertibleBaseAsset) {
                    require(path[0] == underlyingAsset, "RiskFund: s
                    require(
                        path[path.length - 1] == convertibleBaseAsse
                        "RiskFund: finally path must be convertible
                    );
                    IERC20Upgradeable(underlyingAsset).safeApprove(p
                    IERC20Upgradeable(underlyingAsset).safeApprove(p
                    uint256[] memory amounts = IPancakeswapV2Router(
                        balanceOfUnderlyingAsset,
                        amountOutMin,
                        path,
                        address(this),
                        /// @audit does not allow deadline to be pas
                        block.timestamp
                    );
                    ...
                    ...
```

Consider the following scenario:

1. Alice wants to swap 30 vBNB tokens for 1 BNB and later sell the 1 BNB for 300 DAI. She signs the transaction calling `RiskFund.swapPoolsAsset()` with

inputAmount = 30 vBNB and `amountOutmin` = 0.99 BNB to allow for 1% slippage.

2. The transaction is submitted to the mempool; however, Alice chose a transaction fee that is too low for validators to be interested in including her transaction in a block. The transaction stays pending in the mempool for extended periods, which could be hours, days, weeks, or even longer.
3. When the average gas fee dropped far enough for Alice's transaction to become interesting again for miners to include it, her swap will be executed. In the meantime, the price of BNB could have drastically decreased. She will still at least get 0.99 BNB due to `amountOutmin`, but the DAI value of that output might be significantly lower. She has unknowingly performed a bad trade due to the pending transaction she forgot about.

An even worse way this issue can be maliciously exploited is through MEV:

1. The swap transaction is still pending in the mempool. Average fees are still too high for validators to be interested in it. The price of BNB has gone up significantly since the transaction was signed, meaning Alice would receive a lot more ETH when the swap is executed. But that also means that her `minOutput` value is outdated and would allow for significant slippage.
2. A MEV bot detects the pending transaction. Since the outdated `minOut` now allows for high slippage, the bot sandwiches Alice, resulting in significant profit for the bot and significant loss for Alice.

Tools Used

Manual Analysis

Recommendation

Allow users to supply their own deadline parameter within `RiskFund.swapPoolsAsset`.

[chechu (Venus) acknowledged]

[M-12] Fix utilization rate computation

Submitted by SaeedAlipoor01988*, also found by* lanrebayode77.

The `BaseJumpRateModelV2.sol#L131.utilizationRate()` function can return a value above 1 and not between [0, BASE].

Proof of Concept

In the `BaseJumpRateModelV2.sol#L131.utilizationRate()` function, cash and borrows and reserves values get used to calculate the utilization rate between [0, 1e18]. Reserves are currently unused but it will be used in the future.

```
     */
    function utilizationRate(
        uint256 cash,
        uint256 borrows,
        uint256 reserves
    ) public pure returns (uint256) {
        // Utilization rate is 0 when there are no borrows
        if (borrows == 0) {
            return 0;
        }

        return (borrows * BASE) / (cash + borrows - reserves);
    }
```

If the borrow value is 0, then the function will return 0, but in this function, the scenario where the value of reserves exceeds cash is not handled. The system does not guarantee that reserves never exceed cash. The reserves grow automatically over time, so it might be difficult to avoid this entirely.

If reserves > cash (and borrows + cash - reserves > 0), the formula for `utilizationRate` above gives a utilization rate above 1.

Recommended Mitigation Steps

Make the utilization rate computation return 1 if reserves > cash.

Assessed type

Math

chechu (Venus) confirmed

[M-13] Comptroller.healAccount doesn't distribute rewards for a healed borrower

Submitted by rvierdiiev.

As a result, the healed account receives less rewards.

Proof of Concept

`Comptroller.healAccount` can be called by anyone in order to fully close accounts. The healer should repay part of account's debt in order to receive all account's collateral. At the end account debt will be cleared.

This is the part when collateral is seized and debt is cleared: https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Comptroller.sol#L611-L625

```
        for (uint256 i; i < userAssetsCount; ++i) {
            VToken market = userAssets[i];

            (uint256 tokens, uint256 borrowBalance, ) = _safeGet
            uint256 repaymentAmount = mul_ScalarTruncate(percent

            // Seize the entire collateral
            if (tokens != 0) {
                market.seize(liquidator, user, tokens);
            }
            // Repay a certain percentage of the borrow, forgive
            if (borrowBalance != 0) {
                market.healBorrow(liquidator, user, repaymentAmo
            }
        }
```

In order to seize the collateral, `market.seize` is called, which will then call `comptroller.preSeizeHook`. And this hook will distribute supply rewards to both accounts.

In order to clear, healed account debt `market.healBorrow` is called. This function will not call any comptroller function. As a result, the debt of the healed account is set to 0, but rewards that were earned by the account before healing were not distributed. So the user lost rewards for that debt amount.

Tools Used

VsCode

Recommended Mitigation Steps

`Comptroller` should distribute rewards to this account that were earned before and only then set debt to 0.

Oxean (judge) commented:

> Will leave open for sponsor comment, I think this would amount to dust given the state of the user's account.

chechu (Venus) confirmed

Oxean (judge) commented:

> @chechu - can you confirm this has more impact than just dust amounts?

chechu (Venus) commented:

> @chechu - can you confirm this has more impact than just dust amounts?

> It can be more than dust amounts. I can imagine a user position like this:
>
> - Collateral: `$` 1M (just with one asset)
> - Borrow: `$` 500K (healthy, generating a significant amount of borrow rewards if no one interacts with this account for a long time)
>
> Then, a black swan happens and the collateral value moves from `$` 1M to `$` 20, so anyone could invoke the `healAccount` function, repaying part (no more than `$` 20) of the loan.

> The current implementation is not distributing the borrow rewards to the borrower, and given the big amount it could be significant. The key point here is, the borrow rewards depend on the loan, not on the collateral (that will be very low, I agree).

[M-14] placeBid() Possible participation in auctions that have been modified

Submitted by bin2chen

`placeBid()` lacks checking if auctions are restarted and participated in that are not expected by the user, which may result in the user losing funds.

Proof of Concept

When the user makes a bid, simply pass in `comptroller` and `bidBps` with the following code:

```
function placeBid(address comptroller, uint256 bidBps) exter
    Auction storage auction = auctions[comptroller];

    require(_isStarted(auction), "no on-going auction");
    require(!_isStale(auction), "auction is stale, restart i
    require(bidBps <= MAX_BPS, "basis points cannot be more
```

Because `comptroller` corresponds to the `Auction` there are two cases that will generate new `Auction` (the new one and the old one may have completely different types and amounts):

1. If the first auction takes too long and no one bids, `_isStale()` can restart the auction.
2. If the auction ends and the last bid time `>nextBidderBlockLimit` , then you can restart the auction after it ends.

Since bidding can be restarted, `placeBid()` is only based on `comptroller` , and it may be possible to participate in the old `Auction` , but end up participating in the new `Auction` ; as the old and new `Auction` may be very different, resulting in the user losing money.

For example:

1. Alice learns about the auction in the UI, auctions = `{type=LARGE_RISK_FUND,debt=100,seizedRiskFund=100}` and submits it to participate in the auction.
2. When Alice commits, her transaction will exist in memorypool.

   Note: Because Alice needs to stay in the UI interface for some time, or because of the GAS price and block size, Alice's transaction is delayed, resulting in a much higher possibility of preemptive execution in step 3:

3. The auction is restarted due to any of the following situations (before Alice's task is executed):

   a) Auction `>waitForFirstBidder` leads to `_isStale()`. Bob executes restarted auction, restarted auction debt increases. E.g.: auction = `{type=LARGE_POOL_DEBT,debt=100000,seizedRiskFund=100}`.

   b) The auction ends `>nextBidderBlockLimit`. Bob restarts the auction, and the restarted auction `seizedRiskFund` becomes small, like 0, the debt is already very high as auction = `{type=LARGE_POOL_DEBT,debt=100,seizedRiskFund=0}`, there may also be still `LARGE_RISK_FUND`.

4. Alice's turn to execute the transaction, this time the new auction, and Alice expected has been much worse, but the transaction will still be executed. The result is that Alice may pay a lot of debt, but get very little `seizedRiskFund`.

So, we need to add a restriction to `placeBid()` to ensure that the auction has not been restarted when the transaction is executed.

A simple way to do this is to add the parameter: `auctionStartBlock`, and compare it to the `auction.startBlock` of the current transaction. If the two are different, then the auction has been restarted and the transaction reverts.

Recommended Mitigation Steps

```
-   function placeBid(address comptroller, uint256 bidBps) exter
+   function placeBid(address comptroller, uint256 bidBps , uint
        Auction storage auction = auctions[comptroller];

+       require(auction.startBlock == auctionStartBlock,"auctior
        require(_isStarted(auction), "no on-going auction");
        require(!_isStale(auction), "auction is stale, restart i
        require(bidBps <= MAX_BPS, "basis points cannot be more
```

Assessed type

Context

Oxean (judge) decreased severity to Medium and commented:

> I don't see this is as being a High, but do follow the wardens logic on why this could be problematic. Will downgrade to Medium and look forward to sponsor comment.

chechu (Venus) confirmed

[M-15] Borrow rate calculation can cause VToken.accrueInterest() to revert, DoSing all major functionality

Submitted by dacian, *also found by* nadin, CoOnan *and* SaeedAlipoor01988.

Borrow rates are calculated dynamically and `VToken.accrueInterest()` reverts if the calculated rate is greater than a hard-coded maximum. As `accrueInterest()` is called by most VToken functions, this state causes a major DoS.

Proof of Concept

VToken hard-codes the maximum borrow rate and `accrueInterest()` reverts if the dynamically calculated rate is greater than the hard-coded value.

The actual calculation is dynamic [1, 2] and takes no notice of the hard-coded cap, so it is very possible that this state will manifest, causing a major DoS due to most

VToken functions calling `accrueInterest()` and `accrueInterest()` reverting.

Recommended Mitigation Steps

Change `VToken.accrueInterest()` to not revert in this case, but simply to set `borrowRateMantissa = borrowRateMaxMantissa` if the dynamically calculated value would be greater than the hard-coded max. This would:

1. Allow execution to continue operating with the system-allowed maximum borrow rate, allowing all functionality that depends upon `accrueInterest()` to continue as normal.
2. Allow `borrowRateMantissa` to be naturally set to the dynamically calculated rate as soon as that rate becomes less than the hard-coded max.

Assessed type

DoS

chechu (Venus) disagreed with severity and commented:

> We could deploy a new implementation of the VToken contract, with a higher maximum, and fix the lock. Via VIP, with the votes from the community.

chechu (Venus) confirmed via duplicate issue #110

0xean (judge) commented:

> Upgrading a contract does not mitigate that there would be an impact to the protocol, so I think this does qualify as Medium.

[M-16] Sometimes calculateBorrowerReward and calculateSupplierReward return incorrect results

Submitted by **volodya**

Sometimes `calculateBorrowerReward` and `calculateSupplierReward` return incorrect results.

Proof of Concept

Whenever a user wants to know pending rewards they call `getPendingRewards`; sometimes, it returns incorrect results.

There is a bug inside `calculateBorrowerReward` and `calculateSupplierReward`

```
        function calculateBorrowerReward(
            address vToken,
            RewardsDistributor rewardsDistributor,
            address borrower,
            RewardTokenState memory borrowState,
            Exp memory marketBorrowIndex
        ) internal view returns (uint256) {
            Double memory borrowIndex = Double({ mantissa: borrowSta
            Double memory borrowerIndex = Double({
                mantissa: rewardsDistributor.rewardTokenBorrowerInde
            });
    //      @audit
    //        if (borrowerIndex.mantissa == 0 && borrowIndex.mantiss
            if (borrowerIndex.mantissa == 0 && borrowIndex.mantissa
                // Covers the case where users borrowed tokens befor
                borrowerIndex.mantissa = rewardsDistributor.rewardTo
            }
            Double memory deltaIndex = sub_(borrowIndex, borrowerInd
            uint256 borrowerAmount = div_(VToken(vToken).borrowBalar
            uint256 borrowerDelta = mul_(borrowerAmount, deltaIndex)
            return borrowerDelta;
        }
```

[contracts/Lens/PoolLens.sol#L495]

```
        function calculateSupplierReward(
            address vToken,
            RewardsDistributor rewardsDistributor,
            address supplier,
            RewardTokenState memory supplyState
        ) internal view returns (uint256) {
            Double memory supplyIndex = Double({ mantissa: supplySta
            Double memory supplierIndex = Double({
                mantissa: rewardsDistributor.rewardTokenSupplierInde
            });
```

```
//      @audit
//        if (supplierIndex.mantissa == 0 && supplyIndex.mantiss
        if (supplierIndex.mantissa == 0 && supplyIndex.mantissa
            // Covers the case where users supplied tokens befor
            supplierIndex.mantissa = rewardsDistributor.rewardTo
        }
        Double memory deltaIndex = sub_(supplyIndex, supplierInd
        uint256 supplierTokens = VToken(vToken).balanceOf(suppli
        uint256 supplierDelta = mul_(supplierTokens, deltaIndex)
        return supplierDelta;
    }
```

contracts/Lens/PoolLens.sol#L516

Inside `rewardsDistributor` original functions are written like this:

```
    function _distributeSupplierRewardToken(address vToken, addr
...
        if (supplierIndex == 0 && supplyIndex >= rewardTokenInit
            // Covers the case where users supplied tokens befor
            // Rewards the user with REWARD TOKEN accrued from t
            // set for the market.
            supplierIndex = rewardTokenInitialIndex;
        }
...
    }
```

contracts/Rewards/RewardsDistributor.sol#L340

```
    function _distributeBorrowerRewardToken(
        address vToken,
        address borrower,
        Exp memory marketBorrowIndex
    ) internal {
...
        if (borrowerIndex == 0 && borrowIndex >= rewardTokenInit
            // Covers the case where users borrowed tokens befor
            // Rewards the user with REWARD TOKEN accrued from t
            // set for the market.
            borrowerIndex = rewardTokenInitialIndex;
        }
```

```
...
}
```

[Rewards/RewardsDistributor.sol#L374]

Recommended Mitigation Steps

```
    function calculateSupplierReward(
        address vToken,
        RewardsDistributor rewardsDistributor,
        address supplier,
        RewardTokenState memory supplyState
    ) internal view returns (uint256) {
        Double memory supplyIndex = Double({ mantissa: supplySta
        Double memory supplierIndex = Double({
            mantissa: rewardsDistributor.rewardTokenSupplierInde
        });
-         if (supplierIndex.mantissa == 0 && supplyIndex.mantissa
+         if (supplierIndex.mantissa == 0 && supplyIndex.mantissa
            // Covers the case where users supplied tokens befor
            supplierIndex.mantissa = rewardsDistributor.rewardTo
        }
        Double memory deltaIndex = sub_(supplyIndex, supplierInd
        uint256 supplierTokens = VToken(vToken).balanceOf(suppli
        uint256 supplierDelta = mul_(supplierTokens, deltaIndex)
        return supplierDelta;
    }
```

```
    function calculateBorrowerReward(
        address vToken,
        RewardsDistributor rewardsDistributor,
        address borrower,
        RewardTokenState memory borrowState,
        Exp memory marketBorrowIndex
    ) internal view returns (uint256) {
        Double memory borrowIndex = Double({ mantissa: borrowSta
        Double memory borrowerIndex = Double({
            mantissa: rewardsDistributor.rewardTokenBorrowerInde
        });
-         if (borrowerIndex.mantissa == 0 && borrowIndex.mantissa
+         if (borrowerIndex.mantissa == 0 && borrowIndex.mantissa
```

```
            // Covers the case where users borrowed tokens befor
            borrowerIndex.mantissa = rewardsDistributor.rewardTo
        }
        Double memory deltaIndex = sub_(borrowIndex, borrowerInd
        uint256 borrowerAmount = div_(VToken(vToken).borrowBalar
        uint256 borrowerDelta = mul_(borrowerAmount, deltaIndex)
        return borrowerDelta;
    }
```

Assessed type

Invalid Validation

chechu (Venus) confirmed

Low Risk and Non-Critical Issues

For this audit, 42 reports were submitted by wardens detailing low risk and non-critical issues. The **report highlighted below** by **brgltd** received the top score from the judge.

The following wardens also submitted reports: **lfzkoala**, **naman1778**, **0xSmartContract**, **ChrisTina**, **yjrwkk**, **frazerch**, **btk**, **0xkazim**, **fatherOfBlocks**, **Aymen0909**, **Infect3d**, **Cayo**, **Udsen**, **0xAce**, **0x73696d616f**, **berlin-101**, **nadin**, **lukris02**, **PNS**, **tnevler**, **koxuan**, **YoungWolves**, **IceBear**, **Team_Rocket**, **sashik_eth**, **0xWaitress**, **RaymondFam**, **0xnev**, **codeslide**, **wonjun**, **Kose**, **BGSecurity**, **Franfran**, **YakuzaKiawe**, **Bauchibred**, **kodyvim**, **volodya**, **matrix_0wl**, **bin2chen**, **Lilyjjo** *and* **Sathish9098**.

Low Issue Summary

Low	Issue
[01]	`PoolRegistry.supportMarket()` cannot be paused
[02]	Lack of revert if price returned from oracle is zero
[03]	Solidity version
[04]	State update after external calls

Low	Issue
[05]	Check for stale values on setter functions
[06]	Variable shadow
[07]	Consistent usage of require vs custom error
[08]	Avoid duplicated computation in `Comptroller.addRewardsDistributor()`
[09]	Eslint warning in a solidity file
[10]	Interchangeable usage of `msg.sender` and vToken in in `Comptroller.preBorrowCheck()`
[11]	Using underscore in a single struct field
[12]	Uncommented fields in a struct
[13]	Use return named variables or explicit returns consistently

[01] `PoolRegistry.supportMarket()` cannot be paused

Most actions in `PoolRegistry` can be paused, e.g. `_addToMarket()`, `exitMarket()`, `preMintHook()` etc.

Consider adding a "support" field in the enum Action and allowing `supportMarket()` to be paused. Not allowing `supportMarket()` to be paused could result in irregular behavior if everything else (mint, redeeming, borrowing enter market, exit market, etc) is paused but `supportMarket()` is open.

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Comptroller.sol#L801-L824

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Comptroller.sol#L1177

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Comptroller.sol#L188

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Comptroller.sol#L254

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/ComptrollerStorage.sol#L44-L54

[02] Lack of revert if price returned from oracle is zero

`RiskFund._swapAsset()` will not revert if `getUnderlyingPrice` if the price return zero.

This might be intended to avoid making the loop in `RiskFund.swapPoolAssets()` revert if a single swap is not done. However, it might be beneficial to revert the entire transaction, since price zero means the price is unavailable.

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/RiskFund/RiskFund.sol#L174

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/RiskFund/RiskFund.sol#L240-L242

https://github.com/VenusProtocol/venus-protocol/blob/e085f1194bd942c2e75de5787a0a84ec274c6dd4/contracts/Oracle/PriceOracle.sol#L13

[03] Solidity version

All contracts are using 0.8.13. Consider updating to the latest version 0.8.19 to ensure the compiler contains the latest security fixes.

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Pool/PoolRegistry.sol#L2

[04] State update after external calls

Consider making state updates prior to executing external calls to follow the checks-effects-interactions pattern.

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Shortfall/Shortfall.sol#L183

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Shortfall/Shortfall.sol#L187

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Shortfall/Shortfall.sol#L190

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Shortfall/Shortfall.sol#L193

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Shortfall/Shortfall.sol#L197-L199

[05] Check for stale values on setter functions

Add a check ensuring that the new value is different than the old value to avoid emitting unnecessary events.

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Comptroller.sol#L702-L710

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Comptroller.sol#L779-L792

[06] Variable shadow

Consider renaming the input `accessControlManager` in `PoolRegistry.initialize()`. Currently, it's being shadowed by `AccessControlledV8.accessControlManager()`. This will get a warning on common linters/text editors.

Also, consider renaming the input "owner" in `VToken.allowance()` and `VToken.balanceOf()`, since it's being shadowed by `OwnableUpgradeable.owner()`.

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Pool/PoolRegistry.sol#L170

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/VToken.sol#L539

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/VToken.sol#L548

[07] Consistent usage of require vs custom error

Consider using the same approach throughout the codebase to improve the consistency of the code.

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/VToken.sol#L396

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/VToken.sol#L489-L490

[08] Avoid duplicated computation in `Comptroller.addRewardsDistributor()`

`uint256 rewardsDistributorsLen = rewardsDistributors.length;` can be removed and the `rewardsDistributorsLength` from L930 can be reused.

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Comptroller.sol#L940

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Comptroller.sol#L930

[09] Eslint warning in a solidity file

The comment on `Comptroller.preBorrowHook()` seems to have been intended for a js/ts test file.

Consider removing it from the solidity source code or add a comment on why it needs to be there.

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Comptroller.sol#L323

[10] Interchangeable usage of `msg.sender` and `vToken` in in `Comptroller.preBorrowCheck()`

Consider replacing `_addToMarket(VToken(msg.sender), borrower)` with `_addToMarket(VToken(vToken), borrower)`, since `msg.sender` will have to be equal `vToken` for `_checkSender(vToken)` to pass. This can improve code clarity.

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Comptroller.sol#L339-L342

[11] Using underscore in a single struct field

Consider refactoring `kink_` to `kink`.

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Pool/PoolRegistry.sol#L43

[12] Uncommented fields in a struct

Consider adding comments for all the fields in a struct to improve the readability of the codebase.

Example with comments:

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/ComptrollerStorage.sol#L29-L42

Example without comments:

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/ComptrollerStorage.sol#L8-L27

[13] Use return named variables or explicit returns consistently

Some functions are declaring returned named variables on the function header, while other functions are not defining returned named variables.

The following function does not declare a returned named variable:

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Pool/PoolRegistry.sol#L379

The following function is using a return named variable in the function header:

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Pool/PoolRegistry.sol#L222

Consider adopting the same approach throughout the codebase to improve the explicitness and readability of the code.

chechu (Venus) confirmed and commented:

> 01: Confirm
> 02: Confirm
> 03: Acknowledge
> 04: Confirm
> 05: Confirm
> 06: Confirm
> 07: Confirm
> 08: Confirm
> 09: Confirm
> 10: Confirm
> 11: Confirm
> 12: Confirm
> 13: Confirm

0xean (judge) commented:

1. Non-Critical
2. Low
3. Non-Critical
4. Low

5. Non-Critical
6. Non-Critical
7. Non-Critical
8. Non-Critical
9. Non-Critical
10. Non-Critical
11. Non-Critical
12. Non-Critical
13. Non-Critical

Gas Optimizations

For this audit, 27 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **JCN** received the top score from the judge.

The following wardens also submitted reports: petrichor, 0xSmartContract, SM3_SS, naman1778, Raihan, c3phas, fatherOfBlocks, hunter_w3b, Aymen0909, pontifex, Udsen, 0xAce, Illu_23, descharre, ReyAdmirado, j4ld1na, Team_Rocket, codeslide, SAAJ, Rageur, wahedtalash77, matrix_0wl, rapha, K42, souilos *and* Sathish9098.

Summary

A majority of the optimizations were benchmarked via the protocol's tests, i.e. using the following config: `solc version 0.8.13`, `optimizer on`, and `200 runs`. Optimizations that were not benchmarked are explained via EVM gas costs and opcodes.

For full details regarding the overall average gas savings for the tested functions, with all the optimizations applied, please see the warden's original submission.

Gas Optimizations

Number	Issue	Instances
[G-01]	State variables only set in the constructor should be declared immutable	1
[G-02]	State variables can be packed to use fewer storage slots	6
[G-03]	Structs can be packed to use fewer storage slots	3
[G-04]	State variables can be cached instead of re-reading them from storage	40
[G-05]	Cache state variables outside of loop to avoid reading storage on every iteration	2
[G-06]	Avoid emitting storage values	9
[G-07]	Use calldata instead of memory for function arguments that do not get mutate	3
[G-08]	Refactor internal function to avoid unnecessary SLOAD	3
[G-09]	Return values from external calls can be cached to avoid unnecessary call	2
[G-10]	A mapping is more efficient than an array	2
[G-11]	Move storage pointer to top of function to avoid offset calculation	1
[G-12]	Move calldata pointer to top of for loop to avoid offset calculations	1
[G-13]	Using storage instead of memory for structs/arrays saves gas	1
[G-14]	Multiple accesses of a mapping/array should use a storage pointer	-
[G-15]	Use `do while` loops instead of for loops	10
[G-16]	Use assembly to perform efficient back-to-back calls	1

[G-01] State variables only set in the constructor should be declared immutable

The solidity compiler will directly embed the values of immutable variables into your contract bytecode and therefore will save you from incurring a `Gsset (20000 gas)` when you set storage variables in the constructor; a `Gcoldsload (2100 gas)` when you access storage variables for the first time in a transaction, and a `Gwarmaccess (100 gas)` for each subsequent access to that storage slot.

Total Instances: `1`

Estimated Gas Saved: `1 * 2100 = 2100`

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/BaseJumpRateModelV2.sol#L18

```
File: contracts/BaseJumpRateModelV2.sol
18:     IAccessControlManagerV8 public accessControlManager;
```diff
diff --git a/contracts/BaseJumpRateModelV2.sol b/contracts/Base
index 68b535a..bd83624 100644
--- a/contracts/BaseJumpRateModelV2.sol
+++ b/contracts/BaseJumpRateModelV2.sol
@@ -15,7 +15,7 @@ abstract contract BaseJumpRateModelV2 is Inter
      /**
       * @notice The address of the AccessControlManager contract
       */
-     IAccessControlManagerV8 public accessControlManager;
+     IAccessControlManagerV8 public immutable accessControlManag

      /**
       * @notice The approximate number of blocks per year that i
```

[G-02] State variables can be packed to use fewer storage slots

The EVM works with 32 byte words. Variables less than 32 bytes can be declared next to each other in storage and this will pack the values together into a single 32 byte storage slot (if the values combined are <= 32 bytes). If the variables packed together are retrieved together in functions we will effectively save ~2000 gas with every subsequent SLOAD for that storage slot. This is due to us incurring a Gwarmaccess (100 gas) versus a Gcoldsload (2100 gas).

There are 6 instances of this issue. (For in-depth details on this and all further gas optimizations with multiple instances, please see the warden's full report.)

[G-03] Structs can be packed to use fewer storage slots

The EVM works with 32 byte words. Variables less than 32 bytes can be declared next to each other in storage and this will pack the values together into a single 32 byte storage slot (if values combined are <= 32 bytes). If the variables packed together are retrieved together in functions (more likely with structs) we will
```

effectively save ~2000 gas with every subsequent SLOAD for that storage slot. This is due to us incurring a Gwarmaccess (100 gas) versus a Gcoldsload (2100 gas).

There are 3 instances of this issue.

[G-04] State variables can be cached instead of re-reading them from storage

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/RiskFund/ReserveHelpers.sol#L60-L66

There are 40 instances of this issue.

[G-05] Cache state variables outside of loop to avoid reading storage on every iteration

Reading from storage should always try to be avoided within loops. In the following instances, we are able to cache state variables outside of the loop to save a `Gwarmaccess (100 gas)` per loop iteration.

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Rewards/RewardsDistributor.sol#L282-L284

Gas Savings for `RewardsDistributor.claimRewardToken`, obtained via protocol's tests: Avg 77 gas.

	Min	Max	Avg	# calls
Before	262655	291169	276621	3
After	262578	291092	276544	3

```
File: contracts/Rewards/RewardsDistributor.sol
282:         for (uint256 i; i < vTokensCount; ++i) {
283:             VToken vToken = vTokens[i];
284:             require(comptroller.isMarketListed(vToken), "mar
```

```
diff --git a/contracts/Rewards/RewardsDistributor.sol b/contract
```

```
index 434732d..1b92e43 100644
--- a/contracts/Rewards/RewardsDistributor.sol
+++ b/contracts/Rewards/RewardsDistributor.sol
@@ -278,10 +278,11 @@ contract RewardsDistributor is Exponential
         uint256 vTokensCount = vTokens.length;

         _ensureMaxLoops(vTokensCount);
-
+
+        Comptroller _comptroller = comptroller;
         for (uint256 i; i < vTokensCount; ++i) {
             VToken vToken = vTokens[i];
-            require(comptroller.isMarketListed(vToken), "market
+            require(_comptroller.isMarketListed(vToken), "marke
             Exp memory borrowIndex = Exp({ mantissa: vToken.bor
             _updateRewardTokenBorrowIndex(address(vToken), borr
             _distributeBorrowerRewardToken(address(vToken), hol
```

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Comptroller.sol#L584-L586

Gas Savings for `Comptroller.healAccount`, obtained via protocol's tests: Avg 181 gas.

	Min	Max	Avg	# calls	
Before	84584	331091	223131	10	
After	84324	330963	222950	10	

Cache `oracle` outside loop to save 1 SLOAD per iteration.

Note: We must remove cached `msg.sender` variable to fix `stack too deep` error.

```
File: contracts/Comptroller.sol
584:         for (uint256 i; i < userAssetsCount; ++i) {
585:             userAssets[i].accrueInterest();
586:             oracle.updatePrice(address(userAssets[i])); // @
```

```
diff --git a/contracts/Comptroller.sol b/contracts/Comptroller.s
index 41dc518..8ae7709 100644
```

```
--- a/contracts/Comptroller.sol
+++ b/contracts/Comptroller.sol
@@ -579,12 +579,13 @@ contract Comptroller is
         VToken[] memory userAssets = accountAssets[user];
         uint256 userAssetsCount = userAssets.length;

-        address liquidator = msg.sender;
         // We need all user's markets to be fresh for the compu
+        PriceOracle _oracle = oracle;
         for (uint256 i; i < userAssetsCount; ++i) {
             userAssets[i].accrueInterest();
-            oracle.updatePrice(address(userAssets[i]));
+            _oracle.updatePrice(address(userAssets[i]));
         }
+

         AccountLiquiditySnapshot memory snapshot = _getCurrentI

@@ -616,11 +617,11 @@ contract Comptroller is

             // Seize the entire collateral
             if (tokens != 0) {
-                market.seize(liquidator, user, tokens);
+                market.seize(msg.sender, user, tokens);
             }
             // Repay a certain percentage of the borrow, forgiv
             if (borrowBalance != 0) {
-                market.healBorrow(liquidator, user, repaymentAm
+                market.healBorrow(msg.sender, user, repaymentAm
             }
         }
     }
```

[G-06] Avoid emitting storage values

Caching of a state variable replaces each `Gwarmaccess (100 gas)` with a much cheaper stack read. We can avoid unnecessary SLOADs by caching storage values that were previously accessed and emitting those cached values.

There are 9 instances of this issue.

[G-07] Use calldata instead of memory for function arguments that do not get mutated

When you specify a data location as memory, that value will be copied into memory. When you specify the location as calldata, the value will stay static within calldata. If the value is a large, complex type, using memory may result in extra memory expansion costs.

There are 3 instances of this issue.

[G-08] Refactor internal function to avoid unnecessary SLOAD

The internal functions below read storage slots that are previously read in the functions that invoke them. We can refactor the internal functions so we could pass cached storage variables as stack variables and avoid the extra storage reads, which would otherwise take place in the internal functions.

There are 3 instances of this issue.

[G-09] Return values from external calls can be cached to avoid unnecessary call

External calls are expensive as they use the `STATICCALL` / `CALL` opcode (~100 gas). If you are calling the same external function more than once you should cache the return value to avoid an unnecessary `STATICCALL` / `CALL`.

There are 2 instances of this issue.

[G-10] A mapping is more efficient than an array

Fetching data from an array is more expensive than fetching data from a mapping. Fetching data from an array will require iterating over the array until you reach your desired data. When using a mapping you only need to know the key in order to fetch the data from the exact slot it is stored in. **Source**

There are 2 instances of this issue.

[G-11] Move storage pointer to top of function to avoid offset calculation

We can avoid unnecessary offset calculations by moving the storage pointer to the top of the function.

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Comptroller.sol#L804-L810

Gas Savings for `Comptroller.supportMarket`, obtained via protocol's tests: Avg 89 gas.

	Min	Max	Avg	# calls
Before	94253	108914	103153	25
After	94175	108824	103064	25

```
File: contracts/Comptroller.sol
804:         if (markets[address(vToken)].isListed) {
805:             revert MarketAlreadyListed(address(vToken));
806:         }
807:
808:         require(vToken.isVToken(), "Comptroller: Invalid vTo
809:
810:         Market storage newMarket = markets[address(vToken)];
```

```
diff --git a/contracts/Comptroller.sol b/contracts/Comptroller.s
index 41dc518..ad41791 100644
--- a/contracts/Comptroller.sol
+++ b/contracts/Comptroller.sol
@@ -801,13 +801,13 @@ contract Comptroller is
     function supportMarket(VToken vToken) external {
         _checkSenderIs(poolRegistry);

+        Market storage newMarket = markets[address(vToken)];
         if (markets[address(vToken)].isListed) {
             revert MarketAlreadyListed(address(vToken));
         }

         require(vToken.isVToken(), "Comptroller: Invalid vToker
```

```
-         Market storage newMarket = markets[address(vToken)];
          newMarket.isListed = true;
          newMarket.collateralFactorMantissa = 0;
          newMarket.liquidationThresholdMantissa = 0;
```

[G-12] Move calldata pointer to top of for loop to avoid offset calculations

We can avoid unnecessary offset calculations by moving the calldata pointer to the top of the for loop.

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Comptroller.sol#L669-L677

Gas Savings for `Comptroller.liquidateAccount`, obtained via protocol's tests: Avg 194 gas.

	Min	Max	Avg	# calls
Before	91420	373370	233098	4
After	91198	373259	232904	4

```
File: contracts/Comptroller.sol
669:         for (uint256 i; i < ordersCount; ++i) {
670:             if (!markets[address(orders[i].vTokenBorrowed)].
671:                 revert MarketNotListed(address(orders[i].vTo
672:             }
673:             if (!markets[address(orders[i].vTokenCollateral)
674:                 revert MarketNotListed(address(orders[i].vTo
675:             }
676: 
677:             LiquidationOrder calldata order = orders[i];
```

```
diff --git a/contracts/Comptroller.sol b/contracts/Comptroller.s
index 41dc518..2c0abc9 100644
--- a/contracts/Comptroller.sol
+++ b/contracts/Comptroller.sol
@@ -667,14 +667,14 @@ contract Comptroller is
         _ensureMaxLoops(ordersCount);
```

```
            for (uint256 i; i < ordersCount; ++i) {
-               if (!markets[address(orders[i].vTokenBorrowed)].isI
-                   revert MarketNotListed(address(orders[i].vToker
+               LiquidationOrder calldata order = orders[i];
+               if (!markets[address(order.vTokenBorrowed)].isListe
+                   revert MarketNotListed(address(order.vTokenBorr
                }
-               if (!markets[address(orders[i].vTokenCollateral)].i
-                   revert MarketNotListed(address(orders[i].vToker
+               if (!markets[address(order.vTokenCollateral)].isLis
+                   revert MarketNotListed(address(order.vTokenCol
                }

-               LiquidationOrder calldata order = orders[i];
                order.vTokenBorrowed.forceLiquidateBorrow(
                    msg.sender,
                    borrower,
```

[G-13] Using storage instead of memory for structs/arrays saves gas

Using a memory pointer for a storage struct/array will effectively load all the fields of that data type from storage (SLOAD) into memory (MSTORE). Using a storage pointer will allow you to read specific fields from storage as you need them. If you are not going to use all of the fields of your data type then you should use a storage pointer so that you don't incur extra `Gcoldsload (2100 gas)` for fields that you will never use.

Note: These are instances that the automated report missed.

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Pool/PoolRegistry.sol#L394

Gas Savings for `PoolRegistry.createRegistryPool` , obtained via protocol's tests: Avg 853 gas.

	Min	Max	Avg	# calls
Before	645562	683114	664419	23

	Min	Max	Avg	# calls
After	644720	682260	663566	23

```
File: contracts/Pool/PoolRegistry.sol
394:         VenusPool memory venusPool = _poolByComptroller[comp
```

```
diff --git a/contracts/Pool/PoolRegistry.sol b/contracts/Pool/Pc
index 5cf376f..7d7c2e6 100644
--- a/contracts/Pool/PoolRegistry.sol
+++ b/contracts/Pool/PoolRegistry.sol
@@ -391,7 +391,7 @@ contract PoolRegistry is Ownable2StepUpgrade
      * @return The index of the registered Venus pool
      */
     function _registerPool(string calldata name, address comptr
-        VenusPool memory venusPool = _poolByComptroller[comptrc
+        VenusPool storage venusPool = _poolByComptroller[comptr

         require(venusPool.creator == address(0), "PoolRegistry:
         _ensureValidName(name);
```

[G-14] Multiple accesses of a mapping/array should use a storage pointer

Caching a mapping's value in a storage pointer when the value is accessed multiple times saves ~40 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 achieve this, declare a storage pointer for the variable and use it instead of repeatedly fetching the reference in a map or an array. As an example, instead of repeatedly calling `stakes[tokenId_]`, save its reference via a storage pointer: `StakeInfo storage stakeInfo = stakes[tokenId_]` and use the pointer instead.

For all instances and in-depth details of this issue, please see the warden's **full report**.

[G-15] Use `do while loops` instead of for loops

A `do while` loop will cost less gas since the condition is not being checked for the first iteration.

There are 10 instances of this issue.

[G-16] Use assembly to perform efficient back-to-back calls

If a similar external call is performed back-to-back, we can use assembly to reuse any function signatures and function parameters that stay the same. In addition, we can also reuse the same memory space for each function call (`scratch space` + `free memory pointer` + `zero slot`), which can potentially allow us to avoid memory expansion costs.

Note: In order to do this optimization safely we will cache the free memory pointer value and restore it once we are done with our function calls. We will also set the zero slot back to 0 if necessary.

https://github.com/code-423n4/2023-05-venus/blob/main/contracts/Pool/PoolRegistry.sol#L239-L245

Gas Savings for `PoolRegistry.createRegistryPool` , obtained via protocol's tests: Avg 1049 gas.

	Min	Max	Avg	# calls	
Before	645562	683114	664419	23	
After	644525	682065	663370	23	

```
File: contracts/Pool/PoolRegistry.sol
239:         comptrollerProxy.setCloseFactor(closeFactor);
240:         comptrollerProxy.setLiquidationIncentive(liquidatior
241:         comptrollerProxy.setMinLiquidatableCollateral(minLic
242:         comptrollerProxy.setPriceOracle(PriceOracle(priceOra
243:
244:         // Start transferring ownership to msg.sender
245:         comptrollerProxy.transferOwnership(msg.sender);
```

```
diff --git a/contracts/Pool/PoolRegistry.sol b/contracts/Pool/Pc
index 5cf376f..b2d696a 100644
--- a/contracts/Pool/PoolRegistry.sol
+++ b/contracts/Pool/PoolRegistry.sol
@@ -236,13 +236,28 @@ contract PoolRegistry is Ownable2StepUpgra
         uint256 poolId = _registerPool(name, proxyAddress);

         // Set Venus pool parameters
-        comptrollerProxy.setCloseFactor(closeFactor);
-        comptrollerProxy.setLiquidationIncentive(liquidationInc
-        comptrollerProxy.setMinLiquidatableCollateral(minLiquic
-        comptrollerProxy.setPriceOracle(PriceOracle(priceOracle
-
-        // Start transferring ownership to msg.sender
-        comptrollerProxy.transferOwnership(msg.sender);
+        assembly {
+            // function signature for setCloseFactor(uint256)
+            mstore(0x00, 0x12348e96)
+            mstore(0x20, calldataload(0x44))
+            if iszero(call(gas(), comptrollerProxy, 0x00, 0x1c,
+            // function signature for setLiquidationIncentive(u
+            mstore(0x00, 0xa8431081)
+            mstore(0x20, calldataload(0x64))
+            if iszero(call(gas(), comptrollerProxy, 0x00, 0x1c,
+            // function signature for setMinLiquidatableCollate
+            mstore(0x00, 0x520b6c74)
+            mstore(0x20, calldataload(0x84))
+            if iszero(call(gas(), comptrollerProxy, 0x00, 0x1c,
+            // function signature for setPriceOracle(address)
+            mstore(0x00, 0x530e784f)
+            mstore(0x20, calldataload(0xa4))
+            if iszero(call(gas(), comptrollerProxy, 0x00, 0x1c,
+            // function signature for transferOwnership(address
+            mstore(0x00, 0xf2fde38b)
+            mstore(0x20, caller())
+            if iszero(call(gas(), comptrollerProxy, 0x00, 0x1c,
+        }

         // Register the pool with this PoolRegistry
         return (poolId, proxyAddress);
```

GasReport output with all optimizations applied

Note: please see warden's original submission for full details.

chechu (Venus) confirmed and commented:

> G-01 Confirm
> G-02 TBD
> G-03 TBD
> G-04 Confirm
> G-05 Confirm
> G-06 Confirm
> G-07 Confirm
> G-08 Confirm
> G-09 Confirm
> G-10 Disagree with severity
> G-11 Disagree with severity
> G-12 Disagree with severity
> G-13 Confirm
> G-14 TBR
> G-15 Disagree with severity
> G-16 Disagree with severity

Disclosures

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

C4 Audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit 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