SHERLOCK

Security Review For HaHa Technologies

Collaborative Audit Prepared For: **HaHa Technologies**
Lead Security Expert(s): **0xadrii**
PeterSR
T1MOH
Date Audited: **April 6 - April 20, 2026**

Introduction

HaHa Technologies Inc (Permutize) is a developer of numerous Web3 products. This vault smart contract is for a delta-neutral stablecoin vault. It deploys to lending protocols and auto rebalances to achieve better than average market returns. Can support both USDC and AUSDC on Monad (EVM).

Scope

Repository: Permutize/smart-vault-contracts

Audited Commit: 63b7acada71bfef9b8cffb6229e9ab593ed7306e

Final Commit: e2b1b012b2b63ac98a64807da7b5e2ffe7d11834

Files:

- src/adapters/AdapterHarness.sol
- src/adapters/curvance/CurvanceCollateralAdapter.sol
- src/adapters/curvance/CurvanceDebtAdapter.sol
- src/adapters/euler/EulerCollateralAdapter.sol
- src/adapters/euler/EulerDebtAdapter.sol
- src/adapters/merkl/MerklAdapter.sol
- src/adapters/rewards/RewardsAdapter.sol
- src/adapters/rewards/Swapper.sol
- src/common/SwapAdapter.sol
- src/fee/VaultFeeAllocator.sol
- src/GatedVaultImpl.sol
- src/libraries/LTVCalculations.sol
- src/libraries/OracleHelpers.sol
- src/libraries/StrategyCommandSelectors.sol
- src/libraries/StrategyRolesLib.sol
- src/libraries/StrategyStorageLib.sol
- src/oracle/CompositeOracle.sol
- src/strategies/AusdStrategy.sol
- src/strategies/AusdStrategyUnstakeWorker.sol
- src/strategies/BaseStrategy.sol

Final Commit Hash

e2b1b012b2b63ac98a64807da7b5e2ffe7d11834

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
3	7	14

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue H-1: Double Sequential Division Can Round to Zero in `OracleHelpers` [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/issues/23

Summary

`OracleHelpers.convertToUsd` divides twice sequentially. For non-standard oracle decimal configurations, the intermediate result of the first division can be smaller than the second divisor, rounding the final result to zero. A zero return value bypasses the `MIN_LOAN_SIZE` guard in `CurvanceDebtAdapter`, silently allowing sub-minimum borrows.

Vulnerability Detail

```
// OracleHelpers.sol:L60
return (amount * price * USD_DECIMALS) / (DECIMAL_BASE ** tokenDecimals) /
 ↪  (DECIMAL_BASE ** priceDecimals);
```

With `tokenDecimals=18` and `priceDecimals=18`, the intermediate value after the first division can be less than `1e18`, causing the second division to truncate to zero. `finalBorrowUsd == 0` then causes the `MIN_LOAN_SIZE` check to pass unconditionally.

Current deployment (18-decimal tokens, 8-decimal Chainlink feeds) is not affected. The risk materializes if future oracle integrations use higher decimal counts.

Impact

Under non-standard oracle configurations, `MIN_LOAN_SIZE` protection is silently bypassed, allowing the strategy to enter borrow positions that Curvance would normally reject. For current deployment configuration, no practical impact.

Code Snippet

`OracleHelpers.sol:L60`

Tool Used

Manual Review

Recommendation

Restructure to a single combined division:

```
return (amount * price * USD_DECIMALS) /
       ((DECIMAL_BASE ** tokenDecimals) * (DECIMAL_BASE ** priceDecimals));
```

Discussion

PeterSRWeb3

Not a real problem

themuli

Reopening since this is a high impact issue. See Mike's PR.

Issue H-2: Wrong Released Assets Invariant Check Will Lead To DoS In Healthy Vault And Silent Failure Otherwise [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/issues/28

Summary

The `withdraw` function in `EulerCollateralAdapter` incorrectly interprets the return value from Euler's vault, treating shares burned as assets withdrawn. This breaks the withdrawal invariant check, causing spurious reverts on legitimate withdrawals or silently allowing shortfalls.

Vulnerability Detail

The ERC4626 standard specifies that `withdraw(uint256 assets, address receiver, address owner)` returns the number of **shares burned**, not the number of underlying assets released. However, the code comment suggests the developer expected Euler to return assets:

```
// Euler withdraw returns the amount of underlying assets released from the vault
↪  rather
// than the number of shares burned, so compare the returned assets directly.
uint256 assets = COLLATERAL_VAULT.withdraw(amount, address(this), address(this));
if (assets < amount) revert CollateralWithdrawalFailed(amount, assets);
```

In reality, `assets` holds the number of shares burned. When the vault's share price exceeds 1:1 (appreciating collateral):

- To withdraw 100 assets at share price 1.2 requires burning ~83 shares
- The check `if (83 < 100)` triggers spuriously, reverting a valid withdrawal

When the vault's share price falls below 1.0 (vault losses):

- The check may pass silently despite returning fewer assets than requested
- The invariant that protects against shortfalls is broken in both directions

`EVault`'s relevant code can be found here.

Impact

The vulnerability degrades withdrawal reliability and user experience:

1. **Denial of service on normal withdrawals**: Users cannot withdraw collateral once the vault's share price appreciates above 1:1, which is the expected state for a healthy vault with earned rewards or value growth.
2. **Silent shortfalls in adverse conditions**: If vault losses cause share price to drop below 1.0, users can withdraw less than requested without reverting, leading to silent fund loss.

Code Snippet

```
uint256 assets = COLLATERAL_VAULT.withdraw(amount, address(this), address(this));
if (assets < amount) revert CollateralWithdrawalFailed(amount, assets);
```

Tool Used

Manual Review

Recommendation

Adjust the invariant check accordingly:

```
uint256 sharesBurned = COLLATERAL_VAULT.withdraw(amount, address(this),
↪  address(this));
uint256 assetsReleased = COLLATERAL_VAULT.convertToAssets(sharesBurned);
if (assetsReleased < amount) revert CollateralWithdrawalFailed(amount,
↪  assetsReleased);
```

Discussion

themuli

Acknowledged - @0xdavid7

Issue H-3: Unstake requests removed forcefully break vault accounting [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/issues/50

Summary

`AusdStrategy_calculateTotalCollateralUsd` does not account for value held inside force-removed `AusdStrategyUnstakeWorker` contracts. Because Concrete's `_previewStrategyYield` calls `IStrategyTemplate(strategy).totalAllocatedValue()`, force-removing a stuck worker silently drops its locked MON value out of NAV, leading to a potential loss accounted.

Vulnerability Detail

When requesting a unstake via `_requestUnstakeShMon`, a fresh worker is deployed and shMON is transferred to it, incrementing `totalPendingUnstakeMon` by the shMON amount. Then, `_calculateTotalCollateralUsd` values that pending amount through `totalPendingUnstakeMon`:

```
function _calculateTotalCollateralUsd() private view returns (uint256 total) {
        ...

        uint256 pendingMon = AusdStrategyStorageLib.fetch().totalPendingUnstakeMon;
        ↪  <@
        if (pendingMon > 0) {
            total += _wmonToUsd(pendingMon);
        }
```

The problem is that an edge case could occur where `forceRemoveUnstakeRequest` is used to remove an unstake request, decrementing `totalPendingUnstakeMon`, while the worker still holds the shMON shares. This won't be accounted in `_calculateTotalCollateralUsd`. Concrete's `_previewStrategyYield`compares `currentTotalAllocatedValue` to the recorded `strategyData[strategy].allocated`. After force-removal `currentTotalAllocatedValue` drops by the full expected MON value, so Concrete records a `loss` equal to that amount in `_previewYieldNoFees`.

Impact

High. Vault's accounting will break during a period of time, until `recoverForceRemovedUnstakeRequest` is called. Users redeeming during the gap between `forceRemoveUnstakeRequest` and `recoverForceRemovedUnstakeRequest` are diluted at incorrect NAV, due to accounting being broken.

Code Snippet

https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/blob/b840577fc767f161c09450435c695c94294125ef/smart-vault-contracts/src/strategies/AusdStrategy.sol#L667

Tool Used

Manual Review

Recommendation

Keep accounting active for force-removed workers until `recoverForceRemovedUnstakeRequest` is triggered.

Discussion

haha-mike

Acknowledged. @0xdavid7 is working on this.

Issue M-1: `_terminatePositions` Withdraws Euler Collateral Before Repaying Euler Debt [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/issues/29

Summary

`AusdStrategy._terminatePositions()` withdraws the Euler shMON collateral before repaying the Euler AUSD debt. Because the Euler debt vault is registered as a controller for the account, every collateral withdrawal triggers an EVC account health check. At the moment of withdrawal the AUSD debt is still outstanding and the collateral is being removed, leaving the account with zero collateral against non-zero debt. The health check reverts unconditionally, permanently blocking the standard termination path for any strategy instance with an active Euler position.

Vulnerability Detail

`_terminatePositions()` unwinds positions in the following order:

1. **L847–852**: withdraw full shMON balance from `SECONDARY_COLLATERAL_ADAPTER` (Euler)
2. L854–858: unstake shMON → MON, wrap to WMON
3. L860–870: repay Curvance WMON primary debt
4. L872–876: early return to respect Curvance cooldown
5. L878–884: withdraw AUSD from Curvance collateral
6. **L886–892**: repay Euler AUSD secondary debt ← too late

When step 1 executes, `EulerCollateralAdapter.withdraw()` calls:

```
// EulerCollateralAdapter.sol:144
uint256 assets = COLLATERAL_VAULT.withdraw(amount, address(this), address(this));
```

The Euler EVault's `withdraw()` calls `requireAccountAndVaultStatusCheck` at the end of execution. The EVC then asks the enabled controller – the AUSD debt vault – to verify account solvency. At this point:

- AUSD debt: still outstanding (step 6 hasn't run)
- shMON collateral: just removed to zero
- Account LTV: debt / 0 = infinite → **revert**

The controller was enabled during the initial deposit:

```
// EulerCollateralAdapter.sol:119-127
if (!EVC.isControllerEnabled(address(this), address(DEBT_VAULT))) {
```

```
    EVC.enableController(address(this), address(DEBT_VAULT));
}
if (!EVC.isCollateralEnabled(address(this), address(COLLATERAL_VAULT))) {
    EVC.enableCollateral(address(this), address(COLLATERAL_VAULT));
}
```

Once a controller is enabled, it validates account health on every collateral state change. There is no `EVC.batch()` wrapper around `_terminatePositions` that would defer these checks. The function reverts at step 1 and never reaches step 6.

Note that the Curvance leg applies the correct ordering – it repays WMON debt (L860) before withdrawing AUSD collateral (L878). The Euler leg inverts this order.

Impact

Any strategy instance with an active Euler secondary position (shMON collateral deposited, AUSD debt outstanding) cannot be terminated through the standard `_terminatePositions` flow. Vault deallocation calls that trigger termination will revert. The position is not lost – both collateral and debt remain intact – but the strategy is stuck until an admin manually sequences the correct operations outside of the designed code path.

Code Snippet

```
// AusdStrategy.sol:847-892

// Step 1: withdraw Euler collateral - health check fires here, AUSD debt still
↪  outstanding → REVERT
uint256 shMonCollateral =
↪  _getCollateralBalance(address(SECONDARY_COLLATERAL_ADAPTER()), address(SHMON));
if (shMonCollateral > 0) {
    bytes memory withdrawData =
        abi.encodeWithSelector(ICollateralAdapter.withdraw.selector,
        ↪  address(SHMON), shMonCollateral);
    _delegateCallVoid(address(SECONDARY_COLLATERAL_ADAPTER()), withdrawData);
}

// ... (unstake, repay Curvance, withdraw Curvance collateral) ...

// Step 6: repay Euler AUSD debt - never reached
uint256 stablecoinDebt = _getDebtBalance(address(SECONDARY_DEBT_ADAPTER()),
↪  address(STABLECOIN));
uint256 stablecoinBalance = STABLECOIN.balanceOf(address(this));
if (stablecoinDebt > 0 && stablecoinBalance > 0) {
    uint256 toRepay = stablecoinBalance < stablecoinDebt ? stablecoinBalance :
    ↪  stablecoinDebt;
```

```
    bytes memory repayData = abi.encodeWithSelector(IDebtAdapter.repay.selector,
    ↪  address(STABLECOIN), toRepay);
    _delegateCallVoid(address(SECONDARY_DEBT_ADAPTER()), repayData);
}
```

Tool Used

Manual Review

Recommendation

Repay the Euler AUSD debt before withdrawing the Euler shMON collateral. Move the secondary debt repayment block to execute immediately before the secondary collateral withdrawal. This requires the strategy to already hold sufficient AUSD at that point – which it will, since the Curvance collateral withdrawal (L878) releases AUSD before the Euler leg needs to be closed.

Alternatively, wrap both Euler operations in `EVC.batch()`, which defers all health checks until the end of the batch and allows temporarily undercollateralized intermediate states:

```
IEVC.BatchItem[] memory items = new IEVC.BatchItem[](2);
items[0] = IEVC.BatchItem({
    targetContract: address(SECONDARY_DEBT_ADAPTER()),
    onBehalfOfAccount: address(this),
    value: 0,
    data: abi.encodeWithSelector(IDebtAdapter.repay.selector, address(STABLECOIN),
    ↪  toRepay)
});
items[1] = IEVC.BatchItem({
    targetContract: address(SECONDARY_COLLATERAL_ADAPTER()),
    onBehalfOfAccount: address(this),
    value: 0,
    data: abi.encodeWithSelector(ICollateralAdapter.withdraw.selector,
    ↪  address(SHMON), shMonCollateral)
});
EVC.batch(items);
```

Discussion

haha-mike

Acknowledged, overlapped with Octave findings. @0xdavid7 is working on this (first resolve Octane side and then work on Sherlock part).

Issue M-2: Curvance integration doesn't account for pending interest [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/issues/37

Summary

It uses function `debtBalance()` to fetch debt from Curvance, however it doesn't include pending interest after last update. Therefore protocol underestimates debt in Curvance.

Vulnerability Detail

It uses `debtBalance()`:

```
function getDebtBalance(
    address,
    /* asset */
    address account
)
    external
    view
    returns (uint256)
{
    // Curvance already exposes debt in underlying asset terms, so no extra share
    ↪  conversion
    // is needed before returning the visible debt balance.
    return DEBT_TOKEN.debtBalance(account);
}
```

As you can see, it doesn't include pending interest:
https://github.com/curvance/curvance-contracts/blob/develop/contracts/market/token/BorrowableCToken.sol#L474-L478

```
    /// @notice Returns the current debt balance for `account`.
@>  /// @dev Note: Pending interest is not applied in this calculation.
    /// @param account The address whose debt balance should be calculated.
    /// @return r The current outstanding debt balance of `account`.
    function debtBalance(address account) public view returns (uint256 r) {
```

For example result of that function is used in yield accrual in `AusdStrategy`:

```
    function _calculateTotalDebtUsd() private view returns (uint256 total) {
        bytes memory primaryDebtData =
@>          abi.encodeWithSelector(IDebtAdapter.getDebtBalance.selector,
↪   address(WMON), address(this));
```

```
        uint256 primaryDebt = abi.decode(address(PRIMARY_DEBT_ADAPTER()).functionSt
        ↪  aticCall(primaryDebtData), (uint256));
        total += _wmonToUsd(primaryDebt);

        bytes memory secondaryDebtData =
            abi.encodeWithSelector(IDebtAdapter.getDebtBalance.selector,
            ↪  address(STABLECOIN), address(this));
        uint256 secondaryDebt =
            abi.decode(address(SECONDARY_DEBT_ADAPTER()).functionStaticCall(seconda
            ↪  ryDebtData), (uint256));
        total += _stablecoinToUsd(secondaryDebt);
    }
```

Impact

Debt is underestimated, it affects following parts:

1) max withdrawal is overestimated, so will revert in edge case when `targetLtv` is close to health threshold.
2) Yield calculation is incorrect.

Code Snippet

https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/blob/b840577fc767f161c09450435c695c94294125ef/smart-vault-contracts/src/adapters/curvance/CurvanceDebtAdapter.sol#L132-L145 https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/blob/b840577fc767f161c09450435c695c94294125ef/smart-vault-contracts/src/adapters/curvance/CurvanceDebtAdapter.sol#L163 https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/blob/b840577fc767f161c09450435c695c94294125ef/smart-vault-contracts/src/adapters/curvance/CurvanceCollateralAdapter.sol#L193

Tool Used

Manual Review

Recommendation

There is no getter in Curvance to fetch current debt directly, so you can simulate that calculation locally.

Discussion

haha-mike

Acknowledged. @0xdavid7 is working on this.

Issue M-3: ERC4626 view `max` functions return incorrect values in `GatedVaultImpl.sol` [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/issues/42

Summary

There is deposit and withdraw limit implemented which is not reflected in ERC4626 functions `maxDeposit()`, `maxMint()`, `maxWithdraw()`, `maxRedeem()`.

Vulnerability Detail

`GatedVaultImpl` inherits `ConcreteAsyncVaultImpl`, which inherits `ConcreteStandardVaultImpl`. As you can see, it implements deposit and withdraw limits:

```
function deposit(uint256 assets, address receiver)
    public
    virtual
    override(IERC4626, ERC4626Upgradeable)
    nonReentrant
    withYieldAccrual
    returns (uint256)
{
    ...
    (uint256 maxDepositAmount, uint256 minDepositAmount) = getDepositLimits();
    require(
        assets + totalAssetsBeforeDeposit <= maxDepositAmount && assets >=
        ↪  minDepositAmount,
        AssetAmountOutOfBounds(msg.sender, assets, minDepositAmount,
        ↪  maxDepositAmount)
    );
    _deposit(_msgSender(), receiver, assets, shares);
    ...
}

function withdraw(uint256 assets, address receiver, address owner)
    public
    virtual
    override(IERC4626, ERC4626Upgradeable)
    nonReentrant
    withYieldAccrual
    returns (uint256)
{
    ...
    (uint256 maxWithdrawAmount, uint256 minWithdrawAmount) = getWithdrawLimits();
    require(
```

```
        assets <= maxWithdrawAmount && assets >= minWithdrawAmount,
        AssetAmountOutOfBounds(_msgSender(), assets, minWithdrawAmount,
        ↪  maxWithdrawAmount)
    );
    ...
}
```

However `ConcreteAsyncVaultImpl` and `ConcreteStandardVaultImpl` don't override functions `maxDeposit()`, `maxMint()`, `maxWithdraw()`, `maxRedeem()` to include that logic. It means that `GatedVaultImpl` inherits those incorrect functions.

`GatedVaultImpl` delegates execution to those incorrect parent functions.

Impact

It breaks ERC4626 specification for functions `maxDeposit()`, `maxMint()`, `maxWithdraw()`, `maxRedeem()`.

Code Snippet

https://github.com/Blueprint-Finance/concrete-earn-v2-bug-bounty/blob/a9ace6154cb2728bc9d3d4c5019e040fb2d4b562/src/implementation/ConcreteStandardVaultImpl.sol#L211-L215 https://github.com/Blueprint-Finance/concrete-earn-v2-bug-bounty/blob/a9ace6154cb2728bc9d3d4c5019e040fb2d4b562/src/implementation/ConcreteStandardVaultImpl.sol#L298-L302

Tool Used

Manual Review

Recommendation

`GatedVaultImpl` should override those functions to include deposit and withdraw limits.

Discussion

haha-mike

Acknowledged. @0xdavid7 is working on this.

Potential solution: create new functions with different names

Issue M-4: `GatedVaultImpl.sol` takes less fee than it should [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/issues/43

Summary

`GatedVaultImpl.sol` inherits `ConcreteAsyncVaultImpl.sol` which inherits `ConcreteStandardVaultImpl.sol`. It takes 2 types of fee during yield accrual. Problem is that it calculates 2 fees sequentially which is mathematically incorrect.

Vulnerability Detail

Let's look into implementation, basically it calculates 2 types of fee: management (fixed fee over time) and performance (fixed over positive yield). It calculates fee in terms of assets and then converts to amount of shares to mint as fee. For example if there are 100 shares and 150 assets with 10% performance fee, it will mint 3.4589 shares as fee. So after this mint users own 145 assets and fee is 5, which is correct.

Problem is that it calculates both fees sequentially, it breaks that share calculation:

```
    function _accrueYield() internal virtual returns (uint256) {
        ...

        // Accrue management fees after accruing yield to calculate fee asset
        ↪  amount on total vault AUM
@>      accrueManagementFee(totalAssetsCached);

        // Accrue performance fees on net yield amount
@>      accruePerformanceFee(totalAssetsCached, totalPositiveYield,
↪   totalNegativeYield);

        return totalAssetsCached;
    }

    function accrueManagementFee(uint256 totalAssetsAmount) internal {
@>      (uint256 feeShares,) = previewManagementFee(totalAssetsAmount);

        ...

        // Mint shares to management fee recipient
        address managementFeeRecipient = $.managementFeeRecipient;
        if (feeShares != 0 && managementFeeRecipient != address(0)) {
            // Mint shares to management fee recipient
            _mint(managementFeeRecipient, feeShares);
        }
```

```
    }
```

Suppose following example:

1) Management fee is 20%; performance fee is 10%; 1 year has passed; `assets = 150`, `shares = 100`
2) Let's calculate correct fee values: `managementFee = 150 * 20% = 30 assets`; `performanceFee = (150 - 100) * 10% = 5 assets`. Correct amount to mint is `30.4347 shares` in total, which converts to 35 assets
3) Now let's calculate fee sequentially as implemented in code. Management fee is 30 assets which converts to 25 shares. Performance fee is 5 assets which converts to `4.3103 shares`
4) So after sequential fee accrual, `totalSupply = 100 + 25 + 4.3103 = 129.3103`. `managementFee = 150 * 25 / 129.3103 = 29 assets` and `performanceFee = 150 * 4.3103 / 129.3103 = 5 assets`
5) As you can see, real values deviate from mathematically correct ones. Correct amounts to mint are: `managementFee = 26.0869 shares`, `performanceFee = 4.3478 shares`

Impact

Above example shows 2.8% loss from mathematically correct amount. Actual loss in real world is much less to be honest. For example with config `10% APR`, `after 1 year`, `1% managementFee`, `3% performanceFee` deviation from correct amount is only 0.2%. And it decreases significantly with the reduction of the accrual period, which, as the examples show, is 1 year.

Code Snippet

https://github.com/Blueprint-Finance/concrete-earn-v2-bug-bounty/blob/a9ace6154cb2728bc9d3d4c5019e040fb2d4b562/src/implementation/ConcreteStandardVaultImpl.sol#L653-L657 https://github.com/Blueprint-Finance/concrete-earn-v2-bug-bounty/blob/a9ace6154cb2728bc9d3d4c5019e040fb2d4b562/src/implementation/ConcreteStandardVaultImpl.sol#L773-L815

Tool Used

Manual Review

Recommendation

Ideally issue should be fixed on Concrete side. Correct way is to calculate total fee amount in assets; then based on assets calculate total shares to mint. And then split

total shares minted accordingly between 2 fee types.

Discussion

haha-mike

Acknowledged. @0xdavid7 is working on this.

T1MOH593

It should always update variable `lastManagementFeeAccrual`:

```
    /// @dev Accrues management/performance fees using a combined share mint
    ↪  calculation.
    function _accrueFeesCombined(
        uint256 totalAssetsCached,
        uint256 totalPositiveYield,
        uint256 totalNegativeYield
    )
        internal
    {
        SVLib.ConcreteStandardVaultImplStorage storage \$ = SVLib.fetch();
        (
            uint256 managementFeeAmount,
            uint256 performanceFeeAmount,
            uint256 totalCollectibleFeeAmount,
            address managementFeeRecipient,
            address performanceFeeRecipient
        ) = _previewCollectibleFees(totalAssetsCached, totalPositiveYield,
        ↪  totalNegativeYield);

        if (managementFeeAmount > 0) {
@>          \$.lastManagementFeeAccrual = Time.timestamp();
        }
```

It will take massive fee if there was previously 0 fee. Suppose following:

1) Vault is deployed with 0 fee
2) After 1 year you turn on fee
3) It will take fee for previous 1 year because variable still points to deployment timestamp.

So correct way is to always update it:

```
-       if (managementFeeAmount > 0) {
            \$.lastManagementFeeAccrual = Time.timestamp();
-       }
```

0xdavid7

Fixed at: https://github.com/Permutize/smart-vault-contracts/pull/181/changes/4dc1d4048a795914c9f732ddeed1b04dc664abb2

Issue M-5: `BaseStrategy.returnIdleToVault()` reverts because of Curvance hold period [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/issues/44

Summary

During deallocation it always ensures there is no hold period in Curvance. This extensive check block other legitimate actions such as `returnIdleToVault()`.

Vulnerability Detail

As you can see, `deallocateFunds()` always calls `CurvanceCollateralAdapter.verifyCooldownPassed()`, which ensures hold period has passed:

```
    function deallocateFunds(bytes calldata data) public virtual onlyVault returns
    ↪  (uint256 assetsReturned) {
        ...
        $.deallocationTracker = 0; // Reset tracker at start

        _verifyCooldowns();
        _executeCommands(batch.commands);
        ...
        $.deallocationTracker = 0; // Clear tracker after use
    }

    function _verifyCooldowns() internal view virtual {
        BaseStrategyStorageLib.BaseStrategyStorage storage $ =
        ↪  _getBaseStrategyStorage();
@>      $.primaryCollateralAdapter.verifyCooldownPassed();
        $.secondaryCollateralAdapter.verifyCooldownPassed();
    }
```

```
    function verifyCooldownPassed() external view {
        _verifyCooldownPassed();
    }

    /**
     * @notice Verify that Curvance cooldown period has passed
@>   * @dev Curvance records the timestamp of the most recent borrow or collateral
↪   action.
     *      A withdrawal is only allowed once that timestamp plus MIN_HOLD_PERIOD
     ↪  is in the past.
     */
    function _verifyCooldownPassed() internal view {
```

```
        // Curvance stores the last relevant account action timestamp in
        ↪  accountAssets. A zero
        // value means the account has no active cooldown to enforce.
        uint256 cooldownTimestamp = MARKET_MANAGER.accountAssets(address(this));

        if (cooldownTimestamp > 0) {
            uint256 minHoldPeriod = MARKET_MANAGER.MIN_HOLD_PERIOD();
            uint256 cooldownEnd = cooldownTimestamp + minHoldPeriod;

            if (block.timestamp < cooldownEnd) {
                uint256 remaining = cooldownEnd - block.timestamp;
                revert CooldownNotPassed(remaining);
            }
        }
    }
```

The only action which updates `deallocationTracker` and must be called via `deallocateFunds()` is `returnIdleToVault()`. This function transfers AUSD back from Strategy to Vault. Such action doesn't involve Curvance at all, still it's blocked for no reason.

Impact

Strategy can't transfer AUSD to Vault during 20 minutes period after collateral deposit and borrow.

Code Snippet

https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/blob/main/smart-vault-contracts/src/strategies/BaseStrategy.sol#L456
https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/blob/main/smart-vault-contracts/src/strategies/BaseStrategy.sol#L939-L943
https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/blob/main/smart-vault-contracts/src/adapters/curvance/CurvanceCollateralAdapter.sol#L228-L242

Tool Used

Manual Review

Recommendation

Consider removing check at all. Curvance will anyway perform it when necessary

```
    function deallocateFunds(bytes calldata data) public virtual onlyVault returns
    ↪  (uint256 assetsReturned) {
        BatchCommand memory batch = _decodeBatchCommand(data);
```

```
        _enforceNotPaused(batch.allowWhenPaused);

        BaseStrategyStorageLib.BaseStrategyStorage storage $ =
        ↪  _getBaseStrategyStorage();
        $.deallocationTracker = 0; // Reset tracker at start

-       _verifyCooldowns();
        _executeCommands(batch.commands);

        // deallocationTracker now contains the cumulative amount returned via
        ↪  returnIdleToVault
        assetsReturned = $.deallocationTracker;
        $.deallocationTracker = 0; // Clear tracker after use

        emit BatchCommandDeallocated(batch.id, assetsReturned);
    }
```

Discussion

haha-mike

Acknowledged, overlap with Octane findings. @0xdavid7 is working on this, first Octane merge, then remaning changes from Sherlock.

Issue M-6: Curvance's min loan size is checked incorrectly [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/issues/48

Summary

Curvance checks that final debt is greater than min loan size, while Curvance adapter in that protocol assumes it checks that borrowed amount in current call must be greater.

Vulnerability Detail

`CurvanceDebtAdapter.getMaxBorrowAmount()` checks that available amount to borrow is greater than `MIN_LOAN_SIZE`:

```
    function getMaxBorrowAmount(...)
        external
        view
        returns (uint256)
    {
        // Read collateral in underlying asset units instead of cToken share units
        ↪  before passing
        // it into the generic LTV helper.
        uint256 collateralShares = COLLATERAL_TOKEN.collateralPosted(account);
        uint256 collateral = COLLATERAL_TOKEN.convertToAssets(collateralShares);
        uint256 debt = DEBT_TOKEN.debtBalance(account);

        // First compute the generic oracle-aware LTV borrow ceiling for the
        ↪  current position.
        uint256 finalBorrow = LTVCalculations.calculateMaxBorrow(...);

        // Curvance adds a protocol-level minimum loan size denominated in
        ↪  18-decimal USD terms.
        // A borrow below that threshold would revert or leave the position
        ↪  invalid, so clamp it.
        uint256 minLoanSize = MARKET_MANAGER.MIN_LOAN_SIZE();

        // Convert the tentative borrow amount into USD before comparing against
        ↪  Curvance's global
        // minimum loan size threshold.
        if (finalBorrow > 0) {
            uint256 finalBorrowUsd =
                OracleHelpers.convertToUsd(finalBorrow, DEBT_ORACLE, DEBT_DECIMALS,
                ↪  DEBT_ORACLE_HEARTBEAT);

@>          // If the amount is too small to satisfy the market's minimum loan
↪  size, skip the
```

```
@>          // borrow entirely rather than returning an amount that the strategy
 ↪  cannot use safely.
@>          if (finalBorrowUsd < minLoanSize) {
                return 0;
            }
        }

        return finalBorrow;
    }
```

Actually Curvance checks that final borrow amount is greater than `MIN_LOAN_SIZE`. Let's consider scenario:

- Max borrow is 50; current debt is 35; min loan size is 20
- Remaining to borrow 15
- It won't allow to borrow 15, because `15 < min loan size`
- However Curvance will allow it, because checks that final debt after borrow is greater than 20, i.e. always in this example

That check is implemented in `LiquidityManagerIsolated._hypotheticalLiquidityOf()`:

```
                if (action.cTokenModified == snap.asset) {
@>                  snap.debtBalance += action.borrowAssets;
                }

                // CASE: There is no outstanding debt, clean up the position
                // entry as the user was liquidated, otherwise add to the
                // user's outstanding debt.
                if (snap.debtBalance > 0) {
                    // CASE: There is outstanding debt to lenders, add it to
                    // `newDebt` to check against `maxDebt`.
                    newDebt += _assetValue(
                        snap.debtBalance,
                        prices[i],
                        10 ** snap.decimals,
                        false
                    );

                    // Check `newDebt` to make sure the loan size will not be
                    // too small for us to allow issuing the loan.
@>                  if (newDebt < MIN_LOAN_SIZE) {
                        revert LiquidityManager__InsufficientLoanSize();
                    }
                }
```

There `_assetValue` converts `snap.debtBalance` to USD value and performs a check. `snap.debtBalance` contains new debt from current call: `snap.debtBalance += action.borrowAssets;` and previous debt which can be tracked in following function:

https://github.com/curvance/curvance-contracts/blob/0e05deb4ea19f97a948b531c413f7b8305907bc5/contracts/market/token/BorrowableCToken.sol#L461-L472

```
function getSnapshot(
    address account
) public view override returns (AccountSnapshot memory result) {
    uint256 outstandingDebt = debtBalance(account);

    result.asset = address(this);
    result.underlying = address(_asset);
    result.decimals = decimals();
    result.isCollateral = outstandingDebt > 0 ? false : true;
    result.collateralPosted = collateralPosted[account];
    result.debtBalance = outstandingDebt;
}
```

Impact

Curvance adapter reverts on legitimate borrow action.

Code Snippet

https://github.com/curvance/curvance-contracts/blob/0e05deb4ea19f97a948b531c413f7b8305907bc5/contracts/market/isolated/LiquidityManagerIsolated.sol#L445-L467
https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/blob/main/smart-vault-contracts/src/adapters/curvance/CurvanceDebtAdapter.sol#L267-L285

Tool Used

Manual Review

Recommendation

Check final debt:

```
        uint256 debt = DEBT_TOKEN.debtBalance(account);

        // First compute the generic oracle-aware LTV borrow ceiling for the
        ↪  current position.
        uint256 finalBorrow = LTVCalculations.calculateMaxBorrow(...);

        // Curvance adds a protocol-level minimum loan size denominated in
        ↪  18-decimal USD terms.
        // A borrow below that threshold would revert or leave the position
        ↪  invalid, so clamp it.
        uint256 minLoanSize = MARKET_MANAGER.MIN_LOAN_SIZE();
```

```
        // Convert the tentative borrow amount into USD before comparing against
        ↪  Curvance's global
        // minimum loan size threshold.
        if (finalBorrow > 0) {
-           uint256 finalBorrowUsd =
+           uint256 newDebtUsd =
                OracleHelpers.convertToUsd(finalBorrow, DEBT_ORACLE, DEBT_DECIMALS,
                ↪  DEBT_ORACLE_HEARTBEAT);
+           uint256 previousDebtUsd =
+               OracleHelpers.convertToUsd(debt, DEBT_ORACLE, DEBT_DECIMALS,
↪  DEBT_ORACLE_HEARTBEAT);

            // If the amount is too small to satisfy the market's minimum loan
            ↪  size, skip the
            // borrow entirely rather than returning an amount that the strategy
            ↪  cannot use safely.
-           if (finalBorrowUsd < minLoanSize) {
+           if (newDebtUsd + previousDebtUsd < minLoanSize) {
                return 0;
            }
        }

        return finalBorrow;
```

And also update comments

Discussion

haha-mike

Acknowledged, overlapped with Octane findings. @0xdavid7 is working on this, will check after Octane issues merged

0xdavid7

It is different from the Octane issue. I already fixed it

Issue M-7: `_terminatePositions` Reverts When WMON Balance Cannot Cover Accrued Debt [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/issues/49

Summary

`_terminatePositions` hard-reverts if the strategy's WMON balance is less than its Curvance WMON debt at the moment of termination. Because shMON unstaking is asynchronous – the strategy must wait for all pending unstake workers to complete before termination can proceed – WMON debt continues accruing interest during the waiting period. If interest outpaces shMON yield, or if termination is triggered shortly after the position was opened before meaningful yield has accumulated, the WMON balance falls short of the debt and termination is permanently blocked.

Vulnerability Detail

Termination requires all unstake workers to be finalized before proceeding:

```
// AusdStrategy.sol:L843-L844
_finalizeMaturedUnstakes();
_assertNoPendingShMonadUnstakes();
```

`_assertNoPendingShMonadUnstakes` reverts if any active unstake request exists, meaning the strategy must wait for ShMonad's unstaking cooldown (which spans one or more epochs) before the WMON can be recovered. During this wait the Curvance WMON borrow is still accruing interest.

Once all unstakes complete and the WMON is recovered, the code does a hard underflow check:

```
// AusdStrategy.sol:L860-L866
uint256 wmonDebt = _getDebtBalance(address(PRIMARY_DEBT_ADAPTER()), address(WMON));
uint256 wmonBalance = WMON.balanceOf(address(this));
if (wmonDebt > 0 && wmonBalance > 0) {
    if (wmonBalance < wmonDebt) {
        revert InsufficientPrimaryDebtCoverage(wmonBalance, wmonDebt, wmonDebt -
        ↪  wmonBalance);
    }
    ...
}
```

There is no partial repayment path here – the code either repays the full debt or reverts. This creates three compounding shortfall sources:

1. **Interest accrual during the unstake wait**: Curvance WMON debt grows during the multi-epoch unstake window while no fresh WMON is entering the strategy.
2. **Early termination (yield < interest + fees)**: If the strategy is terminated shortly after opening, the shMON position has earned little yield while the WMON borrow has already incurred origination fees and initial interest. The recovered WMON from unstaking the shMON collateral is less than the outstanding debt.
3. **AtomicUnstakePool fee on `redeem`**: The termination path at L855 calls `_unstakeShMonToMon(shMonBalance, 0)`, which routes through `SwapAdapter.unstakeShMonToMon` and calls `shMon.redeem(shares, ...)`. ShMonad's ERC-4626 `redeem` routes through its AtomicUnstakePool and deducts a utilization-based fee before returning MON. The `minMonOut = 0` argument provides zero slippage protection, so the strategy silently accepts the full fee deduction. This means the actual WMON recovered is `shares × PPS − fee`, which is strictly less than the nominal collateral value and further widens the gap against the outstanding debt.

In any of these cases, `wmonBalance < wmonDebt` and `_terminatePositions` reverts unconditionally. The strategy cannot be terminated through the normal code path until the OPERATOR manually injects extra WMON, which requires either sourcing external funds or waiting an indeterminate time for yield to accumulate.

Impact

The strategy can become permanently unterminated via the standard path whenever WMON debt exceeds the recoverable shMON value. This blocks vault deallocation and traps user funds in the strategy. Recovery requires admin-level intervention to inject WMON from outside the strategy – there is no on-chain self-recovery mechanism. The risk is highest at early exit (before yield matures) or during a sustained period where Curvance WMON borrow rates exceed shMON staking yield.

Tool Used

Manual Review

Recommendation

Replace the hard revert with a partial repay that uses all available WMON, and emit an event signaling that termination is incomplete due to a shortfall:

```
if (wmonDebt > 0 && wmonBalance > 0) {
    uint256 toRepay = wmonBalance < wmonDebt ? wmonBalance : wmonDebt;
    if (wmonBalance < wmonDebt) {
        emit TerminationShortfall(address(WMON), wmonDebt - wmonBalance);
    }
    bytes memory repayData = abi.encodeWithSelector(IDebtAdapter.repay.selector,
    ↪  address(WMON), toRepay == wmonDebt ? uint256(0) : toRepay);
```

```
    _delegateCallVoid(address(PRIMARY_DEBT_ADAPTER()), repayData);
    primaryDebtClosedThisCall = toRepay == wmonDebt;
}
```

Additionally, consider tracking the shortfall so a follow-up termination call can resume once the operator supplies the missing WMON, rather than forcing a full restart of the termination sequence.

Discussion

haha-mike

Fixed in Octane list. @0xdavid7 please verify.

0xdavid7

I already fixed it

Issue L-1: Plain Text Private Key Used Across Different Administrative Scripts [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/issues/26

Summary

Multiple deployment and administrative scripts accept private keys via command-line arguments but rely on plaintext environment variables as fallback mechanisms. This creates exposure risks where sensitive private keys are stored in plaintext in process memory and environment listings.

Vulnerability Detail

The core vulnerability exists in `script/ts/utils/auth.ts`, where the `resolvePrivateKey` function retrieves private keys from environment variables when no command-line argument is provided:

```
const envPk = process.env.PRIVATE_KEY ?? process.env.DEPLOYER_PRIVATE_KEY;
if (envPk) return normalizePk(envPk);
```

Plaintext environment variables are vulnerable to:

1. **Process visibility**: Environment variables are visible via `ps` commands or `/proc/[pid]/environ`
2. **History exposure**: If set via shell commands, they may appear in shell history
3. **Child process inheritance**: All child processes inherit the parent's environment
4. **Memory access**: Plaintext in memory is susceptible to core dumps and memory inspection tools

Impact

Exposure of private keys grants attackers complete control over:

- Vault deployments and upgrades
- Strategy modifications and rebalancing
- Fee allocation parameters
- Role-based access control management
- Ownership transfers of critical contracts

Code Snippet

Vulnerable pattern across scripts:

All the following scripts use `resolvePrivateKey` or fallback to `PRIVATE_KEY` environment variable through CLI option parsing:

1. script/ts/deploy-all.cli.ts - `.option("--pk <key>", "Private key")`
2. script/ts/deploy-fee-allocator.cli.ts - uses `resolvePrivateKey`
3. script/ts/grant-roles.cli.ts - uses `resolvePrivateKey`
4. script/ts/merkl-rewards.cli.ts - `.option("--pk <key>", "Private key", process.env.PRIVATE_KEY ?? process.env.DEPLOYER_PRIVATE_KEY)`
5. script/ts/prepare-vault-upgrade.cli.ts - `envPk: ["PRIVATE_KEY"]`
6. script/ts/transfer-concrete-vault-ownership.cli.ts - `envPk: ["CONCRETE_VAULT_OWNER_PRIVATE_KEY", "VAULT_OWNER_PRIVATE_KEY", "PRIVATE_KEY"]`
7. script/ts/cancel-vault-upgrade.cli.ts - `envPk: ["VAULT_MANAGER_PRIVATE_KEY"]`
8. script/ts/upgrade-strategy.cli.ts - `envPk: ["PRIVATE_KEY", "DEPLOYER_PRIVATE_KEY"]`
9. script/ts/execute-vault-upgrade.cli.ts - `envPk: ["PRIVATE_KEY"]`
10. script/ts/transfer-concrete-factory-ownership.cli.ts - `envPk: ["CONCRETE_FACTORY_OWNER_PRIVATE_KEY", "FACTORY_OWNER_PRIVATE_KEY", "PRIVATE_KEY"]`
11. script/ts/terminate-positions.cli.ts - `envPk: ["PRIVATE_KEY", "DEPLOYER_PRIVATE_KEY"]`
12. script/ts/setup-vault-timelock.cli.ts - uses `resolvePrivateKey`

Tool Used

Manual Review

Recommendation

Do the following steps to mitigate this vulnerability:

1. **Remove environment variable fallback**: Require explicit `--pk` argument instead of reading from `PRIVATE_KEY`
2. **Use Foundry Keystore**: Enforce the `--account` / `--password` keystore path which is already implemented as an alternative

Discussion

haha-mike

Acknowledged. @haha-mike will work on that.

Issue L-2: `RewardsAdapter` Only Claims Merkl Rewards — Curvance CVE Emissions and Euler Reward Streams Are Permanently Stranded [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/issues/30

Summary

The strategy's reward harvesting infrastructure is built exclusively around the Merkl off-chain distributor. Curvance distributes CVE gauge emissions through a separate on-chain `RewardManager` using a biweekly epoch + veCVE points system, and Euler distributes incentives through an on-chain `RewardStreams` contract (requiring `enableBalanceForwarding` opt-in). Neither channel routes through Merkl. As a result, any CVE emissions accruing to the strategy's Curvance collateral position and any Euler Reward Streams configured for the shMON vault are permanently uncollectable by the strategy – there is no code path to claim them.

Vulnerability Detail

`RewardsAdapter` exposes three reward entry points, all of which funnel exclusively through `MerklAdapter.claim()`:

```
// RewardsAdapter.sol:68-99
function claimRewards(address distributor, MerklClaimRequest calldata claimRequest)
    external onlyRewardsManager {
    _claimRewards(distributor, claimRequest);  // → MerklAdapter.claim() only
}

function _claimRewards(address distributor, MerklClaimRequest calldata
↪   claimRequest) internal {
    MerklAdapter.claim(distributor, address(this), claimRequest);
    // ↑ Only ever calls IMerklDistributor.claim() - no other reward contracts
}
```

Neither `ICurvance.sol` nor `IEuler.sol` expose any reward-claiming functions. The entire adapter suite – `CurvanceCollateralAdapter`, `CurvanceDebtAdapter`, `EulerCollateralAdapter`, `EulerDebtAdapter` – contains zero calls to reward contracts.

Curvance reward mechanism (on-chain, not Merkl):

Curvance distributes CVE gauge emissions through `RewardManager.claimRewards()`. Rewards accrue per user per biweekly epoch based on veCVE points proportional to posted collateral. The cross-chain fee/reward flow uses Circle CCTP or Wormhole – entirely independent of Merkl. There is no Merkl integration in Curvance's reward architecture.

Euler reward mechanism (dual-channel, partly not Merkl):

Euler operates two separate reward channels:

1. **Merkl (off-chain)** – campaigns updated every ~8-12 hours, claimable via Merkl dashboard. These accrue automatically. The `RewardsAdapter` handles this channel correctly.
2. **Reward Streams (on-chain)** – the permissionless "billion-dollar algorithm" contract. Users must actively opt-in via `enableBalanceForwarding`. The strategy has never called this function and has no interface to the `RewardStreams` contract. Any reward streams configured for the shMON collateral vault are uncollectable.

The `IEuler.sol` interface (529 lines) contains zero `RewardStreams` or `enableBalanceForwarding` methods. No test in `AusdStrategyRewardsTest.t.sol` covers either native protocol reward channel.

Impact

Any Curvance CVE gauge emissions allocated to the strategy's collateral position accumulate in `RewardManager` with no on-chain mechanism to extract them. Similarly, if Euler Reward Streams are configured for the shMON collateral vault, those rewards accumulate unclaimed. In both cases the value is lost to the vault depositors – it sits in the protocol reward contracts indefinitely. The `REWARDS_MANAGER` role has no privileged path to recover these rewards, and the strategy admin cannot add new reward claim functions without a contract upgrade. Operators running `claimAndProcessRewards()` receive incomplete yield – they harvest only the Merkl slice of total protocol rewards, with no indication that other reward streams exist or are being missed.

Code Snippet

```
// RewardsAdapter.sol:123-126
function _claimRewards(address distributor, MerklClaimRequest calldata
↪  claimRequest) internal {
    MerklAdapter.claim(distributor, address(this), claimRequest);
    emit RewardsClaimed(address(this), distributor, claimRequest.tokens.length);
}
```

```
// MerklAdapter.sol:73-101
function claim(address distributor, address recipient, MerklClaimRequest calldata
↪  claimRequest) internal {
    _validateClaim(distributor, claimRequest);
    // ... only calls IMerklDistributor(distributor).claim(...)
    // No CVELocker call. No RewardStreams call.
}
```

Tool Used

Manual Review

Recommendation

Add dedicated reward claim adapters for each protocol's native distribution channel:

Curvance CVE emissions – add a call to GaugeManager's function `claim(address[] calldata tokens, address user)` (from here).

Euler Reward Streams – call `enableBalanceForwarder()` (from here) on the Euler collateral vault during the initial deposit setup so the strategy immediately begins accruing on-chain rewards. Then call Reward Stream contract function `claimReward(address rewarded, address reward, address recipient, bool ignoreRecentReward)` (from here)

Alternatively, if the protocol intends to rely solely on Merkl for all reward distribution, this should be explicitly documented and confirmed with both the Curvance and Euler teams that all incentives for the strategy's positions will be routed through Merkl campaigns – and that no native on-chain reward streams will be active for those markets.

Discussion

haha-mike

Acknowledged. @0xdavid7 is working on this. Low priority we may fix it later.

T1MOH593

Noticed initial report refers wrong functions, updated function signatures in mitigation step. Actual functions to call are: Euler RewardStream

```
claimReward(
        address rewarded,
        address reward,
        address recipient,
        bool ignoreRecentReward
    )
```

Euler Vault

```
/// @inheritdoc IBalanceForwarder
function enableBalanceForwarder() public virtual nonReentrant {
```

Curvance's GaugeManager

```
/// @notice Claim all pending rewards for `tokens` from the Gauge Manager.
/// @param tokens Array containing pool token addresses to claim
///               rewards for.
```

```
/// @param user The user address that gauge rewards should be claimed for,
///             if `user` is not the caller, plugin delegation status is
///             checked.
function claim(
    address[] calldata tokens,
    address user
) external nonReentrant {
```

Sorry for previous incorrect mitigation

T1MOH593

@0xdavid7 We've found following inconsistencies regarding current fix to issue 30, please review:

1) Euler's `claimReward()` has return value `uint256`, but call expects no return value. So it will revert

```
    function claimEulerRewards(
        address rewardStreams,
        address[] calldata rewardTokens,
        bool ignoreRecentReward
    )
        external
        onlyRewardsManager
    {
        if (rewardStreams == address(0)) revert InvalidRewardTarget();
        uint256 rewardTokenCount = rewardTokens.length;
        for (uint256 i; i < rewardTokenCount; ++i) {
@>          _functionCallNoReturn(
                rewardStreams,
                abi.encodeWithSelector(
                    EULER_CLAIM_REWARD_SELECTOR, address(this), rewardTokens[i],
                    ↪  address(this), ignoreRecentReward
                )
            );
        }
        emit EulerRewardsClaimed(rewardStreams, address(this), rewardTokens.length);
    }
```

2) Incorrect parameter is submitted. First argument `rewarded` refers to Euler Vault, but current fix submits address(this). You should submit address of Euler vault.

```
/// @notice Claims earned reward.
/// @dev Rewards are only transferred to the recipient if the recipient is non-zero.
/// @param rewarded The address of the rewarded token.
/// @param reward The address of the reward token.
/// @param recipient The address to receive the claimed reward tokens.
/// @param ignoreRecentReward Whether to ignore the most recent reward and not
↪  update the accumulator.
/// @return The amount of the claimed reward tokens.
```

```
function claimReward(
    address rewarded,
    address reward,
    address recipient,
    bool ignoreRecentReward
) external virtual override nonReentrant returns (uint256) {
```

3) Function `RewardStream.enableReward(address rewarded, address reward)` must also be called additionally to `enableBalanceForwarder()` . Otherwise contract won't track those rewards and therefore it can't be claimed. Note this call is on RewardStream itself, not Vault. https://github.com/euler-xyz/reward-streams/blob/master/src/BaseRewardStreams.sol allowbreak #L271

Issue L-3: `pullFromVault` and `returnIdleToVault` Are Not Context-Restricted — Wrong-Context Use Silently Corrupts Vault Accounting [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/issues/31

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`pullFromVault` and `returnIdleToVault` are lifecycle commands with strict directional semantics: `pullFromVault` belongs only in allocation batches; `returnIdleToVault` belongs only in deallocation batches. However, the command dispatcher `_executeCommands` applies no context check – both commands are accepted in either context. Because each command updates a different tracker (`allocationTracker` vs `deallocationTracker`) and each context function reads only its own tracker, including a command in the wrong context causes its accounting effect to be silently discarded by the vault.

Impact

An OPERATOR can cause the vault to record an incorrect `delta` or `assetsReturned` for a batch execution. The vault's internal accounting of how much capital is deployed in each strategy drifts from reality, which affects share pricing, withdrawal capacity calculations, and NAV reporting. The discrepancy persists silently until a future yield accrual or full reconciliation corrects it.

Code Snippet

```
// BaseStrategy.sol:1043-1051 - no context check, all commands accepted in both
↪  contexts
function _executeCommands(Command[] memory commands) internal {
    uint256 length = commands.length;
    for (uint256 i = 0; i < length; ++i) {
        bytes4 commandSelector = _getCommandSelector(commands[i].id);
        (bool success, bytes memory returnData) =
            address(this).call(abi.encodeWithSelector(commandSelector,
            ↪  commands[i].data));
        if (!success) _bubbleRevert(returnData);
    }
}
```

```
// StrategyCommandSelectors.sol:13,49 - lifecycle commands with opposite
↪  directional semantics
```

```
bytes32 internal constant SELECTOR_PULL_FROM_VAULT      =
↪  keccak256("pullFromVault");
bytes32 internal constant SELECTOR_RETURN_IDLE_TO_VAULT =
↪  keccak256("returnIdleToVault");
```

Tool Used

Manual Review

Recommendation

Add a context flag to the execution pipeline and enforce it per command. The minimal fix is an `ExecutionContext` enum (`ALLOCATE`, `DEALLOCATE`) stored transiently for the duration of `_executeCommands`, with a check at the wrapper level for the two lifecycle commands:

```
// Revert if pullFromVault is called outside an allocation context
function pullFromVault(bytes memory data) external onlySelf {
    if (_getExecutionContext() != ExecutionContext.ALLOCATE) revert WrongContext();
    uint256 pulled = _pullFromVault(abi.decode(data, (uint256)));
    emit PulledFromVault(pulled);
}

// Revert if returnIdleToVault is called outside a deallocation context
function returnIdleToVault(bytes memory data) external onlySelf {
    if (_getExecutionContext() != ExecutionContext.DEALLOCATE) revert
    ↪  WrongContext();
    ...
}
```

Discussion

haha-mike

Acknowledged. @0xdavid7 is working on this. Fixed in beetween PR.

Issue L-4: `getMaxSafeWithdrawal` Always Passes Zero Safety Buffer [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/issues/32

Summary

`_withdrawCollateral` hardcodes `safetyBuffer = 0` when calling `getMaxSafeWithdrawal`. This means the adapter computes the maximum withdrawal that lands the position exactly at `targetLtv` with no margin. A single oracle price tick after withdrawal can push the position above the target LTV.

Vulnerability Detail

```
// BaseStrategy.sol:L654-L658
bytes memory data = abi.encodeWithSelector(
    ICollateralAdapter.getMaxSafeWithdrawal.selector, token, address(this),
    ↪  targetLtv, 0  // safetyBuffer hardcoded to 0
);
```

`LTVCalculations.calculateSafeWithdrawal` uses this buffer to compute a conservative withdrawal:

```
uint256 conservativeLtv = targetLtv > bufferBps ? targetLtv - bufferBps : targetLtv;
```

With `bufferBps = 0`, `conservativeLtv == targetLtv` – no margin is preserved. The maximum withdrawable amount places the position precisely at the target LTV boundary.

Impact

After an auto-compute withdrawal (`amount = 0`), the position sits exactly at `targetLtv`. Any oracle price movement or interest accrual between the withdrawal and the next block can push the actual LTV above the target, making the position eligible for liquidation sooner than intended.

Code Snippet

`BaseStrategy.sol:L654-L658`

Tool Used

Manual Review

Recommendation

Pass a non-zero safety buffer (e.g. 50–100 bps) to preserve a margin below the target.

Discussion

PeterSRWeb3

This issue was described by https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/pull/16 allowbreak #discussion_r3067886160

haha-mike

Acknowledged, potential overlap with Octane. @0xdavid7 is working on this.

Issue L-5: `setBorrowCapThresholds` Stores Values That Are Never Read On-Chain [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/issues/33

Summary

`setBorrowCapThresholds` writes `primaryBorrowCapThresholdBps` and `secondaryBorrowCapThresholdBps` to storage and validates their range, but no on-chain code ever reads these values. The thresholds have no effect on borrow behavior.

Vulnerability Detail

The setter stores two basis-point values defaulting to 90% at initialization:

```
// BaseStrategy.sol:L179-L180
$.primaryBorrowCapThresholdBps = DEFAULT_BORROW_CAP_THRESHOLD_BPS; // 90%
$.secondaryBorrowCapThresholdBps = DEFAULT_BORROW_CAP_THRESHOLD_BPS; // 90%
```

However, `_borrow` passes only `targetLtv` and `maxDebt` to the adapter – neither threshold value is referenced:

```
// BaseStrategy.sol:L833-L839
if (amount == 0) {
    bytes memory data = abi.encodeWithSelector(
        IDebtAdapter.getMaxBorrowAmount.selector, token, address(this), targetLtv,
        ↪  maxDebt
    );
    bytes memory result = address(adapter).functionStaticCall(data);
    amount = abi.decode(result, (uint256));
}
```

Both `CurvanceDebtAdapter.getMaxBorrowAmount` and `EulerDebtAdapter.getMaxBorrowAmount` accept only `targetLtv` and `maxDebt` – there is no code path that reads the stored threshold values.

Impact

The stored thresholds have no on-chain effect. An operator who believes setting a lower threshold will protect against over-borrowing will find the setting silently ignored.

Code Snippet

`BaseStrategy.sol:L1179-L1187`

Tool Used

Manual Review

Recommendation

Either wire the thresholds into `getMaxBorrowAmount`, or remove the storage fields, setter, and event entirely and document that borrow cap enforcement is solely via `targetLtv` in batch payloads.

Discussion

PeterSRWeb3

This issue was described by https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/pull/16 allowbreak #discussion_r3067892271

haha-mike

Acknowledged. @haha-mike is working on this.

Issue L-6: `setBasisGuardBps` Emits No Event [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/issues/34

Summary

`setBasisGuardBps` modifies the single parameter controlling strategy liveness without emitting any event. Changes are invisible to off-chain monitoring until transactions start failing.

Vulnerability Detail

```
// AusdStrategy.sol:L762-L768
function setBasisGuardBps(uint256 newBasisGuardBps) external onlyStrategyAdmin {
    if (newBasisGuardBps > LTVCalculations.BPS) {
        revert InvalidBasisGuardBps(newBasisGuardBps);
    }
    AusdStrategyStorageLib.fetch().basisGuardBps = newBasisGuardBps;
    // No event emitted
}
```

Impact

No on-chain signal when the basis guard changes. Monitoring systems cannot detect misconfiguration until allocation/deallocation transactions begin failing.

Code Snippet

`AusdStrategy.sol:L762-L768`

Tool Used

Manual Review

Recommendation

```
event BasisGuardBpsUpdated(uint256 oldValue, uint256 newValue, address updatedBy);

emit BasisGuardBpsUpdated(oldValue, newBasisGuardBps, msg.sender);
```

Discussion

haha-mike

Acknowledged. @0xdavid7 is working on this.

Issue L-7: Missing Duplicate Check When Adding Partners [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/issues/36

Summary

The `addPartner` function allows the same address to be registered multiple times, enabling duplicate entries in the `partners` array. This leads to overallocation of fees and potential overpayment to a single recipient.

Vulnerability Detail

No duplicate address check exists in `addPartner`:

```
function addPartner(address account, uint16 basisPoints) external onlyOwner {
    if (account == address(0)) revert ZeroAddress();
    if (basisPoints == 0) revert InvalidBasisPoints();
    if (totalAllocatedBps + basisPoints > LTVCalculations.BPS) revert
    ↪  ExceedsMaxAllocation();

    //  Missing duplicate check
    partners.push(Partner({ account: account, basisPoints: basisPoints, active:
    ↪  true }));

    totalAllocatedBps += basisPoints;
    emit PartnerAdded(account, basisPoints);
}
```

During fee distribution, the same address can be processed multiple times:

```
uint256 partnerAmount = (distributableBalance * partner.basisPoints) /
↪  LTVCalculations.BPS;
_payOrReserve(token, partner.account, partnerAmount); // Same account paid
↪  repeatedly
```

This also causes inconsistent accounting in `pendingPartnerFees[token][partner.account]` when payments fail and fallback to reservation.

Impact

- A single address can receive significantly more than its intended share (including up to 100% of fees).
- Overpayment in a single distribution round, underpaying other partners.

- Corrupted pending fee tracking due to shared mapping keys.

Recommendation

Add a duplicate check in `addPartner`:

```
// Prevent duplicates
for (uint256 i = 0; i < partners.length; i++) {
    if (partners[i].account == account && partners[i].active) {
        revert DuplicatePartner(account);
    }
}
```

Or use a mapping for O(1) checks (recommended for scalability):

```
mapping(address => uint256) public partnerIndex;
```

Add error: `error DuplicatePartner(address account);` Update `removePartner` to clear the mapping entry.

Discussion

haha-mike

Acknowledged. @0xdavid7 is working on this.

Issue L-8: Inconsistent comment in `CurvanceDebtAdapter.resolveRepayInstruction()` [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/issues/38

Vulnerability Detail

There is min loan size in Curvance, so repay will revert if remaining debt is too small. It handles such case, comment says it computes largest safe partial repayment:

```
function resolveRepayInstruction(
    uint256 amountAvailable,
    uint256 currentDebt,
    bool closeAllDebt
)
    external
    view
    returns (uint256 repayAmount, bool useFullRepaySentinel)
{
    ...
    // Otherwise compute the largest safe partial repayment that will not violate
    ↪  Curvance's
    // minimum remaining loan-size rule.
    return (_calculateSafeRepayment(amountAvailable, currentDebt), false);
}
```

However actually it skips repay entirely instead of computing largest safe amount:

```
function _calculateSafeRepayment(uint256 amountAvailable, uint256 currentDebt)
↪  internal view returns (uint256) {
    ...
    // If the residual debt would become too small, skip the partial repayment
    ↪  entirely and
    // wait for a later close-all or larger repayment opportunity.
    return 0;
}
```

Code Snippet

https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/blob/b840577fc767f161c09450435c695c94294125ef/smart-vault-contracts/src/adapters/curvance/CurvanceDebtAdapter.sol#L227-L229 https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/blob/b840577fc767f161c09450435c695c94294125ef/smart-vault-contracts/src/adapters/curvance/CurvanceDebtAdapter.sol#L283-L285

Tool Used

Manual Review

Recommendation

Either update code or comment

Discussion

haha-mike

Related issue in smart-vault-contracts

This issue is related to **Octane audit issue #88 — Rounding-down USD-to-token conversion in CurvanceDebtAdapter._calculateSafeRepayment during partial repay causes deallocation/rebalance batch reverts**.

Why they are related

Both issues are about the same function: `CurvanceDebtAdapter._calculateSafeRepayment()`.

- This Sherlock report highlights the misleading comment: `resolveRepayInstruction` documents that it computes "the largest safe partial repayment", but in the edge case where residual debt would fall below the minimum loan size, `_calculateSafeRepayment` returns `0` (skips entirely) rather than computing any partial amount.
- Octane #88 focuses on the rounding mechanism that triggers this skip: the USD-to-token conversion rounds down, creating a dust residual that falls below `MIN_LOAN_SIZE` and causes the function to return 0 – which then causes the calling batch to revert.

These two reports together fully describe the bug: the rounding behaviour (Octane #88) is the trigger; the silent skip instead of a safe partial repayment (this report) is the unexpected outcome. A fix for Octane #88 (PR #106, branch `fix/curvance-debt-adapter`) is open and should also resolve the misleading behaviour described here.

haha-mike

Fixed in octane findings. Check previous comment with links

Issue L-9: Several adapter functions are not used [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/issues/39

Vulnerability Detail

Functions `enableCollateral()` and `disableCollateral()` can be called by Strategy, but Strategy never calls it, other functions have empty body.

Code Snippet

https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/blob/b840577fc767f161c09450435c695c94294125ef/smart-vault-contracts/src/adapters/curvance/CurvanceCollateralAdapter.sol#L129-L139 https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/blob/b840577fc767f161c09450435c695c94294125ef/smart-vault-contracts/src/adapters/curvance/CurvanceDebtAdapter.sol#L123-L125 https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/blob/b840577fc767f161c09450435c695c94294125ef/smart-vault-contracts/src/adapters/euler/EulerCollateralAdapter.sol#L149-L165 https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/blob/b840577fc767f161c09450435c695c94294125ef/smart-vault-contracts/src/adapters/euler/EulerDebtAdapter.sol#L112-L114

Tool Used

Manual Review

Recommendation

Consider removing those functions

Discussion

haha-mike

Wont fix. We decided to keep them for service purposes.

Issue L-10: Anybody can break the purpose of calling `VaultFeeAllocator.distributeFeesAmount()` [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/issues/40

Summary

There is alternative function to distribute all funds without access control, it breaks the purpose of admin access function.

Vulnerability Detail

`distributeFeesAmount()` is used to stream rewards gradually:

```
    /**
     * @notice Distribute a caller-selected portion of the unreserved token balance.
@>   * @dev This owner-only path is useful when fees should be streamed out
 ↪  gradually instead of all at once.
     * @param token Token to distribute.
     * @param amount Amount to distribute.
     */
    function distributeFeesAmount(address token, uint256 amount) external
     ↪  nonReentrant onlyOwner {
```

However anybody can call `distributeFees()` and distribute available balance. For example anybody can frontrun `distributeFeesAmount()` call, which makes `distributeFeesAmount()` useless.

Impact

Anybody can break the purpose of calling `VaultFeeAllocator.distributeFeesAmount()`

Code Snippet

https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/blob/b840577fc767f161c09450435c695c94294125ef/smart-vault-contracts/src/fee/VaultFeeAllocator.sol#L281-L286

Tool Used

Manual Review

Recommendation

Consider adding access control to `distributeFees()`

Discussion

haha-mike

Wrong finding.

```
\#\# Related issue in smart-vault-contracts

This issue is related to **[Octane audit issue \#91 - Lack of
 ↪  partial-claim/redirect/rescue for failed payouts in VaultFeeAllocator causes
 ↪  frozen fees with restrictive
 ↪  ERC20s](https://github.com/Permutize/smart-vault-contracts/issues/91)**.

\#\#\# Why they are related

Both issues concern the fee distribution design of `VaultFeeAllocator` and the lack
 ↪  of controls over how and when fees flow out.

- Octane \#91 identifies that a single failing transfer (e.g. due to a
 ↪  blocklist-based ERC20) causes the entire fee distribution to revert with no
 ↪  rescue path - fees become permanently frozen.
- This Sherlock report identifies that `distributeFees()` is permissionless, so
 ↪  anyone can front-run `distributeFeesAmount()` and force a full distribution,
 ↪  defeating the owner-controlled gradual streaming mechanism entirely.

These are complementary weaknesses in the same contract: one blocks distribution
 ↪  entirely (frozen fees), the other removes all control over distribution timing
 ↪  (forced full flush). Both are addressed by the open PR \#104
 ↪  (`fix/fee-allocator-pending-fees`) which introduces chunked claims and a
 ↪  redirect/rescue mechanism - however the PR should also be reviewed to ensure
 ↪  `distributeFees()` access control is tightened as part of the same fix.
```

haha-mike

Wont fix. Made by design.

Issue L-11: `VaultFeeAllocator` distributes fees retrospectively after partner update [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/issues/41

Summary

`VaultFeeAllocator` doesn't distribute fees before partner updates.

Vulnerability Detail

Functions `updatePartner()`, `addPartner()`, `removePartner()` doesn't distribute fees before update.

Impact

It distributes fees retrospectively. For example admin can remove partner and only then distribute fees. This way partner doesn't receive fees which were earned during time he was a partner.

Code Snippet

https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/blob/b840577fc767f161c09450435c695c94294125ef/smart-vault-contracts/src/fee/VaultFeeAllocator.sol#L178-L180 https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/blob/b840577fc767f161c09450435c695c94294125ef/smart-vault-contracts/src/fee/VaultFeeAllocator.sol#L197-L199 https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/blob/b840577fc767f161c09450435c695c94294125ef/smart-vault-contracts/src/fee/VaultFeeAllocator.sol#L223-L225

Tool Used

Manual Review

Recommendation

Consider distributing fees before partner updates, such as: `updatePartner()`, `addPartner()`, `removePartner()`.

Discussion

haha-mike

Related issue in smart-vault-contracts

This issue is related to **Octane audit issue #91 — Lack of partial-claim/redirect/rescue for failed payouts in VaultFeeAllocator causes frozen fees with restrictive ERC20s**.

Why they are related

Both issues stem from design gaps in `VaultFeeAllocator`'s fee distribution lifecycle.

- Octane #91 identifies that failed transfers (restrictive ERC20s) have no recovery path – fees freeze permanently.
- This Sherlock report identifies that partner configuration changes (`addPartner`, `updatePartner`, `removePartner`) do not trigger a fee distribution snapshot first. A partner can be removed and then fees distributed, causing the partner to lose fees they legitimately earned during their active period.

These issues compound each other: partners who cannot receive funds due to restrictive ERC20s (Octane #91) and partners whose earned fees are wiped by a configuration change before distribution (this report) both result in fee loss with no recovery. The open PR #104 (`fix/fee-allocator-pending-fees`, branch `fix/fee-allocator-pending-fees`) should be reviewed to ensure it also enforces a distribution flush before partner mutations.

haha-mike

Acknowledged. @0xdavid7 is working on this.

T1MOH593

@0xdavid7 Works fine if there is only 1 fee token configured. In case you expect multiple tokens, it's better to submit array of tokens

0xdavid7

The reason behind this is that we only collect the vault token as the fee token.

Issue L-12: `withdrawToOwner` allows transferring the core vault asset [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/issues/45

Summary

`withdrawToOwner` allows transferring the core vault asset to the strategy admin due to a missing validation.

Vulnerability Detail

The function does not check whether `asset_` equals the strategy's core asset (`asset()`), unlike `rescueToken` which blocks this case. Because it is callable via batch execution, a crafted command can withdraw user funds.

Impact

- Unauthorized withdrawal of vault funds
- Potential loss of user assets
- Breaks expected fund isolation

Code Snippet

```
function withdrawToOwner(bytes memory data) external onlySelf {
    (address asset_, uint256 amount) = abi.decode(data, (address, uint256));
    address admin = _getBaseStrategyStorage().strategyAdmin;
    if (admin == address(0)) revert InvalidParams();
    IERC20(asset_).safeTransfer(admin, amount);
}
```

Tool Used

Manual Review

Recommendation

Add a check to block the core asset:

```
if (asset_ == this.asset()) revert CannotRescueCoreAsset();
```

Discussion

haha-mike

Acknowledged. @haha-mike please remove this function

Issue L-13: `BaseStrategy.rescueToken()` can break accounting [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/issues/47

Summary

Only AUSD token is treated as core asset, while actually AUSDStrategy has 4 core assets: MON, shMON, wMON, AUSD.

Vulnerability Detail

`rescueToken()` transfers token to admin, it block transferring AUSD not to break accounting:

```
    function rescueToken(address token, uint256 amount) external virtual
    ↪  onlyStrategyAdmin {
@>      if (token == this.asset()) {
            revert CannotRescueCoreAsset();
        }

        uint256 bal = IERC20(token).balanceOf(address(this));
        uint256 toSend = amount == 0 ? bal : amount;

        address admin = _getBaseStrategyStorage().strategyAdmin;
        if (admin == address(0)) revert InvalidParams();
        IERC20(token).safeTransfer(admin, toSend);
    }
```

Problem is that there are 2 more core assets in AUSDStrategy: wMON and shMON. They are accounted in `totalAllocatedValue()` function which is used for yield calculation.

```
    function _calculateTotalCollateralUsd() private view returns (uint256 total) {
        bytes memory primaryColData = abi.encodeWithSelector(
            ICollateralAdapter.getCollateralBalance.selector, address(STABLECOIN),
            ↪  address(this)
        );
        ...

@>      total += _shMonToUsd(SHMON.balanceOf(address(this)));
@>      total += _wmonToUsd(WMON.balanceOf(address(this)));
        total += _wmonToUsd(address(this).balance);
        total += _stablecoinToUsd(STABLECOIN.balanceOf(address(this)));

        ...
```

```
    }
```

Impact

If rescued token is `wMON` or `shMON`, it will be reported as negative yield, which means depositors lose money.

Code Snippet

https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/blob/main/smart-vault-contracts/src/strategies/BaseStrategy.sol#L1164-L1175

Tool Used

Manual Review

Recommendation

Prevent transferring tokens wMON and shMON.

Discussion

haha-mike

Acknowledged. @haha-mike is working on this. Low priority.

Issue L-14: `borrowCapThresholdBps` is never checked [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/issues/51

Summary

`BaseStrategy` exposes a `borrowCapThresholdBps` limit, but no code path enforces it on borrow operations.

Vulnerability Detail

`borrowCapThresholdBps` initialized and surfaced via `setBorrowCapThresholds`, suggesting an intended cap on borrowing. However, it is never actually checked.

Impact

The bps threshold is completely bypassed, allowing borrowing up to the protocol's own LTV cap regardless of the configured threshold.

Code Snippet

https://github.com/sherlock-audit/2026-04-permutize-apr-6th-2026/blob/b840577fc767f161c09450435c695c94294125ef/smart-vault-contracts/src/strategies/BaseStrategy.sol#L174

Tool Used

Manual Review

Recommendation

Either enforce the cap on every borrow execution or remove the storage and the getter entirely.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.