SHERLOCK

Security Review For Sixpence

Collaborative Audit Prepared For: **Sixpence**
Lead Security Expert(s): **blockace**
TessKimy
Date Audited: **May 1 - May 8, 2026**

Introduction

Sixpence is an on-chain money market built for tokenized equities. You can supply tokenized stocks and ETFs as collateral, borrow USDC against them at transparent rates, or deposit stablecoins into curated vaults to earn yield from borrow demand.

Scope

Repository: 0xJomo/sixpence-market-deploy-audit

Audited Commit: d6a2d4bc8bda63607aa45e1c33f02d0591162a6e

Final Commit: 20a28871e491de642f615618a27bf512b7dac4ed

Files:

- contracts/adapters/ERC4626OracleAdapter.sol
- contracts/adapters/UniswapV2Swapper.sol
- contracts/adapters/UniversalSwapAndRepayAdapter.sol
- contracts/LiquidationBotV3.sol

Repository: 0xJomo/sixpence-vault-audit

Audited Commit: e179ede54d9d19c0742a0060337bced768eb299e

Final Commit: b8a6d406fc95b59985b098f89a3642e853c5e99e

Files:

- contracts/adapters/Lending/AaveV3Adapter.sol
- contracts/adapters/Yield/ERC4626YieldAdapter.sol
- contracts/libraries/VaultCollateralLib.sol
- contracts/SixpenceVaultBundler.sol
- contracts/SixpenceVaultFactory.sol
- contracts/VaultRegistry.sol

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
6	4	3

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue H-1: `yieldToken` as collateral rail breaks `reconcileTokenState` bookkeeping [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-sixpence-follow-up-apr-30th-2026/issues/12

Summary

If the yield vault share token (`getYieldToken()`) is registered like normal collateral and users `depositToken` it, `reconcileTokenState` mixes one Aave balance with two different internal ledgers ("all supplied" vs "loop-only yield collateral"). Numbers for shrink, scales, and borrowed haircuts can be wrong.

Vulnerability detail

- On Aave, the vault holds one combined `yieldToken` supply (user deposits + optional loop re-supply).
- `totalYieldCollateralSupplied[token]` only tracks the loop leg.
- In `reconcileTokenState`, when `p.token == yieldToken`, `liveYieldSupplied` is the full chain balance but it is compared / merged with `totalYieldCollateralSupplied[p.token]`, which is only a subset – so reconciliation logic is inconsistent.
- That can skew `totalYieldShares`, `yieldCollateralScale`, and `effectiveRatioBps` (debt haircut). `executeEmergencyExit` still pulls `yieldToken` from pools based on `collaterals[]`; corrupted prior accounting makes behavior harder to trust.

Impact

Mis-reconciliation and unfair or incorrect withdraw / repay / paused flows if `yieldToken` is both collateral-configured and deposited.

Code snippet

https://github.com/sherlock-audit/2026-04-sixpence-follow-up-apr-30th-2026/blob/main/sixpence-vault-audit/contracts/libraries/VaultCollateralLib.sol#L783-L834

Tool used

Manual Review

Recommendation

`_depositTokenInternal`: `revert` if `token == IYieldAdapter(yieldAdapter).getYieldToken()` (collateral path).

Issue H-2: `reconcileTokenState` updates aggregate `tokenStates.totalSupplied` but not `userTokenStates[][].scaledSupplied` [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-sixpence-follow-up-apr-30th-2026/issues/13

Summary

`VaultCollateralLib.reconcileTokenState` refreshes `tokenStates[p.token].totalSupplied` from live lending balances. It does not adjust each user's `userTokenStates[user][p.token].scaledSupplied`. `withdrawPausedCollateral` (and `withdrawCollateralDirect`) size withdrawals from `userState.scaledSupplied`, which can stay stale after a live shrink (reconciliation, liquidation, donation, rounding drift, etc.). Aggregate and per-user ledgers no longer match user total, so paused exits can withdraw the wrong amount, revert on Aave, leave residual or unfair claims between users, or rely on ad hoc clamps (e.g. `tokenState.totalSupplied` capped to 0 while `userState` is still decremented).

Vulnerability Detail

1. `reconcileTokenState` only writes `state.totalSupplied` (and borrowed / yield fields) from `getSuppliedBalance` → `getScaledAmount`; there is no loop over `userTokenStates`.
2. Deposits set both sides in sync (e.g. `SixpenceVaultBundler` increments `tokenStates[token].totalSupplied` and `userTokenStates[..].scaledSupplied` with the same scaled delta).
3. After any live change that triggers reconcile to lower `tokenStates.totalSupplied`, `userTokenStates[..].scaledSupplied` are unchanged, so `scaledSupplied` (per user) can exceed the updated global `totalSupplied`.
4. `withdrawPausedCollateral` computes

   ```
   scaledAmount = (userState.scaledSupplied * p.shares) / p.totalUserShares
   ```

   then `getRealAmount` and pool withdraw. Unlike `withdrawCollateralDirect`, there is no step that caps the user's withdrawn real amount against reconciled `tokenState.totalSupplied` expressed in real units (the active path's proportional limit when `sharesPreDecremented` is not present here).
5. At the end `userState.scaledSupplied -= scaledToDeduct` and `tokenState.totalSupplied -= scaledToDeduct`, with `if (scaledToDeduct > tokenState.totalSupplied) tokenState.totalSupplied = 0`. That avoids one underflow but does not realign all users' `scaledSupplied` to the reconciled total, so later users or invariant "sum of books = totalSupplied" can break.

Impact

When the vault is PAUSED / legacy and users exit via `withdrawPausedCollateral`, stale `scaledSupplied` after reconcile can cause over- or under-payment relative to on-chain collateral, failed withdrawals, unfair ordering (first redeemers vs last), and persistent accounting drift between `tokenStates` and `userTokenStates`.

Code Snippet

https://github.com/sherlock-audit/2026-04-sixpence-follow-up-apr-30th-2026/blob/main/sixpence-vault-audit/contracts/libraries/VaultCollateralLib.sol#L783-L794

https://github.com/sherlock-audit/2026-04-sixpence-follow-up-apr-30th-2026/blob/main/sixpence-vault-audit/contracts/libraries/VaultCollateralLib.sol#L393-L405

Tool Used

Manual Review

Recommendation

Update each `userTokenStates[user][token].scaledSupplied` using logic similar to `_syncUserYieldCollateralState`.

Issue H-3: `reconcileTokenState` haircuts `tokenStates.totalBorrowed` but not `userTokenStates[][].scaledBorrowed` [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-sixpence-follow-up-apr-30th-2026/issues/14

Summary

In `reconcileTokenState` function, `tokenStates[].totalBorrowed` is updated but per-user `userTokenStates[user][token].scaledBorrowed` is never scaled the same way. Redeem and debt repayment both derive proportional debt from `userState.scaledBorrowed`, while the global ledger says less debt. That desync can cause wrong profit/debt splits, unfair ordering between users, inconsistent `totalBorrowed` decrements during `processDebtRepayment`, and violations of the intended invariant that aggregate user debt tracks aggregate `tokenStates.totalBorrowed` after reconciliation.

Vulnerability Detail

1. Global haircut only - `reconcileTokenState` function updates only `state.totalBorrowed`:

There is no corresponding pass over `userTokenStates` to multiply each `scaledBorrowed` by `effectiveRatioBps / 10_000` (or an equivalent rebasing).

2. Deposits keep the two in sync - On borrow, the bundler increments `tokenStates[token].totalBorrowed` and `userTokenStates[..].scaledBorrowed` with the same scaled borrow amount (`SixpenceVaultBundler` around the leverage loop). After reconcile, that equality need not hold.
3. Redeem (paused / non–debt path) - `_redeemPausedNonDebtToken` uses `userState.scaledBorrowed` to size `scaledDebtPortion` and thus `actualDebtPortion` for profit and stablecoin transfer logic, independent of the post-reconcile `tokenStates[p.token].totalBorrowed`:

If `scaledBorrowed` was never reduced when `totalBorrowed` was, users can be charged against a larger book debt than the protocol's global haircut implies, or symmetrically skew who absorbs implied loss depending on redeem order.

4. Repayment - `processDebtRepayment` computes `scaledDebtProportional` from `userState.scaledBorrowed`, compares to pool debt, repays actualDebt, then does `userState.scaledBorrowed -= scaledDebt` and `tokenState.totalBorrowed -= scaledDebt` with a clamp:

With stale high per-user `scaledBorrowed` and already reduced `tokenState.totalBorrowed`, repayments can zero out aggregate `totalBorrowed` while users still carry non-zero `scaled`

`Borrowed`, or leave inconsistent remaining debt across users. `previewUserYield` also keys off `userState.scaledBorrowed`, so previews diverge from reconciled global state.

Impact

Effects include incorrect redeem economics, wrong repay sizing / preview, first-mover vs last-mover distortion, and broken "sum of user debt totalBorrowed" reasoning for integrations and internal checks.

Code Snippet

https://github.com/sherlock-audit/2026-04-sixpence-follow-up-apr-30th-2026/blob/main/sixpence-vault-audit/contracts/libraries/VaultCollateralLib.sol#L870-L876

https://github.com/sherlock-audit/2026-04-sixpence-follow-up-apr-30th-2026/blob/main/sixpence-vault-audit/contracts/libraries/VaultCollateralLib.sol#L634-L635

https://github.com/sherlock-audit/2026-04-sixpence-follow-up-apr-30th-2026/blob/main/sixpence-vault-audit/contracts/libraries/VaultCollateralLib.sol#L891-L892

Tool Used

Manual Review

Recommendation

Update each `userTokenStates[user][token].scaledBorrowed` using logic similar to _syncUserYieldCollateralState.

Issue H-4: `_reconcileTokenState` Is Not Called on Deposit, Allowing Stale State to Absorb New Depositors [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-sixpence-follow-up-apr-30th-2026/issues/18

Summary

`_reconcileTokenState` is only triggered inside `redeemToken`. A deposit executed after a partial liquidation (which reduces `scaledSupplied` and `totalYieldCollateralSupplied` in reality but not in storage) will credit the new depositor with shares calculated against stale, inflated supply figures. When that depositor later redeems, the reconcile runs and reveals the true (lower) supply, diluting their redemption relative to what they were promised at deposit time.

Root Cause

`redeemToken` calls `_reconcileTokenState` to sync live lending-protocol balances before computing the user's share. `deposit` has no equivalent call. If a liquidation occurs between two deposits, the first post-liquidation depositor receives shares priced against pre-liquidation supply. At redemption time, reconcile is finally applied, retroactively shrinking the supply and reducing the depositor's proceeds without their knowledge or consent.

Impact

A user who deposits after a silent liquidation (no on-chain event they can observe) will receive fewer tokens at redemption than the share price implied at deposit time. The loss is absorbed entirely by the post-liquidation depositor rather than being shared across all stakeholders or properly disclosed. This constitutes a material misrepresentation of the vault's state at deposit time.

Attack Path

1. Vault is liquidated: real `liveYieldSupplied` drops from 10,000 to 6,000, but storage still shows `totalYieldCollateralSupplied = 10,000`.
2. No redemption has been made, so `_reconcileTokenState` has never been called since the liquidation.
3. User deposits 1,000 tokens. Their shares are minted using stale supply = 10,000; they receive shares reflecting a 9.09% stake.

4. User immediately redeems. `_reconcileTokenState` now runs, correcting supply to 6,000. Their 9.09% stake is now worth ~545 tokens – a 45% loss on a same-block deposit/redeem.

Recommendation

Call `_reconcileTokenState` at the start of every state-mutating entry point, including `deposit`, not only `redeemToken`. This ensures all share calculations – both issuance and redemption – operate on live lending-protocol balances.

Discussion

daring5920

The 45% loss scenario described in this report cannot occur. previewDeposit uses strategy.totalAssets and redeemableAmount uses tokenStates[token].totalAssets – neither of these fields is ever modified by _reconcileTokenState. The share pricing is therefore completely decoupled from the supply figures that reconcile updates.

The underlying concern about the proportional collateral withdrawal cap in _withdrawCollateralDirect (which does reference tokenState.totalSupplied) is legitimate, but adding a reconcile call to every deposit would not fix it. Even with a deposit-time reconcile, totalShares – the denominator of the user's share ratio – is unchanged, so the cap computes to the same value. The correct fix is the collateralScale / userCollateralScale mechanism in Issue #13, which proportionally adjusts existing users' scaledSupplied when a liquidation shrink is detected.

We consider this finding Invalid / Duplicate of Issue #13. No code change is required for this specific report.

DemoreXTess

This finding has no relation with share pricing. While solving issue #13, you need to initiate new user based scale variable and whenever if user scale is higher than global scale, user's scaled balance will be updated, however if you only add that reconcile and sync pattern into redeem function, new users will be affected.

This finding is just express that we need to add some scale handling system into deposit function. That's why code change will be required for this issue.

Issue H-5: Legacy Epoch Collateral Remains Exposed in Aave After Epoch Transition, Lost to Subsequent Liquidations [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-sixpence-follow-up-apr-30th-2026/issues/22

Summary

When an emergency exit concludes and `adminResume` starts a new epoch, the collateral tokens deposited by previous-epoch users are never withdrawn from the Aave lending pool. The new epoch then borrows in the same Aave account where that legacy collateral still lives. If the new epoch's position is liquidated, Aave seizes from the entire shared position it has no concept of epoch boundaries and legacy collateral is consumed. When those legacy users later call `redeemToken`, the paused-redemption path attempts to withdraw their collateral from Aave live, but it is no longer there.

Root Cause

`executeEmergencyExit` only repays on-chain debt and redeems yield-vault shares; it never calls `withdrawCollateral` for any epoch's underlying token collateral. `adminResume` resets accounting fields (`totalBorrowed`, `totalShares`, etc.) but explicitly leaves `totalSupplied` untouched and makes no attempt to withdraw or isolate legacy collateral. The new epoch therefore inherits an Aave account that contains both the fresh epoch's collateral and all unsettled prior-epoch collateral a fully shared, undifferentiated position. A liquidation during any subsequent epoch seizes from this whole pool.

Impact

Legacy epoch users can lose their entire collateral to a liquidation they had no part in causing. The loss is not distributed proportionally; whichever legacy user redeems first after an emergency exit drains whatever Aave balance remains, and later redeemers receive nothing. There is no on-chain mechanism to detect or compensate for this shortfall, so the vault silently becomes insolvent for a subset of users.

Attack Path

1. Epoch 1 suffers an adverse event. `adminEmergencyExit` is called: all epoch 1 debt is repaid via `executeEmergencyExit`, yield shares are redeemed, vault transitions to `PAUSED`. Epoch 1 users' collateral tokens (e.g. WETH) remain in Aave – they are never withdrawn.

2. `adminResume` is called: epoch counter increments to 2, accounting fields are reset, vault goes `ACTIVE`. Epoch 1 WETH collateral is still sitting in the same Aave account.
3. Market deteriorates during epoch 2. The vault's aggregate health factor (epoch 1 + epoch 2 collateral, epoch 2 debt) falls below 1.0. Aave liquidates the position, seizing from the combined pool – including epoch 1 WETH.
4. Epoch 1 users call `redeemToken`. The code routes to `_redeemPaused` → `_redeemPausedNonDebtToken` → `withdrawPausedCollateral`, which queries Aave. First redeemers exhaust whatever scraps remain; remaining users receive zero collateral.

Recommendation

When closing an epoch (either in `adminEmergencyExit` or at the latest in `adminResume` before starting the new epoch), withdraw all outstanding collateral for the closing epoch from Aave into the contract itself. Once collateral is held in the contract it is fully isolated from any future epoch's Aave position and cannot be seized by a subsequent liquidation. Legacy-epoch redemptions would then operate against an in-contract balance rather than a live Aave position.

Issue H-6: Incorrect reconciling causes underestimates the total debt [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-sixpence-follow-up-apr-30th-2026/issues/24

Summary

When `reconcileTokenState` detects that on-chain collateral has shrunk (base collateral or yield-token sub-leg), it haircuts `state.totalBorrowed` by `effectiveRatioBps = min(collateralRatioBps, yieldRatioBps)` – that is, it estimates the new debt by applying the *collateral* loss percentage to the *debt* book. The actual debt change after a liquidation, partial liquidation, or external repay is governed by the Aave liquidation-bonus math and the pool's reserve indices; it is in general unrelated to the collateral loss percentage. As a result the book debt diverges from the on-chain debt on every reconcile that fires.

Vulnerability Details

The relevant block, after the two ratio computations:

```
// sixpence-vault-audit/contracts/libraries/VaultCollateralLib.sol:949-959
uint256 effectiveRatioBps = collateralRatioBps < yieldRatioBps
    ? collateralRatioBps
    : yieldRatioBps;

result.borrowedAfter = result.borrowedBefore;
if (effectiveRatioBps < 10_000) {
    result.borrowedAfter =
        (result.borrowedBefore * effectiveRatioBps) /
        10_000;
    state.totalBorrowed = result.borrowedAfter;
}
```

The same value also seeds the lazy-haircut scale in the bundler:

```
// sixpence-vault-audit/contracts/SixpenceVaultBundler.sol:1110-1119
if (
    reconcileResult.borrowedBefore > 0 &&
    reconcileResult.borrowedAfter < reconcileResult.borrowedBefore
) {
    uint256 oldScale = borrowScale[token];
    if (oldScale == 0) oldScale = RAY;
    borrowScale[token] =
        (oldScale * reconcileResult.borrowedAfter) /
        reconcileResult.borrowedBefore;
```

```
}
```

So an incorrect `borrowedAfter` poisons three places at once: the aggregate `state.totalBorrowed`, the global `borrowScale[token]`, and (via the next `_syncUserBorrowState` touch) every user's `userTokenStates[user][token].scaledBorrowed`.

The estimator equates "fraction of collateral remaining" with "fraction of debt remaining".

Case — typical Aave liquidation. Aave seizes `D · (1 + bonus)` worth of collateral for `D` of debt repaid. With a 5% bonus and collateral originally worth `C`, debt originally `D`, after liquidation:

- `collateralRatio = (C − D·1.05) / C`
- `debtRatio_actual = (D − D) / D`

`debtRatio_actual < collateralRatio` (debt drops a larger fraction than actual). The estimator uses `collateralRatio`, which is *higher* than the true ratio, so:

```
state.totalBorrowed = before · collateralRatio  >  before · debtRatio_actual = real
↪  on-chain debt
```

Impact

`state.totalBorrowed` and `borrowScale[token]` diverge from the Aave `getDebtBalance(...)` whenever collateral or yield-collateral shrinks. The divergence is unbounded (it compounds across multiple reconciles).

Recommendation

Replace the collateral-ratio-derived `borrowedAfter` with a direct on-chain debt read, mirroring the way `suppliedAfter` is computed (live balance → `getScaledAmount`). This removes the proxy entirely. The yield-collateral haircut stays (it correctly tracks the yield sub-leg by its own live balance) but is decoupled from the debt update.

```
function reconcileTokenState(
    ReconcileTokenStateParams memory p,
    mapping(address => IVaultStructs.TokenState) storage tokenStates,
    mapping(address => IVaultStructs.TokenConfig) storage tokenConfigs,
    mapping(address => uint256) storage totalYieldCollateralSupplied,
    mapping(address => uint256) storage yieldCollateralScale
) external returns (ReconcileTokenStateResult memory result) {
    IVaultStructs.TokenState storage state = tokenStates[p.token];
    if (state.totalShares == 0) {
        return result;
    }
```

```
    // 1. Record pre-reconcile snapshots.
    result.suppliedBefore = state.totalSupplied;
    result.borrowedBefore = state.totalBorrowed;

    // 2. Fetch live scaled balances from the lending protocol and persist.
    result.suppliedAfter = ILendingAdapter(p.lendingAdapter).getScaledAmount(
        ILendingAdapter(p.lendingAdapter).getSuppliedBalance(p.token,
        ↪  address(this), p.pool),
        p.token,
        p.pool,
        false
    );
    state.totalSupplied = result.suppliedAfter;

    result.borrowedAfter = ILendingAdapter(p.lendingAdapter).getScaledAmount(
        ILendingAdapter(p.lendingAdapter).getDebtBalance(p.debtAsset,
        ↪  address(this), p.pool),
        p.debtAsset,
        p.pool,
        true
    );
    state.totalBorrowed = result.borrowedAfter;
    // 3. Update yield collateral scale if the live scaled balance has decreased.
    address yieldToken = IYieldAdapter(p.yieldAdapter).getYieldToken();
    IVaultStructs.TokenConfig memory yieldCfg = tokenConfigs[p.token];
    uint256 totalYieldCollateralBefore = totalYieldCollateralSupplied[p.token];

    if (
        tokenConfigs[yieldToken].isCollateral &&
        yieldCfg.pool != address(0) &&
        totalYieldCollateralBefore > 0
    ) {
        uint256 liveYieldSuppliedScaled =
        ↪  ILendingAdapter(p.lendingAdapter).getScaledAmount(
            ILendingAdapter(p.lendingAdapter).getSuppliedBalance(yieldToken,
            ↪  address(this), yieldCfg.pool),
            yieldToken,
            yieldCfg.pool,
            false
        );
        if (liveYieldSuppliedScaled < totalYieldCollateralBefore) {
            totalYieldCollateralSupplied[p.token] = liveYieldSuppliedScaled;

            uint256 globalScaleBefore =
            ↪  _getYieldCollateralScale(yieldCollateralScale, p.token);
            yieldCollateralScale[p.token] =
                (globalScaleBefore * liveYieldSuppliedScaled) /
                totalYieldCollateralBefore;
            state.totalYieldShares =
                (state.totalYieldShares * liveYieldSuppliedScaled) /
```

```
                totalYieldCollateralBefore;
        }
    }
}
```

Issue M-1: Global `previewDeposit` mints pathological share inflation when `strategy.totalAssets` is near zero while legacy `totalSupply()` remains large [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-sixpence-follow-up-apr-30th-2026/issues/11

Summary

Vault share minting uses `Math.mulDiv(assets, totalSupply() + SHARE_DECIMALS_OFFSET, totalAssets() + 1)`. For USDC-style principal deposits, `previewDeposit` runs before `strategy.totalAssets` is increased. If `strategy.totalAssets()` is `0` (or tiny) but `totalSupply()` is still large because outstanding share from prior epochs was never burned, the denominator `totalAssets() + 1` is `1`, so a single small deposit can mint `assets × (totalSupply + offset)` shares - orders of magnitude larger than a "normal" ERC4626 epoch. That can repeat after further stress if the same `totalAssets  0` precondition reappears while supply stays huge, breaking economic expectations and risking **reverts** or **DoS** under extreme sizes (Solidity 0.8+ checks), not merely rounding dust.

Vulnerability Detail

1. **Mint path** - `previewDeposit` couples new shares to global `ERC20.totalSupply()` and global `strategy.totalAssets`:
2. **Order of operations** - On debt-asset deposit, `previewDeposit(assets)` executes before `strategy.totalAssets += assets`, while `totalSupply()` still includes all legacy holders until `_burn`:
3. **Broken state** - After `adminEmergencyExit` / `adminResume`, design comments require `strategy.totalAssets` to keep backing legacy shares while `totalSupply()` still counts them; alternatively, a drained `strategy.totalAssets` with non-zero `totalSupply()` is achievable in loss scenarios. In the drained case, `totalAssets() + 1 = 1`:
4. **Numerical example** - With `totalSupply() = 1e14`, `SHARE_DECIMALS_OFFSET = 1e6`, `assets = 1e8` (100 USDC, 6 decimals), `strategy.totalAssets() = 0`: `1e8 * (1e14 + 1e6) / 1  1e22`

So `totalSupply` can jump from `1e14` to `~1e22` in one deposit. Repeating emergencies that again leave `strategy.totalAssets  0` while supply remains enormous can compound minted share counts.

5. **Legacy vs global** - Legacy redemption can use `epochStates[userEpoch]` in some `redeemToken` branches; that does not remove ERC20 from `totalSupply()` until burn. So

”epoch isolation” on exit pricing does not remove legacy supply from the global mint ratio.

Impact

Reverts or **DoS** under extreme sizes

Code Snippet

https://github.com/sherlock-audit/2026-04-sixpence-follow-up-apr-30th-2026/blob/main/sixpence-vault-audit/contracts/SixpenceVaultBundler.sol#L246-L253

Tool Used

Manual Review

Recommendation

Use `epochShareSupply[currentEpoch]` instead of `totalSupply()` in `previewDeposit` function.

Issue M-2: Partial legacy redemptions can desynchronize user accounting in `withdrawPausedCollateral` [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-sixpence-follow-up-apr-30th-2026/issues/16

Summary

`VaultCollateralLib.withdrawPausedCollateral` function first calculates a pro-rata share amount (`scaledAmount`) from user shares, but later reduces user balances using a different value (`scaledToDeduct`) derived from the actual withdrawn amount. Because these two values can differ during capped withdrawals, repeated partial redemptions may create accounting drift in `userState.scaledSupplied` and `userState.tokenAmount`.

Vulnerability Detail

At the start of the flow, the user's proportional position is computed as. However, if `realAmount` is reduced (for example due to `maxWithdraw` limits), the function recalculates the deduction:

```
uint256 scaledAmount = (userState.scaledSupplied * p.shares) / p.totalUserShares;
uint256 realAmountFull = ILendingAdapter(p.lendingAdapter).getRealAmount(
    scaledAmount,
    p.token,
    p.pool,
    false
);

uint256 realAmount = realAmountFull;
if (p.hasDebt) {
    uint256 maxWithdraw = ILendingAdapter(p.lendingAdapter)
        .getMaxWithdrawAmount(p.token, p.pool, address(this), p.oracle);
    if (realAmount > maxWithdraw) {
        realAmount = maxWithdraw;
    }
}

if (realAmount > 0) {

    uint256 scaledToDeduct = scaledAmount;
    uint256 tokenAmount = (userState.tokenAmount * p.shares) / p.totalUserShares;
    uint256 tokenToDeduct = tokenAmount;

    if (realAmount < realAmountFull) {
```

```
        scaledToDeduct =
        ↪  ILendingAdapter(p.lendingAdapter).getScaledAmount(realAmount, p.token,
        ↪  p.pool, false);
        if (scaledToDeduct > scaledAmount) {
            scaledToDeduct = scaledAmount;
        }
        tokenToDeduct = (tokenAmount * realAmount) / realAmountFull;
    }
```

Then it subtracts `scaledToDeduct` and `tokenToDeduct` from user state, not the original proportional `scaledAmount/tokenAmount`.
This means user balances are no longer reduced strictly by share proportion, which can introduce inconsistencies over multiple partial withdrawals.

Impact

This mismatch can cause state drift between share-based entitlement and stored user balances, especially if a user performs multiple partial redemptions under constrained withdrawal conditions. Over time, this may lead to unfair outcomes between users and harder-to-reason-about redemption accounting.

Code Snippet

https://github.com/sherlock-audit/2026-04-sixpence-follow-up-apr-30th-2026/blob/main/sixpence-vault-audit/contracts/libraries/VaultCollateralLib.sol#L398-L445

Tool Used

Manual Review

Recommendation

Consider allowing only **full redemption** for legacy-epoch users (disallow partial redemption) to simplify state transitions and reduce potential accounting edge-case risk.

Discussion

daring5920

aaveDebt is structurally 0 here because the only path into PAUSED state is adminEmergencyExit(), which repays all pool debt before flipping the status flag. Therefore hasDebt = false, realAmount == realAmountFull, and scaledToDeduct == scaledAmount – the partial-cap accounting drift described in the audit finding cannot occur under normal protocol operation.

blockace256

Agree that `hasDebt` is effectively always false.

If that invariant is guaranteed, I recommend removing the `hasDebt` handling logic entirely, since the related branch is unreachable in practice.

Additionally, even if `hasDebt` remains always false, I still recommend restricting legacy-epoch users to full redemption only (i.e. disallow partial redemption). This would simplify the state transition logic and reduce the risk of accounting edge cases or rounding dust accumulation.

Given that I could not identify any concrete H/M impact yet, the issue severity could reasonably be downgraded to Low/Informational.

If you agree, I will update this report.

DemoreXTess

> Agree that hasDebt is effectively always false.

That's not true, when we resume the protocol `hasDebt` will be true (after new deposits) and legacy epoch may still contain some users who will redeem with this path. However, I think we need to focus on #22, it's also related with the same thing. If we solve #22 as I recommended this issue won't be exist anymore.

blockace256

Thanks, @DemoreXTess. That issue should remain classified as Medium severity. I still recommend restricting redemptions to full redemption only - even though the recommendation for #22 is applied, I think the mismatch still exits. Additionally, there is no clear reason to allow partial redemption, since the total redeemable amount remains the same whether the user performs a single full redemption or multiple partial redemptions.

Issue M-3: Cross-collateral liquidation can shift losses across rails (single Aave account, per-token accounting) [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-04-sixpence-follow-up-apr-30th-2026/issues/17

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

Summary

The vault maintains one Aave V3 position (`address(this)`) across multiple collateral assets, while internal accounting tracks users per collateral token. Aave liquidations are account-level and liquidators can choose which collateral to seize, so losses may be realized on a different collateral than the one that caused the health factor drop.

Vulnerability Detail

When the vault becomes liquidatable, liquidators repay the debt asset and select collateral to receive based on profitability/availability. This can cause a liquidation event to disproportionately reduce the vault's holdings of a particular collateral (e.g., WBTC), while internal accounting may conceptually attribute risk to another rail (e.g., WETH).

Example: Collaterals are **WETH** and **WBTC**, debt is **USDC**. **WETH** drops sharply and the **whole** vault position becomes liquidatable. A liquidator may still **take WBTC** (or mostly WBTC) to cover **USDC** debt if that is more profitable for them than seizing WETH.

Impact

Potential unfair loss allocation across depositor groups: depositors of one collateral rail may bear liquidation losses triggered by price movement in another collateral.

Code Snippet

https://github.com/sherlock-audit/2026-04-sixpence-follow-up-apr-30th-2026/blob/main/sixpence-vault-audit/contracts/SixpenceVaultBundler.sol

Tool Used

Manual Review

Recommendation

Use one vault per collateral token.

Issue M-4: Missing Zero-Amount Guard Before External Withdrawal Call [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-sixpence-follow-up-apr-30th-2026/issues/19

Summary

After the debt shortfall deduction in step 3, `realAmount` can be reduced to exactly 0. The code proceeds unconditionally to build and execute the external call to the lending adapter with `amount = 0`. Most lending protocols (including Aave) revert on a zero-amount withdrawal, causing the entire user withdrawal to revert even though a graceful `return 0` would be the correct behavior.

Root Cause

There is no `if (realAmount == 0) return 0;` guard between the debt shortfall block (step 3) and the external call construction (step 5). The `getWithdrawCall` + `target.call(data)` sequence is unconditional.

Impact

Any user whose collateral is entirely consumed by debt shortfall (a legitimate, expected scenario at the edge of insolvency) cannot complete their withdrawal. The transaction reverts, leaving them unable to exit the position and potentially preventing debt accounting cleanup, which can cascade into a stuck vault state.

Attack Path

1. User has collateral worth 100 USDC and debt worth 100 USDC.
2. `debtShortfall = 100`, `collateralReduction >= realAmount`, so `realAmount = 0`.
3. Code proceeds to call the lending adapter with `withdraw(token, 0, ...)`.
4. Aave reverts with `INVALID_AMOUNT`. The user's withdrawal permanently fails.

Recommendation

Add an explicit `if (realAmount == 0) return 0;` check immediately after the debt shortfall block (step 3) and before constructing the withdrawal call in step 5.

Issue L-1: Redundant unreachable branch in `collateralRatioBps` and `yieldRatioBps` computation [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-sixpence-follow-up-apr-30th-2026/issues/15

Summary

`collateralRatioBps` and `yieldRatioBps` calculation has a redundant branch in `VaultCollateralLib.reconcileTokenState`: when `result.suppliedBefore > 0`, the first condition `result.suppliedAfter < result.suppliedBefore` already covers `result.suppliedAfter == 0`, so the `else if` block is unreachable.

Code snippets

https://github.com/sherlock-audit/2026-04-sixpence-follow-up-apr-30th-2026/blob/main/sixpence-vault-audit/contracts/libraries/VaultCollateralLib.sol#L837-L846

https://github.com/sherlock-audit/2026-04-sixpence-follow-up-apr-30th-2026/blob/main/sixpence-vault-audit/contracts/libraries/VaultCollateralLib.sol#L852-L861

Issue L-2: Precision Loss in `yieldSharesAfter` Calculation Due to Unnecessary Scaled Intermediate [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-sixpence-follow-up-apr-30th-2026/issues/20

Summary

When recomputing `yieldSharesAfter` after a yield collateral loss, the formula multiplies `yieldSharesBefore` by `liveYieldSuppliedScaled` and divides by `totalYieldCollateralBefore` – all in scaled units. Since `liveYieldSupplied` (the real amount) is already available at this point, using the scaled intermediate introduces unnecessary rounding error.

Recommendation

Use `liveYieldSupplied` directly.

Issue L-3: Imprecise Scaled Debt Adjustment Using Cross-Multiplication Instead of Adapter Lookup [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-sixpence-follow-up-apr-30th-2026/issues/21

Summary

When `poolDebt < actualDebt`, the code derives the adjusted `scaledDebt` via `(poolDebt * scaledDebt) / actualDebt`. This proportional cross-multiplication loses precision compared to simply calling `getScaledAmount(poolDebt)` on the lending adapter, which returns the authoritative scaled representation of the live debt.

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.