AUDIT

SPINE V2

GETRECON.XYZ
@GALLODASBALLO

Spine V2 Audit

Introduction

Alex The Entreprenerd performed a 2 weeks Security Review of Spine V2

Repos: https://github.com/spine-finance/spine-v2-sm

This review uses Code4rena Severity Classification

The Manual Review is done as a best effort service, while a lot of time and attention was dedicated to the security review, it cannot guarantee that no bug is left

As a general rule we always recommend doing one additional security review until no bugs are found, this in conjunction with a Guarded Launch and a Bug Bounty can help further reduce the likelihood that any specific bug was missed

About Recon

Recon offers boutique security reviews, invariant testing development and is pioneering Cloud Fuzzing as a best practice by offering Recon Pro, the most complete tool to run tools such as Echidna, Medusa, Foundry, Kontrol and Halmos in the cloud with just a few clicks

About Alex

Alex is the cofounder of Recon and a well known Lead Security Researcher with $500,000 in bounties and contest winnings.

- Code4rena - One of the most prolific and respected judges, won the Tapioca contest, at the time the 3rd highest contest pot ever
- Spearbit - Have done reviews for Tapioca, Threshold USD, Velodrome and more
- Recon - Centrifuge Invariant Testing Suite, Corn and Badger invariants as well as live monitoring

Additional Services by Recon

Recon offers:

- Audits powered by Invariant Testing - We'll write your invariant tests then perform an audit on your code.
- Cloud Fuzzing as a Service - The easiest way to run invariant tests in the cloud - Ask about Recon Pro.
- Invariant Tests writing - An engineer will write Chimera based Invariant Tests on your codebase.

Mitigation Status for Findings

Issue	Mitigated	PR Resolution
M-01 `totalBadDebt -= absorbedAmount;` paired with a transfer of tokens causes totalAssets to increase by a 2x rate. Causing Insolvency to the Vault	✅	https://github.com/spine-finance/spine-v2-sm/pull/25
M-02 Relative loss in `_withdraw` is not imputed to user when there's idle funds in the vault	✅	https://github.com/spine-finance/spine-v2-sm/pull/26
M-03 Setting a vault to Inactive can cause reverts for withdrawals	–	Acknowledged
M-04 minPTOut not validated	✅	https://github.com/spine-finance/spine-v2-sm/pull/29
Q-01 Adding timelocked to deposit and withdraw UX Improvement	✅	https://github.com/spine-finance/spine-v2-sm/pull/28/commits/a25f41c185c52a3013c4ff04c1f3965a2b92ae06
Q-02 QA Notes	✅	https://github.com/spine-finance/spine-v2-sm/pull/28/commits/a25f41c185c52a3013c4ff04c1f3965a2b92ae06
Q-03 LTV is not accounted in the borrow macro with pendle check	✅	https://github.com/spine-finance/spine-v2-sm/pull/23
Q-04 `msg.sender != borrower` can be incorrect for safes and ERC7702, as well as for other EOAs performing operations on behalf of users	✅	https://github.com/spine-finance/spine-v2-sm/pull/24
Q-05 Can reduce small losses by using previewDeposit into mint and refactoring to handle liquid assets as part of withdrawals	✅	https://github.com/spine-finance/spine-v2-sm/pull/27
Q-06 Dust in the vault can cause last withdrawer to be unable to redeem all their shares	✅	https://github.com/spine-finance/spine-v2-s

Table of Contents

- **Analysis**
  - A-01 Suggested Next Steps
  - A-02 SpineVault and BorrowController require pausing tied to absorbing bad debt to prevent sandwiching | Alternative ERC7540
  - A-03 Integration Risks
  - A-04 Admin Risks
- **Invariants**
  - I-01 Invariants Reviewed
- **Economic**
  - E-01 Economic considerations
  - E-02 LTV, LLTV & Compounded Deviation Threshold when using 2 oracles

M-01 `totalBadDebt -= absorbedAmount;` paired with a transfer of tokens causes totalAssets to increase by a 2x rate. Causing Insolvency to the Vault

Impact

Repaying bad debt in the current design leads to a 2x delta increase in `totalAssets` which leads to insolvency of the `SpineVault`

When repaying bad debt, 2 options can happen:

- The Vault performs the call (implying assets have been transferred to the vault), in that case, the increase of assets would have already occurred, meaning that you should not decrease the `totalBadDebt` as otherwise you'd end up reducing the debt (1x delta) and then adding assets (2x delta).
- Anyone else performs the call (assets are transfered to the vault then invested in subvaults (1x delta), then bad debt is reduced (2x delta)

Both scenarios lead to insolvency.

MATH

totalAssets = borrow + interest + idle Assets - bad debt -> Bad debt happens -> Borrow + interest (over stated) - bad debt (balances it out) + idle assets -> Repay bad debt

idleAssets + repaid bad debt (compensates for the bad debt) + borrow + interest (over stated) - bad debt -> idleAssets (with repayment) + borrow + interest (over stated) - 0 -> It's overstating the part tied to borrow + interest

Mitigation

Do not remove the bad debt once repaid, it should remain tracked

If you want to show how much bad debt was repaid you can add a second storage variable counting the debt repaid, however this should not be part of the `totalAssets` formula

M-02 Relative loss in `_withdraw` is not imputed to user when there's idle funds in the vault

Impact

`_withdraw` is using `_redeemFromSubVaults` to limit the amount of assets that the user will receive:

https://github.com/GalloBurner2/1661fc2a-spine-v2-invariants-1767450534/blob/1417be4b529577e854e71318a7d9d7ff7217b690/src/contracts/SpineVault.sol#L315-L328

```
    function _withdraw(
        address caller,
        address receiver,
        address owner,
        uint256 assets,
        uint256 shares
    ) internal virtual override {
        FeeInfo memory info = feeInfo;
        uint256 fee = _feeOnRaw(assets, info.exitFeeBps);
        uint256 totalNeeded = assets + fee;

        // Pull enough liquidity to pay receiver + fee.
        _redeemFromSubVaults(totalNeeded, address(this));
```

https://github.com/GalloBurner2/1661fc2a-spine-v2-invariants-1767450534/blob/1417be4b529577e854e71318a7d9d7ff7217b690/src/contracts/SpineVault.sol#L389-L407

```
    function _redeemFromSubVaults(uint256 assets, address to) internal {
        SubVault[] memory vaults = subVaults;
        uint256 _totalWeight = totalWeight;
        if (_totalWeight == 0) {
            // No sub vaults configured.
            // Transfer directly from this vault.
            IERC20(asset()).safeTransfer(to, assets);
        } else {
            for (uint256 i = 0; i < vaults.length; i++) {
                if (!vaults[i].active) {
                    continue;
                }
                uint256 vaultAssets = (assets * vaults[i].weight) /
                    _totalWeight;
                uint256 shares = vaults[i].vault.convertToShares(vaultAssets);
                vaults[i].vault.redeem(shares, to, address(this));
            }
        }
    }
```

`_redeemFromSubVaults` is limiting the assets lost from the vault and sent to the user by using `convertToShares` which rounds down, and in the case of Euler and Morpho is accrued with current interest (it's consistent with previewWithdraw`).

This means that the user will withdraw, on average, a lower amount of assets from the sub vault.

This is a safe approach, however, the code that follows it is:

https://github.com/GalloBurner2/1661fc2a-spine-v2-invariants-1767450534/blob/1417be4b529577e854e71318a7d9d7ff7217b690/src/contracts/SpineVault.sol#L326-L334

```
        // Pull enough liquidity to pay receiver + fee.
        _redeemFromSubVaults(totalNeeded, address(this)); /// @audit AMT is less than totalNeeded

        if (fee > 0 && info.feeVault != address(this)) {
            IERC20(asset()).safeTransfer(info.feeVault, fee);
        }
        uint256 assetBalance = IERC20(asset()).balanceOf(address(this));
        // Use the min value in case of rounding issues.
        assetBalance = Math.min(assetBalance, assets);
```

This means that when we call:

```
        uint256 assetBalance = IERC20(asset()).balanceOf(address(this));
        // Use the min value in case of rounding issues.
        assetBalance = Math.min(assetBalance, assets);
```

`assetBalance` can be greater than `totalNeeded`.

This is socializing the relative loss back to the vault, instead of imputing it to the withdrawer.

For the current integrations the risks is acceptable, however, from first principles, the vault is still socializing losses to the vault and not the caller.

POC

```
/// @notice POC: SpineVault sends more than it withdraws from subvault due to idle assets
    function test_pco_spineVaultSendsMoreThanWithdrawn() public {
        address actor = _getActor();
        address user2 = _getActors()[1];
        address subVaultAddr = deployedSubVaults[0];
        MockERC4626Tester subVault = MockERC4626Tester(subVaultAddr);

        // 1. Actor deposits
        spineVault_deposit(10e18, actor);

        // 2. Sub-vault gains 1 wei
        vm.prank(address(subVault));
        MockERC20(underlyingAsset).transfer(address(0), 1);

        // 3. User2 deposits
        vm.prank(user2);
        spineVault.deposit(10e18, user2);

        // 4. Donate idle assets to SpineVault
        vm.prank(actor);
        MockERC20(underlyingAsset).transfer(address(spineVault), 2);

        // 5. Check what subvault would give for actor's shares
        uint256 actorShares = spineVault.balanceOf(actor);
        uint256 expectedFromSubVault = subVault.previewRedeem(subVault.balanceOf(address(spineVault)) *
actorShares / spineVault.totalSupply());
        console2.log("Expected from subvault:", expectedFromSubVault);

        // 6. Redeem and check actual received
        uint256 balanceBefore = MockERC20(underlyingAsset).balanceOf(actor);
        vm.prank(actor);
        spineVault.redeem(actorShares, actor, actor);
        uint256 balanceAfter = MockERC20(underlyingAsset).balanceOf(actor);

        uint256 actualReceived = balanceAfter - balanceBefore;
        console2.log("Actual received:", actualReceived);

        // SpineVault sent more than subvault provided (used idle assets)
        assertGt(actualReceived, expectedFromSubVault, "Got more than subvault provided");
    }
```

Integration Notes

For Morpho and Euler it's not possible to cause a loss to the vault operation, since `converToShares` is accrued.

NOTE: I think this is not possible for Morpho or Euler https://github.com/morpho-org/metamorpho/blob/37714d67104523f32f8e7e31cd2c7a0506f800aa/src/MetaMorpho.sol#L638 | https://github.com/euler-xyz/euler-vault-kit/blob/5b98b42048ba11ae82fb62dfec06d1010c8e41e6/src/EVault/modules/Vault.sol#L35

Mitigation

I believe you should make the following change:

```
uint256 amtWithdrawn = _redeemFromSubVaults(totalNeeded, address(this));
// Where `amtWithdrawn` should be less than `totalNeeded` but could be more.
if(amtWithdrawn > totalNeeded) {
    amtWithdrawn = totalNeeded
}

if (fee > 0 && info.feeVault != address(this)) {
    IERC20(asset()).safeTransfer(info.feeVault, fee);
}

uint256 assetBalance = IERC20(asset()).balanceOf(address(this));
// Use the min value in case of rounding issues.
assetBalance = Math.min(assets, amtWithdrawn - fees);
```

M-03 Setting a vault to Inactive can cause reverts for withdrawals

Impact

When calling `setSubVaultStatus`, to set a vault to inactive, funds are not rebalanced.

https://github.com/GalloBurner2/1661fc2a-spine-v2-invariants-1767450534/blob/1417be4b529577e854e71318a7d9d7ff7217b690/src/contracts/SpineVault.sol#L221-L242

```
function setSubVaultStatus(
        address vaultAddress,
        bool isActive
    ) external onlyOwner {
        _timelocked();

        uint256 index = _getSubVaultIndex(vaultAddress);
        require(index < subVaults.length, "SpineVault: sub vault not exists");
        require(subVaults[index].active != isActive, "SpineVault: same status");

        subVaults[index].active = isActive;
        if (isActive) {
            _redeemAll();
            totalWeight += subVaults[index].weight;
            _supplyAll();
        } else {
            totalWeight -= subVaults[index].weight;
            // @dev: If deactivating, do not redeem; assets remain deposited in the vault.
        }

        emit SubVaultStatusSet(vaultAddress, isActive);
    }
```

This can cause withdrawals to revert, since the `totalAssets` will correctly account for these 0 weight vaults, but they won't be usable for `_withdraw` which will attempt to redeem more shares than the ones it has available for the vaults that are active.

POC

The POC setups up 3 vaults, has a user deposit, then changes the weights to have only 2 vaults active.

It will revert when attempting to withdraw more than all shares available to the 2 vaults that have non zero weights:

```
Backtrace:
  at MockERC4626Tester.redeem
  at SpineVault.withdraw
  at CryticToFoundry.test_pco_withdrawal_revertsWhenVaultGoesInactive_threeVaults

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 28.14ms (25.14ms CPU time)

Ran 1 test suite in 158.53ms (28.14ms CPU time): 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/recon/CryticToFoundry.sol:CryticToFoundry
[FAIL: ERC20InsufficientBalance(0xa0Cb889707d426A7A386870A03bc70d1b0697598,
100000000000000000000000000000 [1e29], 150000000000000000000000000000 [1.5e29])]
test_pco_withdrawal_revertsWhenVaultGoesInactive_threeVaults() (gas: 6632757)
```

```
// ==================== PCO: Withdrawal Logic Burns More Shares ====================

    /// @notice PCO: 3 vaults with 33.3% weight each, one goes inactive, withdrawal should revert
    /// @dev Tests that:
    ///      1. 3 vaults are setup with equal weights (33.3% each)
    ///      2. User deposits
    ///      3. One vault goes inactive (weight drops to 0%)
    ///      4. User tries to withdraw full amount and it reverts because
    ///         withdrawal logic tries to redeem proportionally from active vaults only,
    ///         but assets are stuck in the inactive vault
    function test_pco_withdrawal_revertsWhenVaultGoesInactive_threeVaults() public {
        address actor = _getActor();

        // 1. Deploy 2 additional sub-vaults (we already have 1 from setup)
        MockERC4626Tester subVault2 = new MockERC4626Tester(underlyingAsset);
        MockERC4626Tester subVault3 = new MockERC4626Tester(underlyingAsset);
        deployedSubVaults.push(address(subVault2));
        deployedSubVaults.push(address(subVault3));

        // Approve underlying for new sub-vaults
        vm.prank(actor);
        MockERC20(underlyingAsset).approve(address(subVault2), type(uint256).max);
        vm.prank(actor);
        MockERC20(underlyingAsset).approve(address(subVault3), type(uint256).max);

        // 2. Set first vault weight to 100 (we'll add two more with 100 each = 33.3% each)
        address subVault1Addr = deployedSubVaults[0];
        bytes memory setWeightData = abi.encodeWithSelector(
            ISpineVault.setSubVaultWeight.selector,
            subVault1Addr,
            100
        );
        spineVault.submit(setWeightData);
        spineVault.setSubVaultWeight(subVault1Addr, 100);

        // 3. Add vault 2 with weight 100
        SubVault memory newSubVault2 = SubVault({
            vault: IERC4626(address(subVault2)),
            weight: 100,
            active: true
        });
        bytes memory addVault2Data = abi.encodeWithSelector(ISpineVault.addSubVault.selector,
newSubVault2);
        spineVault.submit(addVault2Data);
        spineVault.addSubVault(newSubVault2);

        // 4. Add vault 3 with weight 100
        SubVault memory newSubVault3 = SubVault({
            vault: IERC4626(address(subVault3)),
            weight: 100,
            active: true
        });
        bytes memory addVault3Data = abi.encodeWithSelector(ISpineVault.addSubVault.selector,
newSubVault3);
        spineVault.submit(addVault3Data);
        spineVault.addSubVault(newSubVault3);

        // Verify we have 3 vaults with equal weights (100 each, totalWeight = 300)
        assertEq(spineVault.subVaultsLength(), 3, "Should have 3 sub-vaults");

        // 5. User deposits
        uint256 depositAmount = 30e18; // 30 tokens, should distribute ~10 each
        spineVault_deposit(depositAmount, actor);

        uint256 userShares = spineVault.balanceOf(actor);
```

```
        console2.log("User shares after deposit:", userShares);
        assertTrue(userShares > 0, "User should have shares");

        // Check distribution - each vault should have ~10e18 (33.3%)
        uint256 vault1Assets = MockERC4626Tester(subVault1Addr).totalAssets();
        uint256 vault2Assets = subVault2.totalAssets();
        uint256 vault3Assets = subVault3.totalAssets();
        console2.log("Vault 1 assets:", vault1Assets);
        console2.log("Vault 2 assets:", vault2Assets);
        console2.log("Vault 3 assets:", vault3Assets);

        // 6. Set vault 1 to inactive (33.3% of assets become "stuck" for withdrawal purposes)
        // When inactive, assets stay in vault but it's not used for withdrawal redemption
        bytes memory setStatusData = abi.encodeWithSelector(
            ISpineVault.setSubVaultStatus.selector,
            subVault1Addr,
            false
        );
        spineVault.submit(setStatusData);
        spineVault.setSubVaultStatus(subVault1Addr, false);

        // Verify vault 1 is inactive
        (,, bool active) = spineVault.subVaults(0);
        assertFalse(active, "Vault 1 should be inactive");

        // 7. User tries to withdraw full amount
        // This should revert because:
        // - totalWeight is now 200 (only vaults 2 and 3)
        // - But ~33% of assets are in vault 1 which is inactive
        // - _redeemFromSubVaults will try to redeem proportionally from active vaults
        // - Active vaults don't have enough assets to cover the full withdrawal

        uint256 maxWithdrawable = spineVault.maxWithdraw(actor);
        console2.log("Max withdrawable:", maxWithdrawable);

        // Try to withdraw full deposit amount - should fail
        // The withdrawal logic calculates: (assets * vault.weight) / totalWeight for each active vault
        // With 30e18 withdrawal and totalWeight=200, it tries to get 15e18 from each active vault
        // But each active vault only has ~10e18

        vm.prank(actor);
        vm.expectRevert(); // Should revert due to insufficient assets in active vaults
        spineVault.withdraw(depositAmount, actor, actor);
    }
```

Mitigation

I'm not sure this functionality is necessary, I recommend that when a vault is set to inactive, you rebalance funds off of it.

M-04 minPTOut not validated

Impact

`_pendleMinPtAmount` can be 0:

https://github.com/spine-finance/spine-v2-sm/blob/55b0a226273099520ee11ffd838450fa8eef474c/src/contracts/strategies/PendleStrategy.sol#L42-L47

```
        bytes memory callbackData = abi.encode(
            _pendleMarket,
            _pendleMinPtAmount, /// @audit missing slippage check.
            _pendleTokenInput,
            _approxParams
        );
```

This is not validated in `PendleStrategy` nor in the `PendleRouter`

POC

Because of this, an attacker can sandwich swap operations and cause them to lose value.

https://claude.ai/share/5eef28c6-bce0-4e76-bd88-fcc2cc88f307

Mitigation

While a non-zero check is not fully sufficient, it's the best practice against sandwiching

Additional Instances

The same can happen for `_pendleTokenOutput.minTokenOut` here: https://github.com/spine-finance/spine-v2-sm/blob/55b0a226273099520ee11ffd838450fa8eef474c/src/contracts/strategies/PendleStrategy.sol#L75-L76

```
            _pendleTokenOutput
```

Q-01 Adding timelocked to deposit and withdraw UX Improvement

Impact

Spine is considering adding a timelock that will not always be active. For certain operations.

The current code for the timelock is the following:

https://github.com/spine-finance/spine-v2-sm/blob/55b0a226273099520ee11ffd838450fa8eef474c/src/contracts/TimeLock.sol#L17-L41

```
    function submit(bytes calldata data) external onlyOwner {
        bytes32 dataHash = keccak256(data);
        require(executableAt[dataHash] == 0, "Already submitted");

        bytes4 selector = bytes4(data);
        uint256 delay = selector == this.decreaseTimelock.selector
            ? timelock[bytes4(data[4:8])] // first argument of decreaseTimelock() /// @audit ???
            : timelock[selector];
        executableAt[dataHash] = block.timestamp + delay;

        emit TimelockSubmitted(selector, data, executableAt[dataHash]);
    }

    function _timelocked() internal virtual {
        bytes4 selector = bytes4(msg.data);
        uint256 delay = selector == this.decreaseTimelock.selector
            ? timelock[bytes4(msg.data[4:8])] // first argument of decreaseTimelock() /// @audit ???
            : timelock[selector];

       if(delay == 0) {
          return; // accepted // Can also emit an event
       }


        bytes32 dataHash = keccak256(msg.data);
        require(executableAt[dataHash] != 0, "Data not timelocked");
        require(
            block.timestamp >= executableAt[dataHash],
            "Timelock not expired"
        );

        executableAt[dataHash] = 0;
        emit TimelockAccepted(selector, msg.data);
    }
```

This would force a caller to first submit, then call the actual function.

In order to improve the UX you could do the following:

- When the timelock is 0, the _timelocked function should allow the call without having to call submit.

Refactoring

```
    function _timelocked() internal virtual {
        bytes4 selector = bytes4(msg.data);
        bytes32 dataHash = keccak256(msg.data);
        require(executableAt[dataHash] != 0, "Data not timelocked");
        require(
            block.timestamp >= executableAt[dataHash],
            "Timelock not expired"
        );

        executableAt[dataHash] = 0;
        emit TimelockAccepted(selector, msg.data);
    }
```

Q-02 QA Notes

_prunePositions can be changed on a total liquidation

Since you're closing all positions you don't need to swap and pop You can just pop.

PRICE_FEED_BUFFER_DELAY is too short

https://github.com/spine-finance/spine-v2-sm/blob/55b0a226273099520ee11ffd838450fa8eef474c/src/contracts/BorrowController.sol#L40-L41

```
    uint32 public constant PRICE_FEED_BUFFER_DELAY = 10; // 10 seconds
```

While heartbeat can be adapted, PRICE_FEED_BUFFER_DELAY would either need to be removed (heartbeat should contain a buffer) or should be set to a higher value, perhaps half an hour.

In times of high congestion, 10 seconds (less than a block) would cause DOS

Unused param

https://github.com/spine-finance/spine-v2-sm/blob/55b0a226273099520ee11ffd838450fa8eef474c/src/contracts/BorrowController.sol#L551-L552

```
        (tokenAmountIn, , res) = _repay( /// @audit QA: Unused param, can remove
```

Cannot closeBorrowStrategy when the position is underwater

A user is unable to call `closeBorrowStrategy` when their position is underwater due to this check: require(assets <= pendleNetTokenOut, "INSUFFICIENT_REPAY"); /// @audit ?? Is this correct?

This forces the user to add collateral first, then use the strategy

Whereas an alternative would be to differentiate between the pendleToken the user will withdraw from the collateral and the amount that will be transferred from the user

```
        // withdraw PT from collateral
        IBorrowController(msg.sender).withdrawCollateral(
            _borrower,
            _collateral,
            ptExitAmount
        );
        // transfer PT to this contract
        IERC20(_collateral).safeTransferFrom(
            _borrower,
            address(this),
            ptExitAmount + extraAMT /// @audit ExtraAMT would be the extra the user can provide
        );
```

Q-03 LTV is not accounted in the borrow macro with pendle check

Impact

`openBorrowStrategy` uses the following code:

https://github.com/spine-finance/spine-v2-sm/blob/55b0a226273099520ee11ffd838450fa8eef474c/src/contracts/strategies/PendleStrategy.sol#L59-L60

```
        require(totalDebt <= totalPt, "INSUFFICIENT_LEVERAGE"); /// @audit looks insufficient.
Shouldn't this be a ratio?
```

The check is not particularly useful for slippage since Pendle will use the `_pendleMinPtAmount` for slippage.

The check could be there to ensure that the collateral will be sufficient for the entirety of the loan. However, that's not the case. Because `totalDebt` is a flat amount just like `totalPt` however, when computing LTVs we will need to scale `totalPt` by the `lltv`.

When executing the code, that line will be called the `ltvProtected(borrower, collateralToken)` check in the `BorrowController` due to this, you won't need to check for `ltv`.

Therefore the only valuable check from that line would be tied to the `lltv` as it would ensure that the value of the collateral will always be greater than the value of the debt.

Note that PT tokens have their value vest linearly as they reach maturity, therefore a spot lltv check would ensure the collateral is sufficient and that it will be sufficient for the entirety of the loan, however this will come at the cost of capital efficiency.

Suggested Change

I suggest removing the check

Q-04 `msg.sender != borrower` can be incorrect for safes and ERC7702, as well as for other EOAs performing operations on behalf of users

Impact

The `BorrowController` allows a callback whenever `msg.sender != borrower`

https://github.com/spine-finance/spine-v2-sm/blob/55b0a226273099520ee11ffd838450fa8eef474c/src/contracts/BorrowController.sol#L499-L509

```
        if (msg.sender != borrower) { /// @audit Should this be something else? Perhaps having callbackdata?
            res = IBorrowBroker(msg.sender).onBorrow(
                borrower,
                collateralToken,
                maturity,
                interestRate,
                assets,
                debtAmount,
                callbackData
            );
        }
```

This can be incorrect in the following scenarios:

- Another EOA is performing the operation
- The caller is a Safe
- The caller uses ERC7702

Mitigation

Check if `callbackData` is non empty and perform the callback in that case

Q-05 Can reduce small losses by using previewDeposit into mint and refactoring to handle liquid assets as part of withdrawals

Impact

`_supplyAll` looks as follows:

https://github.com/GalloBurner2/1661fc2a-spine-v2-invariants-1767450534/blob/1417be4b529577e854e71318a7d9d7ff7217b690/src/contracts/SpineVault.sol#L340-L357

```
    function _supplyAll() internal {
        uint256 _totalWeight = totalWeight;
        if (_totalWeight == 0) {
            return; // No sub vaults configured.
        }

        uint256 assetBalance = IERC20(asset()).balanceOf(address(this));
        SubVault[] memory vaults = subVaults;
        for (uint256 i = 0; i < vaults.length; i++) {
            if (!vaults[i].active) {
                continue; // Skip inactive vaults.
            }
            uint256 vaultAssets = (assetBalance * vaults[i].weight) /
                _totalWeight;
            IERC20(asset()).forceApprove(address(vaults[i].vault), vaultAssets);
            vaults[i].vault.deposit(vaultAssets, address(this));
        }
    }
```

It's using `deposit` which is a lossy operation.

This causes it to mint 1 less share, causing a loss of up to `price per share - 1`

In contrast, calling `mint` to mint the same amount of shares that `deposit` will mint, will cause no losses.

Therefore, you can refactor the code, to be slightly more gas expensive, and use: `shares = vault.previewDeposit` to determine the amount of shares you'll receive, then call `vault.mint(shares)` to mint them at no loss.

This, paired with a refactoring that uses idle liquidity for withdrawals first, will reduce execution losses.

Q-06 Dust in the vault can cause last withdrawer to be unable to redeem all their shares

Impact

`redeem` and `withdraw` will compute the correct amount of assets to transfer away from the `SpineVault`.

It will then call `_withdraw`.

https://github.com/GalloBurner2/1661fc2a-spine-v2-invariants-1767450534/blob/1417be4b529577e854e71318a7d9d7ff7217b690/src/contracts/SpineVault.sol#L315-L338

```
    function _withdraw(
        address caller,
        address receiver,
        address owner,
        uint256 assets,
        uint256 shares
    ) internal virtual override {
        FeeInfo memory info = feeInfo;
        uint256 fee = _feeOnRaw(assets, info.exitFeeBps);
        uint256 totalNeeded = assets + fee;

        // Pull enough liquidity to pay receiver + fee.
        _redeemFromSubVaults(totalNeeded, address(this));

        if (fee > 0 && info.feeVault != address(this)) {
            IERC20(asset()).safeTransfer(info.feeVault, fee);
        }
        uint256 assetBalance = IERC20(asset()).balanceOf(address(this));
        // Use the min value in case of rounding issues.
        assetBalance = Math.min(assetBalance, assets);

        super._withdraw(caller, receiver, owner, assetBalance, shares);
    }
```

This will use `_redeemFromSubVaults(totalNeeded, address(this));` which is ignoring the idle assets available to the vault.

https://github.com/GalloBurner2/1661fc2a-spine-v2-invariants-1767450534/blob/1417be4b529577e854e71318a7d9d7ff7217b690/src/contracts/SpineVault.sol#L388-L407

```
function _redeemFromSubVaults(uint256 assets, address to) internal {
    SubVault[] memory vaults = subVaults;
    uint256 _totalWeight = totalWeight;
    if (_totalWeight == 0) {
        // No sub vaults configured.
        // Transfer directly from this vault.
        IERC20(asset()).safeTransfer(to, assets);
    } else {
        for (uint256 i = 0; i < vaults.length; i++) {
            if (!vaults[i].active) {
                continue;
            }
            uint256 vaultAssets = (assets * vaults[i].weight) /
                _totalWeight;
            uint256 shares = vaults[i].vault.convertToShares(vaultAssets);
            vaults[i].vault.redeem(shares, to, address(this));
        }
    }
}
```

Due to this, in some scenarios, such as when you have only one depositor, they will be unable to withdraw all of the assets that they are entitled to.

POC

The issue can happen due to rounding of amountDeposited / vaultWeights, or via a donation.

By performing a simple donation we can show this as a grief:

```
// ==================== PCO: Withdrawal Ignores Idle Assets ====================

/// @notice POC: Deposit, donate 10 wei, redeem all shares - fails due to idle assets ignored
function test_pco_withdrawal_ignoresIdleAssets_simple() public {
    address actor = _getActor();

    // User deposits
    uint256 depositAmount = 10e18;
    spineVault_deposit(depositAmount, actor);
    uint256 userShares = spineVault.balanceOf(actor);

    // Donate 10 wei to SpineVault (creates idle assets)
    vm.prank(actor);
    MockERC20(underlyingAsset).transfer(address(spineVault), 10);

    // Try to redeem all shares - reverts because _redeemFromSubVaults ignores idle assets
    vm.prank(actor);
    vm.expectRevert();
    spineVault.redeem(userShares, actor, actor);
}
```

Mitigation

Withdrawals should use available assets first, then attempt to withdraw only the assets necessary from the subvaults.

https://github.com/GalloBurner2/1661fc2a-spine-v2-invariants-1767450534/blob/1417be4b529577e854e71318a7d9d7ff7217b690/src/contracts/SpineVault.sol#L315-L327

```
    function _withdraw(
        address caller,
        address receiver,
        address owner,
        uint256 assets,
        uint256 shares
    ) internal virtual override {
        FeeInfo memory info = feeInfo;
        uint256 fee = _feeOnRaw(assets, info.exitFeeBps);
        uint256 totalNeeded = assets + fee;

        // Pull enough liquidity to pay receiver + fee.
        _redeemFromSubVaults(totalNeeded, address(this));
```

Should change to:

```
uint256 available = IERC20(asset()).balanceOf(address(this));
uint256 totalNeeded = assets + fee;
if(totalNeeded < available) {
  totalNeeded = 0;
} else {
  totalNeeded -= available; // We will withdraw something
}
```

G-01 Deployment can be changed to set the borrow controller to be immutable

Saves 2.1k gas per operation SpinVault.constructor -> calls factory and deploys borrowController passing the asset as to break the circular dependency

A-01 Suggested Next Steps

Executive Summary

The codebase is massively more mature.

Most of the findings from this review are tied to very unlikely edge cases or mistakes.

Overall I wasn't able to find issues on a vault that is properly setup (has an initial liquidity and integrations are safe).

The codebases was provided to me with both unit and invariant testing.

This indicates to me a high level of code maturity.

I'm reticent to ever recommend deploying after a solo review, as the risk of missing something is too high.

For this reason my recommendation is for the codebase to undergo a public contest (recommended), or a bug bounty during guarded launch.

Exposing the codebase to a high amount of people increases the likelihood that any remaining edge case will be flagged.

Recommendation

Make the code public and the invariant testing suite public, along with a corpus that produces 100% coverage.

Do a contest and reward new Properties as part of the QA pot.

Then proceed with a guarded launch and bug bounty.

A-02 SpineVault and BorrowController require pausing tied to absorbing bad debt to prevent sandwiching | Alternative ERC7540

Bad Debt Socialization

Per the current design bad debt will be locked in during a liquidation

This means that anyone that can frontrun the liquidation, withdraw their position, then mint shares after, will avoid the loss

Pausing withdrawals before socializing bad debt would prevent this, although this risk may be best documented and not fixed.

The alternative to more fairly avoid this issue is to lock deposits and withdrawals when loans are about to mature.

It's worth noting that this is an extremely common issue in vaults, and is one of the reasons for using ERC7540 as the manager can define periods for redemptions and periods in which redemptions are locked.

Bad Debt Repayment

Repaying bad debt causes a step wise increase in the PPS. This can be sandwiched for a gain.

Pausing deposits before repaying prevents the frontrunning. Although it's not a great design decision.

An alternative would be to vest the bad debt repayment, but this also is not optimal.

The optimal solution would be to repay the bad debt as it is formed, which may possible by creating an onchain "insurance fund" that when the `totalBadDebt` for a vault changes, allows repaying those same amounts. However, this change can be error prone.

It's also worth noting that bad debt is not a high likelihood case when working with sound collaterals and a proper risk framework

Force Shutdown

Another important design decision missing are forced shutdown, meaning enforcing that ltv will decrease, that positions can no longer be opened, but that withdrawal, repayment and liquidations are.

I believe that given the fact that you have a separation between LTV and LLTV, then shutdowns cannot be forced (without altering the LLTV and generating a ton of extractable value), however you have the ability to reduce LTV to reduce risk in the future.

Mitigation

Implement selective pause, or timelocks to ensure that bad debt and gains are correctly socialized

A-03 Integration Risks

Bad debt repayment can be sandwiched. [MED]

> If you add pausing you can prevent it. Alternatively you'd repay the bad debt as some sort of vested amount, so that it takes time to take that interest.

SpineVault cannot be borrowed [MED]

> Since it can be donated to, the totalAssets can be inflated. Do not allow borroiwng of the SpineVault. NOTE: For usage as collateral It will required ltv < lltv with a few % points to avoid self liquidations

SpineVault cannot be used with vaults that charge a fee on withdrawal

If you use SpineVault as collateral you must have a separation between ltv and lltv

In lack of a separation, users can self liquidate by calling repay and causing slight losses, since the debt will decrease by the exact amount but the assets will decrease by a bit less (since they are invested in the vault and can have slight losses)

The collateral for the vault must be non rebasing and not a vault that can have losses

A vault that can have losses can open up to self liquidation attacks

A-04 Admin Risks

Admin can borrow all then add malicious vault

Since manager == admin, the malicious manager could add a malicious integration

The timelock is bypassed by borrowing all assets from the vault, which prevents withdrawals, by doing so the manager can steal the assets inspite of the timelock, the timelock doesn't provide the necessary protection

I believe that you need different roles for whitelisting global integrations and adding them to each SpineVault

Same role between System Admin and Manager means the timelock can be disabled after one delay

Since owner is the admin and the manager, they can set the timelock duration after one delay

Meaning that if they go rogue / get exploited, users will have one timelock duration to exit their positions

I-01 Invariants Reviewed

Executive Summary

During the 2 week review I wrote a new Invariant Testing suite using Recon Magic: https://github.com/GalloDaSballo/spine-v2-invariants

The corpus is available here: https://staging.getrecon.xyz/shares/66de9fa2-db72-4dda-8254-93e61beef031

I used it to write a few global properties as to test well as a few important scenarios

Properties Tested

#	Function Name	Property Description	Passing
1	property_can_never_self_liquidate	Cannot self liquidate if not liquidatable before the operation	✅
2	property_debt_consistency	Total debt should be greater than or equal to total value of debt for all actors	✅
3	property_sum_of_weights_equals_total	Sum of active sub-vault weights == totalWeight tracked by vault	✅
4	property_debt_relationships_hold	valueOf <= debtAmount before maturity (PV <= FV)	✅
5	property_debt_relationships_hold	valueOf == debtAmount at/after maturity	✅
6	property_debt_value_bounded	Current value should never exceed debtAmount	✅
7	property_healthy_position_not_liquidatable	A healthy position (healthFactor >= 100%) should not be liquidatable	✅
8	property_deposit_increases_total_assets	Deposit should increase totalAssets by the exact deposited amount	❌
9	spineVault_borrowAsset	Debt value should be less than or equal to debt of actor at all times	✅
10	spineVault_borrowAsset	Debt (debtOf) must increase by at least 1 wei to avoid abuse	✅
11	spineVault_borrowAsset	Debt value (valueOf) must increase by at least 1 wei to avoid abuse	✅
12	borrowController_repay	Repay should decrease debt by the repaid amount	✅
13	borrowController_repay	User should pay at least the current value (no underpayment)	✅
14	borrowController_repay	User should not pay more than 2x the current value (bounded overpayment)	✅
15	doomsday_can_always_liquidate	Liquidation should succeed if actor has enough assets to cover max debt	✅
16	doomsday_spineVault_withdraw_clamped_can_never_revert	Withdraw up to maxWithdraw should always succeed	❌

E-01 Economic considerations

Morpho and Euler can have non locked losses

Monitor the vaults to avoid socializing losses

disableCollateral prevents more loans

But you can still deposit collateral, which means that a liquidation could be postponed

You could have a rule that you will decrease the LLTV for certain positions after some time.

Lack of minimum position size can make liquidations hard to perform

require(assets > 0, "Amount must be greater than 0");

Could be changed to be a higher value, e.g. 1e6 to ensure that the debt and collateral pricing doesn't lead to a scenario where a tick of interest would lead to a position that is not liquidatable in a healthy way.

Reduction of premium may lead to less likely liquidations

https://github.com/spine-finance/spine-v2-sm/blob/55b0a226273099520ee11ffd838450fa8eef474c/src/contracts/BorrowController.sol#L642-L646

```
            if (borrowerCollateralValue >= originalDebtPV) {
                // Collateral value covers the debt PV; penalty just isn't fully realizable
                tokenAmountIn = originalDebtPV; /// @audit and premium is reduced, but that's OK
                badDebt = 0;
            } else {
```

This line implies that the premium paid will be reduced.

From an analysis on Euler, even a small amount of premium are sufficient for liquidators to take action.

I believe this won't be an issue.

LVR Feeds could be a profitable addition to the protocol

LVR feeds are feeds that incorporate an auction that has to be paid before the price is updated.

This is a way for Spine to recapture most of the value it leaks to pay for liquidations

And can lead to reducing other fees

E-02 LTV, LLTV & Compounded Deviation Threshold when using 2 oracles

Compounded Deviation Threshold Risks

`BorrowController` will use a collateral and a debt token.

Both of these will be subject to oracle inaccuracies, typically constrained by the oracle Deviation Threshold.

When you have 2 Deviation Threshold, this inaccuracy becomes multiplicative.

It's important that when setting up the discrepancy between `ltv` and `lltv` you consider it.

Allowing for a safety margin between `ltv` and `lltv` will prevent self liquidation attacks.

Self Liquidation Risks

Given sane parameters, the main risk tied to self liquidations is the ability to escape the early closing fee

Additionally, liquidations pay `valueOf` on the "intended" interest rate, meaning a self liquidation is technically cheaper for a user.

Separating ltv from lltv to prevent self liquidations (ensuring that the next tick of interest will not allow for a liquidation), will prevent this from happening.

Suggestion

Use this tool Then ensure you use a lltv - ltv that is at least greater than the result

https://getrecon.xyz/tools/oracle-drift

Additional Services by Recon

Recon offers:

- Audits powered by Invariant Testing - We'll write your invariant tests then perform an audit on your code.
- Cloud Fuzzing as a Service - The easiest way to run invariant tests in the cloud - Ask about Recon Pro.
- Invariant Tests writing - An engineer will write Chimera based Invariant Tests on your codebase.