

Venus Leverage Strategies Manager Audit

Please Note

1. The analysis of the Severity is purely based on the smart contracts mentioned in the Audit Scope and does not include any other potential contracts deployed by the Owner. No applications or operations were reviewed for Severity. No product code has been reviewed.
2. Due to the time limit, the audit team did not do much in-depth research on the business logic of the project. It is more about discovering issues in the smart contracts themselves.

Audit Period: 2025/11/24 - 2025/12/11 (YYYY/MM/DD)

Overall Risk: Medium

Project Name	Venus Leverage Strategies Manager Audit
Github	https://github.com/VenusProtocol/venus-periphery/pull/12
Commit	6a4ba3e01b38ddc96673490c6b407b8d28769bb9

Smart contracts list:

No.	Contract Name	Link	Verdict	Details
1	contracts/LeverageManager/ILeverageStrategiesManager.sol	https://github.com/VenusProtocol/venus-periphery/blob/feat/VPD-70/contracts/LeverageManager/ILeverageStrategiesManager.sol	Green	
2	contracts/LeverageManager/LeverageStrategiesManager.sol	https://github.com/VenusProtocol/venus-periphery/blob/feat/VPD-70/contracts/LeverageManager/LeverageStrategiesManager.sol	Medium	[M01] [M02] [M03] [L01] [L02] [I01]

Findings

[M01] Missing interest accrual in account safety checks

Contract LeverageStrategiesManager.sol

Severity Level Medium

Description In all 3 market actions involving flash borrowing actions of leverage entrance, the `_checkAccountSafe` is first invoked to ensure that the user does not have any short fall i.e. liquidity after accounting for collateralization ratio minus borrowed amount must not be greater than zero

JavaScript

```
function _checkAccountSafe(address user) internal view {
    (uint256 err, , uint256 shortfall) =
    COMPTROLLER.getBorrowingPower(user);
    if (err != SUCCESS || shortfall > 0) revert
    OperationCausesLiquidation();
}
```

This calls towards the `Comptroller.getBorrowingPower` (`PolicyFacet`), which in turns utilizes the internal `_getAccountLiquidity`

JavaScript

```
function getBorrowingPower(address account) external view returns
(uint256, uint256, uint256) {
    return _getAccountLiquidity(account,
WeightFunction.USE_COLLATERAL_FACTOR);
}
```

If you follow the full call flow below

LeverageStrategiesManager.`_checkAccountSafe` → PolicyFacet.`getBorrowingPower`
→ FacetBase.`_getAccountLiquidity` →
FacetBase.`getHypotheticalAccountLiquidityInternal` →
ComptrollerLens.`getHypotheticalAccountLiquidity` →
ComptrollerLens.`_calculateAccountPosition`

We notice that in all of these function calls, the `VToken.accrueInterest` function is never called. Then, within the `ComptrollerLens._calculateAccountPosition` function, the `VToken.getAccountSnapshot` is called to retrieve a snapshot of the users vToken balance (collateral), borrowed balance and current vToken:underlying token exchange rate, which is subsequently used to compute the users account position values

```

JavaScript
function _calculateAccountPosition(
    address comptroller,
    address account,
    VToken vTokenModify,
    uint redeemTokens,
    uint borrowAmount,
    WeightFunction weightingStrategy
) internal view returns (uint oErr, AccountLiquidityLocalVars
memory vars) {
    // For each asset the account is in
    VToken[] memory assets =
    ComptrollerInterface(comptroller).getAssetsIn(account);
    uint assetsCount = assets.length;
    for (uint i = 0; i < assetsCount; ++i) {
        VToken asset = assets[i];

        // Read the balances and exchange rate from the vToken
        (oErr, vars.vTokenBalance, vars.borrowBalance,
        vars.exchangeRateMantissa) = asset.getAccountSnapshot(
            account
        );
}

```

In the VToken.getAccountSnapshot, notice how the stored internal borrow balance and exchange rate balances are retrieved directly with the borrowBalanceStoredInternal and exchangeRateStoredInternal that does not utilize the latest global borrow state, e.g. borrowIndex, totalBorrows

```

JavaScript
function getAccountSnapshot(address account) external view override
returns (uint, uint, uint, uint) {
    uint borrowBalance;
    uint exchangeRateMantissa;

    MathError mErr;

    (mErr, borrowBalance) = borrowBalanceStoredInternal(account);
    if (mErr != MathError.NO_ERROR) {
        return (uint(Error.MATH_ERROR), 0, 0, 0);
    }

    (mErr, exchangeRateMantissa) = exchangeRateStoredInternal();
    if (mErr != MathError.NO_ERROR) {
        return (uint(Error.MATH_ERROR), 0, 0, 0);
    }
}

```

```

        return (uint(Error.NO_ERROR), accountTokens[account],
borrowBalance, exchangeRateMantissa);
    }
}

```

As such, any user with pending interest from existing positions in their respective VToken markets can possibly bypass safety checks. While the subsequent executeFlashLoan will accrue interest for the involved collateral/borrow markets, the other entered VToken markets will still not have interest accrued, so the second _checkAccountSafe still does not account for this, resulting in a secondary bypass.

Recommendation	Consider looping through the users entered VToken markets via getAssetsIn (MarketFacet) and accrue interest for all markets before executing the flash loan action.
Status	Acknowledged, partially fixed, interest is now accrued for the involved collateralMarket and/or borrowMarket (depending on leverage actions). This design is consistent with Venus Core market actions (in which only the markets involve will have interest accrued) to avoid excessive consumption of gas incase the user has entered a large number of markets.

[M02] Swap Input Amount Mismatch in _handleExitCollateral could cause leverage exit reverts due to non-zero treasury fee

Contract	LeverageStrategiesManager.sol
Severity Level	Medium
Description	In _handleExitCollateral, notice that code logic does not verify the actual collateral amount received after redeemUnderlyingBehalf and directly uses the requested amount for swap represented by collateralAmountToRedeem
<pre> JavaScript function _handleExitCollateral(address onBehalf, IVToken borrowMarket, uint256 borrowedAssetAmountToRepayFromFlashLoan, uint256 borrowedAssetFees, bytes calldata swapCallData) internal returns (uint256 flashLoanRepayAmount) { //... err = _collateralMarket.redeemUnderlyingBehalf(onBehalf, collateralAmountToRedeem); if (err != SUCCESS) { revert RedeemBehalfFailed(err); } } </pre>	

```

IERC20Upgradeable collateralAsset =
IERC20Upgradeable(_collateralMarket.underlying());
uint256 swappedBorrowedAmountOut =
_performSwap(collateralAsset, collateralAmountToRedeem, borrowedAsset,
_minAmountOutAfterSwap, swapCallData);

//...
}

```

While the redeemAmount input generally represents the exact number of underlying assets to receive from redemption of the VToken, there is an edge case where collateralAmountToRedeem will not be the exact amount received by the LSM contract to perform the swap

The edge case occurs due to a non-zero treasury protocol fee percentage, which can be seen in the redeemFresh function, where we can see below that a portion of the fee is transferred to the treasuryAddress, with only the remainedAmount that has subtracted the treasury fee being transferred to the LSM contract to perform the swap

```

JavaScript
uint feeAmount;
uint remainedAmount;
if (IComptroller(address(comptroller)).treasuryPercent() != 0)
{
    (vars.mathErr, feeAmount) = mulUInt(
        vars.redeemAmount,
        IComptroller(address(comptroller)).treasuryPercent()
    );
    ensureNoMathError(vars.mathErr);

    (vars.mathErr, feeAmount) = divUInt(feeAmount,
MANTISSA_ONE);
    ensureNoMathError(vars.mathErr);

    (vars.mathErr, remainedAmount) =
subUInt(vars.redeemAmount, feeAmount);
    ensureNoMathError(vars.mathErr);

    address payable treasuryAddress =
payable(IComptroller(address(comptroller)).treasuryAddress());
    doTransferOut(treasuryAddress, feeAmount);

    emit RedeemFee(redeemer, feeAmount, vars.redeemTokens);
} else {
    remainedAmount = vars.redeemAmount;
}

```

```
doTransferOut(receiver, remainedAmount);
```

As such, since the LSM contract do not have exactly sufficient funds to perform the swap, the exitLeverage function will always revert for a vToken market with a non-zero treasury fee, unless a user had previously performed a batch call to transfer tokens directly into the LSM contract to cover this treasury fee before executing the exitLeverage function

Note: This also affects the _handleExitSingleAsset internal function and causes it to revert due to the following check, since the balance of the underlying collateral asset redeemed will be equal to flashLoanRepayAmount - treasury fees.

JavaScript

```
flashLoanRepayAmount = flashloanedCollateralAmount +  
collateralAmountFees;  
  
err = market.redeemUnderlyingBehalf(onBehalf,  
flashLoanRepayAmount);  
if (err != SUCCESS) {  
    revert RedeemBehalfFailed(err);  
}  
  
if (collateralAsset.balanceOf(address(this)) <  
flashLoanRepayAmount) {  
    revert InsufficientFundsToRepayFlashloan();  
}
```

Recommendation	Fix the swap input mismatch in _handleExitCollateral by verifying the actual collateral received after redeemUnderlyingBehalf instead of using the requested amount. Record the collateral balance before redemption by computing the actual received amount as the post-redemption difference, and use that value for the swap call.
Status	Fixed, now precomputes the required redemption amount required to cover treasury protocol fees. This is performed in a conservative way by invoking a ceiling division in the numerator to ensure that the redeemed amount is never lesser than redeemAmount. Any subsequent excess will be sent back to the user via _transferDustToInitiator appropriately

[M03] Incorrect dust asset handling logic for EXIT_COLLATERAL collateral exit for excess input tokens

Contract LeverageStrategiesManager

Severity Level	Medium
Description	In the <code>_handleExitCollateral</code> function, if <code>swappedBorrowedAmountOut</code> exceeds <code>flashLoanRepayAmount</code> , the excess amount will be taken by the protocol treasury via the <code>_transferDustToTreasury</code> and the end of the <code>exitLeverage</code> function call. This can occur when there the price favours the collateral token in the underlying swap protocols that results in excess borrowed tokens from the swap after repayment
JavaScript	<pre>function _handleExitCollateral(address onBehalf, IVToken borrowMarket, uint256 borrowedAssetAmountToRepayFromFlashLoan, uint256 borrowedAssetFees, bytes calldata swapCallData) internal returns (uint256 flashLoanRepayAmount) { IERC20Upgradeable borrowedAsset = IERC20Upgradeable(borrowMarket.underlying()); uint256 borrowedTotalDebtAmount = borrowMarket.borrowBalanceCurrent(onBehalf); uint256 repayAmount = borrowedAssetAmountToRepayFromFlashLoan > borrowedTotalDebtAmount ? borrowedTotalDebtAmount : borrowedAssetAmountToRepayFromFlashLoan; borrowedAsset.forceApprove(address(borrowMarket), repayAmount); uint256 err = borrowMarket.repayBorrowBehalf(onBehalf, repayAmount); if (err != SUCCESS) { revert RepayBehalfFailed(err); } // Cache transient storage reads for variables used more than once to save gas IVToken _collateralMarket = collateralMarket; uint256 collateralAmountToRedeem = collateralAmount; err = _collateralMarket.redeemUnderlyingBehalf(onBehalf, collateralAmountToRedeem); if (err != SUCCESS) { revert RedeemBehalfFailed(err); } IERC20Upgradeable collateralAsset = IERC20Upgradeable(_collateralMarket.underlying()); uint256 swappedBorrowedAmountOut = _performSwap(collateralAsset, collateralAmountToRedeem, borrowedAsset, minAmountOutAfterSwap, swapCallData); flashLoanRepayAmount = borrowedAssetAmountToRepayFromFlashLoan + borrowedAssetFees;</pre>

```

    if (swappedBorrowedAmountOut < flashLoanRepayAmount) {
        revert InsufficientFundsToRepayFlashloan();
    }

    borrowedAsset.forceApprove(address(borrowMarket),
    flashLoanRepayAmount);
}

```

As such, excess input collateral tokens that are unused are not rightfully returned to the user, but instead transferred to the protocol treasury instead.

Recommendation	Consider differentiating between genuine "dust" (small residual amounts) and excess "surplus" from unused input redeemed collateral tokens. If swappedBorrowedAmountOut > flashLoanRepayAmount, any excess amount should be returned back to the user.
Status	Fixed, excess output borrowed token amount will now be transferred to the flashloan initiator instead of the treasury address

[L01] Incorrect _checkAccountSafe order can cause leverage action reverts

Contract	LeverageStrategiesManager.sol
Severity Level	Low
Description	In all 3 market actions involving flash borrowing actions of leverage entrance, the _checkAccountSafe is first invoked to ensure that the user does not have any short fall i.e. liquidity after accounting for collateralization ratio minus borrowed amount must not be greater than zero. This check is always performed before the _validateAndEnterMarket, which will enter a market for a user if not previously done so

```

JavaScript
_checkAccountSafe(msg.sender);

_validateAndEnterMarket(msg.sender, _collateralMarket);

```

In _validateAndEnterMarket, the LSM contract will enter the specific collateral/borrow market (depending on leverage action) for the user if not previously done so

```

JavaScript
function _validateAndEnterMarket(address user, IVToken market)
internal {
    if (!COMPTROLLER.checkMembership(user, market)) {
        uint256 err = COMPTROLLER.enterMarketBehalf(user,
address(market));
        if (err != SUCCESS) revert EnterMarketFailed(err);
    }
}

```

Normally, there are no issues with checks of such an order since when a user performs a market exit, all VToken balances must have been redeemed to underlying assets.

However, consider a scenario where a user has a healthy account (no shortfall) and holds a certain amount of vToken transferred to them, which is not obtained from collateral supply but via direct transfers, while simultaneously, not having previously entered this specific vToken market.

Since `_checkAccountSafe` is invoked first, if the users collateral position is computed with a shortfall before entering this market, they will be blocked from performing this leverage position. The caveat here is that liquidation could have occurred first before a leverage action is invoked.

Recommendation	Consider switching the position of the <code>_checkAccountSafe</code> and <code>_validateAndEnterMarket</code> checks
Status	Fixed, in <code>enterSingleAssetLeverage/enterLeverage/enterLeverageFromBorrow</code> , the <code>_validateAndEnterMarket</code> is invoked before <code>_checkAccountSafe</code>

[L02] Special Cases May Prevent Users from Exiting Leverage

Contract	LeverageStrategiesManager
Severity Level	Low
Description	In the redemption logic of the <code>_handleExitSingleAsset</code> function, the amount to redeem is based on the amount of flash loaned collateral and the associated fees. However, since Venus shares assets globally, and according to the logic of this contract, if a user sets <code>_collateralAmountSeed</code> to 0 when entering the leverage position, their staked collateral will be the same as the borrowed amount. In this case, the redemption amount will be <code>flashloanedCollateralAmount + collateralAmountFees</code> , which may exceed the user's actual staked collateral. As a result,

`redeemUnderlyingBehalf` could fail, potentially preventing the user from exiting their leveraged position.

JavaScript

```
IERC20Upgradeable collateralAsset =
IERC20Upgradeable(market.underlying());

collateralAsset.forceApprove(address(market),
flashloanedCollateralAmount);
uint256 err = market.repayBorrowBehalf(onBehalf,
flashloanedCollateralAmount);

if (err != SUCCESS) {
    revert RepayBehalfFailed(err);
}

flashLoanRepayAmount = flashloanedCollateralAmount +
collateralAmountFees; // The total amount to redeem may exceed the
actual collateral amount.
err = market.redemUnderlyingBehalf(onBehalf,
flashLoanRepayAmount);
```

To illustrate the issue more clearly, consider the following example:

A user has already supplied assets worth 200U in the vBNB market. Then, the user opens a 100U leveraged position in the vUSDT market with a seed amount of 0. After leverage is executed, the resulting state becomes: 100 USDT supplied and 102U borrowed (including 2U flash-loan fee). At this point, if the user attempts to fully exit the leverage position, a flash loan of 102U is required. However, during the exit process, the redeem amount will reach 104U (assuming another 2U flash-loan fee is charged during exit, i.e., 102U + 2U in total). Since the redeem amount exceeds what is allowable, the transaction will revert.

Recommendation

- Consider calculating the maximum redeemable amount first.
- Alternatively, modify the logic to only redeem the necessary amount to repay the flash loan.

Status

Fixed, now limits the redemption amount to the current user underlying collateral balance converted from their vToken balance with the latest exchange rates by invoking `VToken.balanceOfUnderlying`. This means that single asset exit leverage actions cannot be invoked unless the full flash borrow amount + flash borrow fees is repaid, if no the `InsufficientFundsToRepayFlashloan` error will be invoked

[I01] Missing Validation When Collateral and Borrow Markets Are Identical

Contract

LeverageStrategiesManager

Severity Level	Informational
Description	The intended purpose of the enterLeverage, enterLeverageFromBorrow, and exitLeverage functions is to allow users to supply and borrow two different assets, and internally perform swaps using flash loans to fulfill collateral and debt requirements. However, these functions currently lack validation to ensure that <code>_collateralMarket</code> and <code>_borrowedMarket</code> are different. If both parameters reference the same market, subsequent swap operations will fail, as most DEXes do not support swapping an asset to itself.
Recommendation	Add a validation check in the entry functions and revert when <code>_collateralMarket</code> and <code>_borrowedMarket</code> refer to the same market.
Status	Fixed as recommended, now includes check within <code>enterLeverage/enterLeverageFromBorrow/exitLeverage</code> to ensure that <code>_collateralMarket != _borrowedMarket</code> , if not it will revert.