

SECURITY REVIEW

SPINE FINANCE



GETRECON.XYZ
@GALLODASBALLO

Spine Finance Security Review

Introduction

Alex the Entrepreneur performed a 2 week Solo Review of Spine Finance

He also wrote a complimentary invariant testing suite for the team.

Repo: <https://github.com/spine-finance/resupplied-sm/>

Commit: ae4a9cdadc0cf96ae6267a2797afe717b52e60d7

Repo: <https://github.com/spine-finance/spine-pendle-strategies/>

Commit: a97f465878e61bd33ab663b11050727aa37d1c03

This review uses [Code4rena Severity Classification](#)

The 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

Given the state of the codebase (no tests), as well as the significant number of issues that was identified in the review. I cannot recommend the code is deployed in this state. Please see Suggested Next Steps for more information.

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.

Table of Contents

• High

- H-01 `onCloseBorrowingPositionEarly` and `onRepayPosition` take the collateral balance from the wrong address
- H-02 `100% - 1` wei Swap, Combined with Lossy ERC4626 Vault Withdrawal Results in permanent DOS of the Pool
- H-03 Pendle on Spine Strategy Callbacks trigger the `nonReentrant` guard and revert

• Med

- M-01 `closeLeverageBorrowingStrategyEarly` slippage check is insufficient
- M-02 `PendleOnSpineStrategy` is not compatible with `USDT` (use `forceApprove`)
- M-03 Lack of Slippage check in Pendle Strategy
- M-04 Pool Creator can permanently DOS the pool by withdrawing all liquidity
- M-05 Incorrect `lltv` and `ltv` reset for `poolData.tokenAddress`
- M-06 `PRICE_FEED_BUFFER_DELAY` is using the wrong units and is only 60 seconds
- M-07 `RestakingRouter.openLendingPosition` is computing the wrong equity risk
- M-08 Fee math is inconsistent when using `amountIn` vs `amountOut`
- M-09 The way fee impacts the Invariant is Inconsistent leading to repricing of bonds
- M-10 `handleCashOut` Loss should be imputed to receiver, not the system, risk of loss socialization
- M-11 The system will not work with USDT or other tokens that do not return a `bool` on `approve`
- M-12 Venus Integration leaks yield due to using `exchangeRateStored`

• QA

- Q-01 Interest rate arbitrage was possible when `handleCashOut` was causing a loss to the system and not the caller
- Q-02 `closeLeverageBorrowingStrategyEarly` is approving an incorrect amount of tokens
- Q-03 `_pool.tokenAddress` should never be a token that can rebase (vaults)
- Q-04 `PendleOnSpineStrategy` risky patterns that can be fixed
- Q-05 `onOpenBorrowingPosition` could use a offchain computed hint for efficiency
- Q-06 Gas & QA spine-pendle-strategies
- Q-07 `removeCollateralToken` doesn't clear allowances
- Q-08 Bond data can be deleted before all bonds have been redeemed
- Q-09 Informational: You can open a borrowing position on zero lent due to first liquidity provider
- Q-10 Inaccurate `lpDepositAmount` causes `equityRiskRatio` math to underestimate the available liquidity
- Q-11 Precision loss of y/X when y is very small could be used to break the invariant and reprice bonds
- Q-12 UX Improvement - Make it easier to fetch deployment info
- Q-13 `poolData.vault` is provided but never used
- Q-14 `RESTAKING_POOL_TYPE.NONE` is not supported

- Q-15 `lltv` is validated only when first adding a collateral
- Q-16 AMM Fees should be rounded up as they are consistently undercharged
- Q-17 Euler deposits can be forced to revert to ensure that they do not earn yield
- Q-18 CEI Breaking Code
- Q-19 Off by one error in `handlePaymentForSingleMaturity`
- Q-20 `LoanCalculator` should use feeds with 8 decimals
- Q-21 Discrepancy between view functions and actual swap functions
- Q-22 `addMaturity` allows adding maturities in the past
- Q-23 Gotcha: Collateral Withdrawal can be Blocked by Cross-Collateral Solvency Check
- Q-24 Early return true on 0 borrows
- Q-25 No Event on Liquidation
- Q-26 Wrong event params on `LiquidityAdded`

- **Analysis**

- A-01 Suggested Next Steps
- A-02 Governance Mistake / PK Leak Risks

- **Economic**

- E-01 LTV vs LLTV needs to be conscious of compounded Oracle Drift

- **Invariants**

- I-01 Additional Properties
- I-02 Invariant Testing

H-01 `onCloseBorrowingPositionEarly` and `onRepayPosition` take the collateral balance from the wrong address

`onRepayPosition` and `onCloseBorrowingPositionEarly` withdraw the user collateral in this way:

<https://github.com/spine-finance/spine-pendle-strategies/blob/a97f465878e61bd33ab663b11050727aa37d1c03/contracts/PendleOnSpineStrategy.sol#L294-L320>

```
function onRepayPosition( /// @audit Forces Maturities in Spine to match the PT token, not always
the case
    address _account,
    address _poolAddress,
    address _collateralToken,
    uint256 _maturity,
    bytes memory _callbackData,
    uint _tokenAmountIn
) external override onlySpineRouter returns (bytes memory res) {
    // decode callback data
    (
        PendleActionParams memory pendleData,
        TokenOutput memory pendleTokenOutput
    ) = abi.decode(_callbackData, (PendleActionParams, TokenOutput));

    // check collateral balance
    uint256 ptBalance = IRestakingRouter(spineRouter).getCollateralBalance(
        msg.sender,
        _collateralToken,
        _poolAddress
    );
    // withdraw collateral
    IRestakingRouter(spineRouter).withdrawCollateral(
        _account,
        _collateralToken,
        _poolAddress,
        ptBalance
    );
};
```

They first `getCollateralBalance(msg.sender)` and then `withdrawCollateral(_account)`

The calls is a callback done by the `onlySpineRouter`

This means that `msg.sender` will be the `spineRouter`

This will cause `getCollateralBalance` to be incorrectly computed (most likely as 0)

Making the code not work in prod

Mitigation

Replace:

```
uint256 ptBalance = IRestakingRouter(spineRouter).getCollateralBalance(  
    msg.sender,  
    _collateralToken,  
    _poolAddress  
);
```

With:

```
uint256 ptBalance = IRestakingRouter(spineRouter).getCollateralBalance(  
    _account,  
    _collateralToken,  
    _poolAddress  
);
```

H-02 `100% - 1` wei Swap, Combined with Lossy ERC4626 Vault Withdrawal Results in permanent DOS of the Pool

Impact

Throughout the codebase `y` and `X` are scaled, following this formula:

<https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingBondMM.sol#L252-L255>

```
y_new += IERC20(underlyingToken).balanceOf(address(this));
uint _X = X.intoUint256();
X = ud((_X * y_new) / y);
y = y_new;
```

The formula fundamentally uses the old `y` and scales `X` based on the ratio at which `y_new` has changed

This ensures that repricing of LP tokens is imputed to LPs

However, the division can be unsafe and can result in a permanently bricked pool whenever `y` is zero.

This scenario is made possible by the fact that we can trade on all available `y`, no reserve (even small) is left to ensure that the ratio between `X` and `y` maintains the invariants at the edge cases.

An attacker would in theory simply want to swap out 100% of the `y` as to cause the pool to stop working

However, this same overflow will protect against a direct swap per this line:

<https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingBondMM.sol#L528-L529>

```
X = y_new * ((_y / y_new).pow(alpha) * (X / _y + ud(ONE)) - ud(ONE));
```

In order to achieve the overflow we combine #21 with this idea.

The high level attack path is the following:

- open a borrow position for all but 1 wei of `y`
- Trigger the ERC4626 to have a loss due to using `withdraw` (which rounds up shares)
- Leave the `y` in storage to be 1
- On the next call to `SyncReward` the BondMM will check for it's balance and update `y` to be 0
- The compiler inserted revert on division by zero will permanently DOS the pool

POC

Please note that this POC uses 0 fees in order to more easily identify the scenario for the attack

Adding fees does make the attack more complex mathematically, however, given certain conditions (e.g. Vault Shares have a high price, therefore losing 1 wei causes a sizeable loss), the attack can be performed with fees as well.

We:

- Donate some assets to the Vault in order to rebase the shares
- Borrow all but 1 wei
- Perform another action to demonstrate that it will revert

```
function test_can_i_zero_y() public {
    switchActor(1);
    switch_asset(1);
    asset_mint(address(mockERC4626Vault), uint128(mockERC4626Vault.totalSupply()) - 1); // Cause
rounding

    // Trigger Accrual
    uint256 shares = restakingRouter_addLiquidity(1e6, 0);
    restakingRouter_withdrawLiquidity(shares);

    collOracle_setLatestRoundDataReturn(1, 1e18, block.timestamp, block.timestamp, 0);
    debtOracle_setLatestRoundDataReturn(1, 1e18, block.timestamp, block.timestamp, 0);

    restakingRouter_depositCollateral(_getActor(), 100e18);
    _logY();

    // Remove rounding error via donation

    asset_mint(address(restakingBondMM), 1);
    _logY();

    console2.log("balance -1", mockERC4626Vault.balanceOf(address(restakingBondMM)));
    console2.log("maxWithdraw -1", mockERC4626Vault.maxWithdraw(address(restakingBondMM)));
    // 1000200000001000198 - 1
    restakingRouter_openBorrowingPosition(2000000000000000000 - 1, type(uint256).max, 365 days);

    _logY();

    console2.log("balance 0", mockERC4626Vault.balanceOf(address(restakingBondMM)));
    console2.log("maxWithdraw 0", mockERC4626Vault.maxWithdraw(address(restakingBondMM)));
    console2.log("previewWithdraw 0",
mockERC4626Vault.previewWithdraw(mockERC4626Vault.maxWithdraw(address(restakingBondMM))));
    console2.log("totalAssets 0", mockERC4626Vault.totalAssets());
    console2.log("totalSupply 0", mockERC4626Vault.totalSupply());

    restakingRouter_addLiquidity(1e18, 0);
}
```

Mitigation

I believe that having the ability to alter the ratio of y and X at the limits is already opening up to breaking the invariant due to precision loss.

Due to this I believe the best mitigation is to ensure that a small but sufficient amount of liquidity is burned permanently into each bond

Meaning y should not be allowed to go below e.g. $1e18$ at all times

This achieves 2 goals:

1. Prevents division by zero
2. Maintains sufficient precision in order to ensure the Invariant correctly prices bonds

H-03 Pendle on Spine Strategy Callbacks trigger the `nonReentrant` guard and revert

Impact

All functions in `RestakingRouter` are `nonReentrant` `closeLeverageBorrowingStrategyEarly` and `repayLeverageStrategy` will call their respective function in the router, then receive a callback.

These callbacks are called in this `nonReentrant` state and call `withdrawCollateral` which is also protected by the `nonReentrant` guard

```
// withdraw collateral
IRestakingRouter(spineRouter).withdrawCollateral(
    _account,
    _collateralToken,
    _poolAddress,
    ptBalance
);
```

<https://github.com/spine-finance/resupplied-sm/blob/d9357ad5f706bfa1ec246009bca7f13512dcfce4/contracts/RestakingRouter.sol#L318-L328>

```
function withdrawCollateral(
    address _account,
    address _collateralToken,
    address _poolAddress,
    uint256 _amount
)
    external
    nonReentrant
    checkPermission(_account)
    CollateralLTVProtected(_account, _poolAddress, _collateralToken)
{
```

This will cause them to revert

Impact

Rewrite the code in the router to be CEI compliant, as to make `withdrawCollateral` not need the reentrancy guards.

M-01 `closeLeverageBorrowingStrategyEarly` slippage check is insufficient

Impact

`closeLeverageBorrowingStrategyEarly` calls `closeBorrowingPositionEarly` with the following params:

<https://github.com/spine-finance/spine-pendle-strategies/blob/a97f465878e61bd33ab663b11050727aa37d1c03/contracts/PendleOnSpineStrategy.sol#L126-L137>

```
// close borrowing position
(uint256 tokenAmountIn, bytes memory callbackRes) = IRestakingRouter(
    spineRouter
).closeBorrowingPositionEarly(
    msg.sender,
    _spineData.poolAddress,
    _pendleData.ptToken,
    _bondAmount,
    _bondAmount, /// @audit 2 scenarios: 1) Sometimes this can revert (although user
rationally should not use it)
    _spineData.maturity, /// 2) This should be provided by the user to be more accurate
    callbackData
);
```

Where the first `_bondAmount` is the amount of debt to be repaid (expressed in bonds), whereas the second `_bondAmount` is the maximum amount of asset used to repay the bonds

The parameter enforces a 1:1 conversion which (excluding fees) should be considered the worst case scenario repayment

Meaning that the slippage check is not tight enough

Mitigation

Have the user pass a slippage check that is computed offchain

M-02 `PendleOnSpineStrategy` is not compatible with `USDT` (use `forceApprove`)

Impact

<https://github.com/spine-finance/spine-pendle-strategies/blob/a97f465878e61bd33ab663b11050727aa37d1c03/contracts/PendleOnSpineStrategy.sol#L279-L280>

```
IERC20(_collateralToken).approve(pendleRouter, 0);  
IERC20(_collateralToken).approve(pendleRouter, ptBalance);
```

Mitigation

Use `forceApprove` like in the rest of the codebase

M-03 Lack of Slippage check in Pendle Strategy

Impact

`onOpenBorrowingPosition` performs a swap from the token to the PT

The swap can incur slippage, and the slippage is hardcoded at 0, meaning any value will be accepted

<https://github.com/spine-finance/spine-pendle-strategies/blob/a97f465878e61bd33ab663b11050727aa37d1c03/contracts/PendleOnSpineStrategy.sol#L219-L227>

```
(uint256 netPtOut, , ) = IPAllActionV3(pendleRouter)
    .swapExactTokenForPt(
        address(this),
        pendleData.market,
        0, /// @audit Slippage check is 0!
        createDefaultApproxParams(), /// @audit could provide offchain guess
        pendleTokenInput,
        createEmptyLimitOrderData()
    );
```

This makes the code vulnerable to front-running and sandwiching

Mitigation

Provide the slippage check as a parameter and pass it in the callbackData

M-04 Pool Creator can permanently DOS the pool by withdrawing all liquidity

Impact

The pool deployer will own 100% of the shares and the assets They can redeem them and cause `y` to become 0

This will cause a revert in `syncReward` <https://github.com/GalloDaSballo/spine-invariants/blob/be99364c41735b22055dff0ece9e2ecce7a2b036/src/RestakingBondMM.sol#L241-L244>

```
y_new += IERC20(underlyingToken).balanceOf(address(this));
uint256 _X = X.intoUint256();
X = ud((_X * y_new) / y);
y = y_new;
```

POC

```
function test_property_y_never_negative_0() public {
    restakingRouter_withdrawLiquidity(1000000000000000000);
    // y will become 0
    // property_y_never_negative();
    restakingRouter_addLiquidity(100e18, 0);
}
```

Mitigation

Either the creator should burn their initial shares, or be prevented from withdrawing them

M-05 Incorrect `lltv` and `ltv` reset for `poolData.tokenAddress`

Impact

`updateCollateralTokenInfo` looks as follows:

<https://github.com/spine-finance/resupplied->

[sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingRouter.sol#L171-L199](https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingRouter.sol#L171-L199)

```
function updateCollateralTokenInfo(
    address _poolAddress,
    address _token,
    CollateralTokenData memory _data
) public onlyAdmin(_poolAddress) {
    PoolData memory poolData = pools[_poolAddress];

    if (_token == poolData.tokenAddress) { /// @audit WHY?
        collateralTokenInfo[_poolAddress][_token].lltv = 0;
        collateralTokenInfo[_poolAddress][_token].ltv = 0;
    } else if (collateralTokenInfo[_poolAddress][_token].lltv == 0) {
        require(
            _data.lltv > 0 &&
            _data.lltv <= TEN_THOUSANDS &&
            _data.ltv <= _data.lltv,
            "Invalid liquidation ratio"
        );
        // new collateral token
        require(
            listCollateralAssets[_poolAddress].length <
            MAX_COLLATERAL_PER_POOL,
            "Collateral list too long"
        );
        listCollateralAssets[_poolAddress].push(_token);
    }
    collateralTokenInfo[_poolAddress][_token] = _data;
    ICollateralVault(poolData.collateralVault).approveToken(_token);
    emit CollateralTokenSet(_poolAddress, _token, _data.ltv, _data.lltv);
}
```

It will set the storage values

```
collateralTokenInfo[_poolAddress][_token].lltv = 0;
collateralTokenInfo[_poolAddress][_token].ltv = 0;
```

To 0

And then pass:

```
collateralTokenInfo[_poolAddress][_token] = _data;
```

This allows the pool admin to pass non-zero values and those values will be used even for the pool asset

Mitigation

Either zero out the fields in the `_data` or revert if those values are non-zero

M-06 `PRICE_FEED_BUFFER_DELAY` is using the wrong units and is only 60 seconds

Impact

The code for `getPrice` looks as follows:

<https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/lib/Router/LoanCalculator.sol#L65-L79>

```
function getPrice(
    address _priceFeed,
    uint256 _heartbeat
) public view returns (uint256) {
    (, int256 price, , uint256 updatedAt, ) = AggregatorV3Interface(
        _priceFeed
    ).latestRoundData();
    require(
        price > 0 &&
        block.timestamp - updatedAt <=
            _heartbeat + PRICE_FEED_BUFFER_DELAY,
        "Price feed is stale"
    );
    return uint256(price);
}
```

It will use `PRICE_FEED_BUFFER_DELAY` which is said to be 1 hour:

<https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/lib/Router/LoanCalculator.sol#L9>

```
uint256 constant PRICE_FEED_BUFFER_DELAY = 60; // 1 hour
```

However the units are in seconds, meaning this is just a 60 second delay

Mitigation

Use $60 * 60 = 3600$

M-07 `RestakingRouter. openLendingPosition` is computing the wrong equity risk

Impact

The function `openLendingPosition` has the following check:

[https://github.com/spine-finance/resupplied-](https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingRouter.sol#L547-L553)

[sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingRouter.sol#L547-L553](https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingRouter.sol#L547-L553)

```
uint256 minE = IRestakingBondMM(_poolAddress).getEquity();
require( /// @audit Could we cycle in some way and bypass this?
    minE + _tokenAmountIn - bondAmount >=
        (poolData.equityRiskRatio * poolData.lpDepositAmount) /
        TEN_THOUSANDS,
    "Reach equity risk"
); /// @audit Equity math is wrong!!
```

This check is meant to ensure that LPs are not losing too much equity due to how the AMM mints bonds to lenders (it mints more bonds than the cash they provide as to pay their interest)

The check intuitively adds the `_tokenAmountIn` (cash added) and subtracts the `bondAmount` minted (which is typically higher than cash)

However, `openLendingPosition` calls `swapQuoteTokenForBond` which already updates the equity (X and y) and mints the bonds

Meaning that a call to `IRestakingBondMM(_poolAddress).getEquity();` will return the already updated equity

Mitigation

Change the formula to use `uint256 currentE = IRestakingBondMM(_poolAddress).getEquity();`

```
currentE >= * (poolData.equityRiskRatio * poolData.lpDepositAmount) /
            TEN_THOUSANDS
```

Since equity is already updated with the deposit at that point in the code

M-08 Fee math is inconsistent when using `amountIn` vs `amountOut`

Impact

Assessing a fee on amount in vs amount out is inconsistent if you use the same formula

Throughout the code, fees are assessed in the following way: [https://github.com/spine-](https://github.com/spine-finance/resupplied-)

[sm/blob/4fc85b1944f8c4f901bbbf218839b6a2a5f4e9e/contracts/RestakingBondMM.sol#L346-L347](https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbf218839b6a2a5f4e9e/contracts/RestakingBondMM.sol#L346-L347)

```
uint256 fee = _getFee(_maturity);  
uint256 swapFee = (uint_delta_y * fee) / TEN_THOUSANDS; /// @audit Shouldn't this round up?
```

This is fine for computing amounts in, however when used to compute amounts out (given in), you'll end up having inconsistencies, specifically you'll be overcharging the fee

POC

Imagine Bond and Asset at 1:1 ratio I have 100 Bonds and 100 Assets

I put 100 Asset and I should receive 100 bonds I pay a 1% fee I receive 99 bonds


$100/99 = 1.0101010101$

The fee is higher than 1%!

Vice verse

I want to receive 100 bonds I have to pay 100 Asset We then add the fee of 1% I have to pay 101 assets and I'll receive 100 bonds

$101/100 = 1.01$, I paid a 1% fee!

 Image

Mitigation

Consider using formulas similar to the already audited eBTC BSM: <https://github.com/ebtc-protocol/ebtc-bsm/blob/e5935b77a19a1b6036a3a1b44f70ac128fbdf7d9/src/EbtcBSM.sol#L114-L129>

M-09 The way fee impacts the Invariant is Inconsistent leading to repricing of bonds

Impact

Per discussion with the dev, the BondMM should handle a swap in the following way:

- Update the invariant without fees
- Leave the fees as the delta between the balance and `y`
- Fees will be processed on the next call to `syncReward`

The alternative would be that fees would alter the relation between X and y causing the invariant to be altered in a step-wise way (as the fees are a static %)

Whereas adding them after, and handling them as yield, is consistent with how yield from rehypothecation works (which increases the absolute values of X and y but keeps their ratio constant)

swapBondForQuoteToken - includes fee in `delta_y` - Inconsistent!

Computes `delta_y` which includes fees

<https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingBondMM.sol#L339-L349>

```
UD60x18 delta_y = poolState.y -
    (poolState.K.mul(poolState.x.pow(poolState.alpha)) +
     poolState.y.pow(poolState.alpha) -
     poolState.K.mul((poolState.x + delta_x).pow(poolState.alpha)))
    .pow(poolState.alpha.inv());

uint256 uint_delta_y = delta_y.intoUint256();
uint256 fee = _getFee(_maturity);
uint256 swapFee = (uint_delta_y * fee) / TEN_THOUSANDS; /// @audit Shouldn't this round up?
amountOut = uint_delta_y - swapFee;
_updateXY(poolState.y, delta_y, 1, poolState.alpha); /// @audit delta_y includes fees
```

This means that the invariant is being updated while including the fee, instead of excluding it

swapQuoteTokenForExactBond - Consistent

<https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingBondMM.sol#L370-L379>

```

UD60x18 delta_y = (poolState.K.mul(poolState.x.pow(poolState.alpha)) +
    poolState.y.pow(poolState.alpha) -
    poolState.K.mul((poolState.x - delta_x).pow(poolState.alpha))).pow(
    poolState.alpha.inv()
    ) - poolState.y;
uint256 uint_delta_y = delta_y.intoUint256();
uint256 fee = _getFee(_maturity);
uint256 swapFee = (uint_delta_y * fee) / TEN_THOUSANDS;
amountIn = uint_delta_y + swapFee;
_updateXY(poolState.y, delta_y, 0, poolState.alpha); /// @audit delta_y does NOT include the fee

```

The function doesn't include fee in `delta_y`, it is transferred to here (and prob accounted as yield in the next syncReward)

This is consistent with the stated behaviour

swapQuoteTokenForBond – Consistent

<https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingBondMM.sol#L397-L408>

```

UD60x18 delta_y = ud(_amountIn);
delta_y = delta_y - ud(swapFee); /// @audit Swap fee changes X/y
UD60x18 delta_x = poolState.x -
    (
        (poolState.K.mul(poolState.x.pow(poolState.alpha)) +
            poolState.y.pow(poolState.alpha) -
            (poolState.y + delta_y).pow(poolState.alpha)).div(
                poolState.K
            )
        ).pow(poolState.alpha.inv());
amountOut = delta_x.intoUint256();
_updateXY(poolState.y, delta_y, 0, poolState.alpha); /// @audit delta_y does NOT include fees

```

Removes the fee explicitly (and prob accounted as yield in the next syncReward)

swapBondForExactQuoteToken – Inconsistent!

<https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingBondMM.sol#L426-L434>

```

UD60x18 delta_y = ud(_amountOut);
delta_y = delta_y + ud(swapFee);
UD60x18 delta_x = (
    (poolState.K.mul(poolState.x.pow(poolState.alpha)) +
        poolState.y.pow(poolState.alpha) -
        (poolState.y - delta_y).pow(poolState.alpha)).div(poolState.K)
    ).pow(poolState.alpha.inv()) - poolState.x;
amountIn = delta_x.intoUint256();
_updateXY(poolState.y, delta_y, 1, poolState.alpha);

```

Adds the fee explicitly in `delta_y`, breaking the intended behaviour

Mitigation

Change `swapBondForExactQuoteToken` and `swapBondForQuoteToken` to use `delta_y` without the fee

```
_updateXY(poolState.y, delta_y - ud(swapFee), 1, poolState.alpha);
```

M-10 `handleCashOut` Loss should be imputed to receiver, not the system, risk of loss socialization

Impact

`handleCashOut` looks as follows:

<https://github.com/spine-finance/resupplied->

[sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingBondMM.sol#L556-L599](https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingBondMM.sol#L556-L599)

```
function handleCashOut( /// @audit TODO: These all socialize losses
    address externalLPRouter,
    uint _tokenAmountOut,
    address _to
) external onlyRouter {
    uint256 underlyingBalance = IERC20(underlyingToken).balanceOf(
        address(this)
    );
    if (underlyingBalance > 0) {
        if (underlyingBalance >= _tokenAmountOut) {
            // enough underlying token => transfer then exit
            IERC20(underlyingToken).safeTransfer(_to, _tokenAmountOut);
            return;
        } else {
            // not enough underlying token => transfer all then withdraw from externalPool
            IERC20(underlyingToken).safeTransfer(_to, underlyingBalance);
            _tokenAmountOut -= underlyingBalance;
        }
    }
    if (poolType == RESTAKING_POOL_TYPE.AAVE) {
        IPool(externalLPRouter).withdraw(
            underlyingToken,
            _tokenAmountOut,
            _to
        );
    } else if (poolType == RESTAKING_POOL_TYPE.VENUS) {
        VToken(quoteToken).redeemUnderlying(_tokenAmountOut);
        IERC20(underlyingToken).safeTransfer(_to, _tokenAmountOut);
    } else if (poolType == RESTAKING_POOL_TYPE.MORPHO) {
        IMorphoVault(quoteToken).withdraw(
            _tokenAmountOut,
            _to,
            address(this)
        );
    } else if (poolType == RESTAKING_POOL_TYPE.EULER) {
        IEulerVault(quoteToken).withdraw(
            _tokenAmountOut,
            _to,
            address(this)
        );
    } else {
        revert("DO NOT SUPPORT THIS PROTOCOL");
    }
}
```


When calling `withdraw` we will burn `previewWithdraw` shares which is rounded up by one, this causes a (typically) small loss to the Bond and other LPs. In some scenarios the loss can be big.

It would be more appropriate to impute the loss to the caller.

Mitigation

After determining how much to withdraw:

- Use `ERC4626.convertToShares` to determine how many shares to use (this rounds down the value so the caller will lose instead of the system)
- Proceed to redeem those shares
- Return at most the original amount, keep the rest in the vault

You can see an example of similar logic in another project I worked on:

<https://github.com/ebtc-protocol/ebtc->

[bsm/blob/e5935b77a19a1b6036a3a1b44f70ac128fbd7d9/src/ERC4626Escrow.sol#L67-L102](https://github.com/ebtc-protocol/ebtc-/blob/e5935b77a19a1b6036a3a1b44f70ac128fbd7d9/src/ERC4626Escrow.sol#L67-L102)

M-11 The system will not work with USDT or other tokens that do not return a `bool` on `approve`

Impact

In various parts of the codebase the following approval pattern is used:

<https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/vaults/CollateralVault.sol#L20-L23>

```
function approveToken(address token) external onlyRouter {  
    IERC20(token).approve(msg.sender, 0);  
    IERC20(token).approve(msg.sender, MAX_INT);  
}
```

The IERC20 interface expects a `bool` to be returned by the call

This causes the solidity compiler to add a check for it.

A call that results in no return value will revert at that check

USDT on mainnet has the following implementation for approve:

```
// Forward ERC20 methods to upgraded contract if this one is deprecated  
function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {  
    if (deprecated) {  
        return UpgradedStandardToken(upgradedAddress).approveByLegacy(msg.sender, _spender, _value);  
    } else {  
        return super.approve(_spender, _value);  
    }  
}
```

<https://etherscan.io/address/0xdac17f958d2ee523a2206206994597c13d831ec7#code#L368>

Which will cause the system to not work with USDT

Mitigation

Use `forceApprove` which also handles the non-zero to non-zero allowance changes

M-12 Venus Integration leaks yield due to using `exchangeRateStored`

Impact

`syncReward` is meant to update the latest `y` as a means to ensure that no value is leaked

When restaking is done on Venus, we have the following code:

<https://github.com/spine-finance/resupplied->

[sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingBondMM.sol#L238-L241](https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingBondMM.sol#L238-L241)

```
} else if (poolType == RESTAKING_POOL_TYPE.VENUS) {
    uint exchangeRate = VToken(quoteToken).exchangeRateStored();
    uint vTokenAmount = IERC20(quoteToken).balanceOf(address(this));
    y_new = (vTokenAmount * exchangeRate) / 10 ** 18;
```

Per Venus' documentation, this value is stale

<https://github.com/VenusProtocol/venus->

[protocol/blob/8f9fc162e0a924d66302f0d7186a7c08f0c3ef8c/contracts/Tokens/VTokens/VToken.sol#L570](https://github.com/VenusProtocol/venus-protocol/blob/8f9fc162e0a924d66302f0d7186a7c08f0c3ef8c/contracts/Tokens/VTokens/VToken.sol#L570)

```
/**
 * @notice Calculates the exchange rate from the underlying to the VToken
 * @dev This function does not accrue interest before calculating the exchange rate
 * @return Calculated exchange rate scaled by 1e18
 */
```

This will cause new LPs to receive more shares than intended, as the yield from Venus will be socialized to new depositors until an actual accrue happens

POC

- Victim provide liquidity and their tokens are rehypothecated
- Yield accrues
- Exchange rate is cached and doesn't update
- Attacker provides new liquidity via a flashloan, they own 99.99% of the LP
- Attacker accrues the Venus Token
- Attacker withdraws, they have received 99.99% of the yield generated by the Victim

Mitigation

Use `exchangeRateCurrent`

Q-01 Interest rate arbitrage was possible when `handleCashOut` was causing a loss to the system and not the caller

Impact

Please note that I found this after the end of the review via invariant tests, so I haven't spent a lot of time on this

POC

```
// forge test --match-test test_doomsday_open_close_borrow_arbitrage_0 -vvv

function test_doomsday_open_close_borrow_arbitrage_0() public {

    vm.roll(50268);
    vm.warp(430306);

    mockERC4626Tester_setLossOnWithdraw(86271829334882047342934448278462818155638862152130183000728819336316
0);

    vm.roll(50268);
    vm.warp(430306);
    restakingRouter_depositCollateral(9468997301679750019);

    vm.roll(50268);
    vm.warp(430306);
    doomsday_open_close_borrow_arbitrage(72274330671842167,
356025697327532428319760045793444193138440349384122242696504728017542235);
}
```

Mitigation

Investigate the root cause and determine if this behaviour is intended.

It seems like a loss would cause borrowing to be cheaper due to reducing the `y`, however I would expect the loss to supersede the gain from the interest rate.

Q-02 `closeLeverageBorrowingStrategyEarly` is approving an incorrect amount of tokens

Impact

`closeBorrowingPositionEarly` should in general cost less than `repay`. Although in some scenarios it can cost more (similar price + fees causing it to cost more than repay).

<https://github.com/spine-finance/spine-pendle-strategies/blob/a97f465878e61bd33ab663b11050727aa37d1c03/contracts/PendleOnSpineStrategy.sol#L122-L137>

```
IERC20(_spineData.poolToken).approve(spineRouter, 0);
IERC20(_spineData.poolToken).approve(spineRouter, _bondAmount);

bytes memory callbackData = abi.encode(_pendleData, _pendleTokenOutput);
// close borrowing position
(uint256 tokenAmountIn, bytes memory callbackRes) = IRestakingRouter(
    spineRouter
).closeBorrowingPositionEarly(
    msg.sender,
    _spineData.poolAddress,
    _pendleData.ptToken,
    _bondAmount,
    _bondAmount, /// @audit 2 scenarios: 1) Sometimes this can revert (although user
rationally should not use it)
    _spineData.maturity, /// 2) This should be provided by the user to be more accurate
    callbackData
);
```

The check: <https://github.com/spine-finance/spine-pendle-strategies/blob/a97f465878e61bd33ab663b11050727aa37d1c03/contracts/PendleOnSpineStrategy.sol#L122-L123>

```
IERC20(_spineData.poolToken).approve(spineRouter, 0);
IERC20(_spineData.poolToken).approve(spineRouter, _bondAmount);
```

Hardcodes `_bondAmount` when in reality the user should pay less

Mitigation

It's probably fine to leave the code as is, however you should comment that the amount is incorrect

Q-03 `_pool.tokenAddress` should never be a token that can rebase (vaults)

Impact

`closeBorrowingPositionEarly` is forced to defer the call to `_handleCashIn(_poolAddress, tokenAmountIn)`;

<https://github.com/GalloDaSballo/spine->

[invariants/blob/1d7b25237788606cafcdddc1154ab0e18ecb19c2/src/RestakingRouter.sol#L311-L351](https://github.com/GalloDaSballo/spine-invariants/blob/1d7b25237788606cafcdddc1154ab0e18ecb19c2/src/RestakingRouter.sol#L311-L351)

```
function closeBorrowingPositionEarly(
    address _account,
    address _poolAddress,
    address _collateralToken,
    uint256 _bondAmount,
    uint256 _maxTokenAmountIn,
    uint256 _maturity,
    bytes memory _callbackData
)
    external
    nonReentrant
    checkPermission(_account)
    syncReward(_poolAddress)
    checkValidAmountPerAction(_poolAddress, _bondAmount)
    returns (uint256 tokenAmountIn, bytes memory callbackRes)
{
    require(_bondAmount > 0, "invalid amount");

    // swap cash to bond
    PoolData memory poolData = pools[_poolAddress];
    require(poolData.created, "Pool must be created");
    tokenAmountIn = IRestakingBondMM(_poolAddress).swapQuoteTokenForExactBond(_bondAmount,
    _maturity, ACTION.CB);
    require(tokenAmountIn <= _maxTokenAmountIn, "Exceed max token amount");

    bytes32 userBorrowedKey = LoanCalculator.getBorrowedKey(_account, _poolAddress,
    _collateralToken, _maturity);

    userBorrowed[userBorrowedKey] -= _bondAmount;

    if (_callbackData.length > 0) {
        callbackRes = IStrategies(msg.sender).onCloseBorrowingPositionEarly(
            _account, _poolAddress, _collateralToken, _bondAmount, _maturity, _callbackData,
tokenAmountIn
        );
    }
    _handleCashIn(_poolAddress, tokenAmountIn);

    require(
        userBorrowed[userBorrowedKey] > poolData.minAmountPerAction || userBorrowed[userBorrowedKey]
== 0,
        "Invalid remaining bond amount"
    );
    emit BorrowingPositionClosedEarly(_account, _poolAddress, _maturity, tokenAmountIn,
    _bondAmount);
}
```

This allows a pattern in which a user can withdraw their collateral and use it to repay the debt

I was unable to weaponize this with the current config

However, it's worth noting that the safety of the system relies on the fact that the cached `tokenAmountIn`'s value is expected not to change

Therefore, using a `_pool.tokenAddress` that is itself a vault (e.g. staked USD*) could result in a loss of value to the system

POC

- sUSDx is the Collateral and the system + the user own all of the supply
- The share value is $1.3e18$
- The user `closeBorrowingPositionEarly`
- They now have 100% of the sUSDx total supply
- They withdraw from the vault and burn all supply
- This resets the Price Per Share back to $1e18$
- The check will then transfer back the same amount
- Because the vault price has changed, the amount is no longer worth the same

Note that this also requires the oracle being used to not reprice Which may be unrealistic

Q-04 `PendleOnSpineStrategy` risky patterns that can be fixed

Make `openBorrowingPosition` CEI Conformant and safer

<https://github.com/GalloDaSballo/spine->

[invariants/blob/1d7b25237788606cafcdddc1154ab0e18ecb19c2/src/RestakingRouter.sol#L301-L308](https://github.com/GalloDaSballo/spine-invariants/blob/1d7b25237788606cafcdddc1154ab0e18ecb19c2/src/RestakingRouter.sol#L301-L308)

```
    _handleCashOut(_poolAddress, _tokenAmountOut, _account);
    bytes32 userBorrowedKey = LoanCalculator.getBorrowedKey(_account, _poolAddress,
    _collateralToken, _maturity);
    if (_callbackData.length > 0) {
        callbackRes = IStrategies(msg.sender).onOpenBorrowingPosition(
            _account, _poolAddress, _collateralToken, _tokenAmountOut, _maturity, _callbackData,
bondAmount
        );
    }
    userBorrowed[userBorrowedKey] += bondAmount;
    emit BorrowingPositionOpened(_account, _poolAddress, _maturity, _tokenAmountOut, bondAmount);
```

You can move the effects:

```
    userBorrowed[userBorrowedKey] += bondAmount;
    emit BorrowingPositionOpened(_account, _poolAddress, _maturity, _tokenAmountOut, bondAmount);
    _handleCashOut(_poolAddress, _tokenAmountOut, _account);
```

To happen before the token transfer and the callback

Since the `PendleOnSpineStrategy` will deposit coll (which has no solvency check), you can already add the debit to the account, which makes the code more consistent

Q-05 `onOpenBorrowingPosition` could use a offchain computed hint for efficiency

Impact

<https://github.com/spine-finance/spine-pendle-strategies/blob/a97f465878e61bd33ab663b11050727aa37d1c03/contracts/PendleOnSpineStrategy.sol#L219-L227>

```
(uint256 netPtOut, , ) = IPAllActionV3(pendleRouter)
    .swapExactTokenForPt(
        address(this),
        pendleData.market,
        0, /// @audit Slippage check is 0!
        createDefaultApproxParams(), /// @audit could provide offchain guess
        pendleTokenInput,
        createEmptyLimitOrderData()
    );
```

Mitigation

You can customize the approxParams:

<https://docs.pendle.finance/Developers/Contracts/PendleRouter#approxparams>

Which should save gas in the best case and have no impact in the worst case

Q-06 Gas & QA spine-pendle-strategies

Make immutable

<https://github.com/spine-finance/spine-pendle-strategies/blob/a97f465878e61bd33ab663b11050727aa37d1c03/contracts/PendleOnSpineStrategy.sol#L14-L15>

```
address public pendleRouter;  
address public spineRouter;
```

Move to the start (style guide)

<https://github.com/spine-finance/spine-pendle-strategies/blob/a97f465878e61bd33ab663b11050727aa37d1c03/contracts/PendleOnSpineStrategy.sol#L50>

```
using SafeERC20 for IERC20;
```

Q-07 `removeCollateralToken` doesn't clear allowances

Impact

Adding a collateral token grants approval to the `poolData.collateralVault`

<https://github.com/spine-finance/resupplied-sm/blob/a817417d50fdb7029525b5a511d477b7dc93a6c5/contracts/RestakingRouter.sol#L179-L180>

```
ICollateralVault(poolData.collateralVault).approveToken(_token);
```

The allowance is not revoked in `removeCollateralToken`

<https://github.com/spine-finance/resupplied-sm/blob/a817417d50fdb7029525b5a511d477b7dc93a6c5/contracts/RestakingRouter.sol#L183-L210>

```
function removeCollateralToken(
    address _poolAddress,
    address _token
) external onlyAdmin(_poolAddress) {
    address[] storage listPoolCollateralAssets = listCollateralAssets[
        _poolAddress
    ];
    for (uint i = 0; i < listPoolCollateralAssets.length; i++) {
        if (listPoolCollateralAssets[i] == _token) {
            uint lastIndex = listPoolCollateralAssets.length - 1;
            if (i != lastIndex) {
                listPoolCollateralAssets[i] = listPoolCollateralAssets[
                    lastIndex
                ];
            }
            listPoolCollateralAssets.pop();
            break;
        }
    }
    delete collateralTokenInfo[_poolAddress][_token]
        .priceFeedData
        .priceFeed;
    delete collateralTokenInfo[_poolAddress][_token]
        .priceFeedData
        .heartbeat;
    delete collateralTokenInfo[_poolAddress][_token].lltv;
    delete collateralTokenInfo[_poolAddress][_token].ltv;
}
```

Q-08 Bond data can be deleted before all bonds have been redeemed

Impact

`cleanOldMaturities` allows deleting the data for old maturities

These may still have a non-zero supply, and in spite of being deprecated that can impact the pricing of newer bonds.

Deleting the maturities also permanently prevents users from redeeming the bonds

POC

```
// forge test --match-test test_property_basic_maturity_total_supply_0 -vvv
function test_property_basic_maturity_total_supply_0() public {

    switchActor(23946281263061655376086291792190763952000858539490433);

    restakingRouter_openLendingPosition(1099536285697, 1071653891,
138834251709356845004279828504697896864601708559270828216354369);

    switchActor(115792089237316195420432434140994567471352589954036730831378634385090486125892);

    vm.roll(1522830);
    vm.warp(18287501);
    restakingRouter_cleanOldMaturities();

    vm.roll(1522830);
    vm.warp(18287501);
    property_basic_maturity_total_supply(); /// @audit Total supply is non-zero but `(, uint256 lent) =
restakingBondMM_loanData(maturities[i]) is 0 due to deletion`
}
```

Mitigation

Setup monitoring to prevent this from happening.

Alternatively you could prevent maturities from being deleted when their total supply is non-zero, this could cause maturities to never be deleted.

Given the fact that you have a max maturity this seems like an acceptable tradeoff.

Q-09 Informational: You can open a borrowing position on zero lent due to first liquidity provider

Impact

The first depositor provides liquidity and sets the rate

Their liquidity can be borrowed against

In spite of the fact that it's not part of the `l` storage variable

This breaks a key invariant that you can never borrow more than what is lent

POC

```
function test_property_borrow_lent_soundness_1() public {  
  
    vm.roll(59268);  
    vm.warp(890774);  
    restakingRouter_depositCollateral(19200488702525766297);  
  
    vm.roll(59268);  
    vm.warp(890774);  
    restakingRouter_openBorrowingPosition(8796629908532,  
354516361703819870745436027198490339042248062002959503993920,  
3450873173395281894465666769451935020345841120435943443889280717403265);  
  
    vm.roll(59268);  
    vm.warp(890774);  
    property_borrow_lent_soundness();  
}
```

Q-10 Inaccurate `lpDepositAmount` causes `equityRiskRatio` math to underestimate the available liquidity

Impact

`openLendingPosition` has the following check:

<https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingRouter.sol#L549-L554>

```
require(
    minE + _tokenAmountIn - bondAmount >=
        (poolData.equityRiskRatio * poolData.lpDepositAmount) /
            TEN_THOUSANDS,
    "Reach equity risk"
);
```

This will compute a percentage of `lpDepositAmount` as a means to determine if the LPs are taking too much of a loss due to lender payouts.

Adding 1 unit of asset increases `lpDepositAmount` linearly

On `withdrawLiquidity` the amount is changed in this way:

<https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingRouter.sol#L658-L662>

```
if (tokenAmountOut > pools[_poolAddress].lpDepositAmount) {
    pools[_poolAddress].lpDepositAmount = 0;
} else {
    pools[_poolAddress].lpDepositAmount -= tokenAmountOut;
}
```

where `tokenAmountOut` is the pro-rata asset received from the pool.

Because of:

- Yield bearing deposits
- Swap Fees

LPs deposits are expected to grow in value

This means that for each unit of asset deposited (which increases `lpDepositAmount`) the LP may withdraw more `tokenAmountOut`

Due to this, the formula in `openLendingPosition` will underestimate the amount of lending positions that can be opened

Mitigation

From a risk perspective, the inaccuracy is not particularly dangerous, meaning it may be best to document this and if necessary adjust the `poolData.equityRiskRatio` to account for the inaccuracy

I believe you could try the following:

- On gain (`syncReward`) you'd add the extra rewards to `lpDepositAmount` this should ensure that the value is tracked more appropriately

Q-11 Precision loss of y/X when y is very small could be used to break the invariant and reprice bonds

Impact

X/y can have rounding error due to truncation

These are typically negligible, however if X/y are very small, then the rounding error can become very significant

This may be weaponizeable to alter rates drastically when there's very little liquidity

Mitigation

Prevent y and X from ever reaching a value that would lose too much precision

Q-12 UX Improvement – Make it easier to fetch deployment info

Impact

`pools` is not public which prevents us from fetching the pool state and config

Mitigation

Make `pools` public

Q-13 `poolData.vault` is provided but never used

Impact

`handleInitNewPool` has a parameter `vault`

This is passed to `RestakingBondMM` via `initPool`

However, the value `vault` is `RestakingBondMM` is never used

Mitigation

Delete the unused value

Q-14 `RESTAKING_POOL_TYPE.NONE` is not supported

Impact

The Enum `RESTAKING_POOL_TYPE` allows the option `NONE`

We can provide this value and cashing out is correctly handled But depositing in reverts every time

```
| | | | emit Transfer(from: CryticToFoundry: [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496],
to: RestakingRouter: [0x1d1499e622D69689cdf9004d05Ec547d650Ff211], value: 10000000000000000 [1e18])
| | | |   ↳ [Return] true
| | | |   ↳ [Revert] DO NOT SUPPORT THIS PROTOCOL
| | | |   ↳ [Revert] DO NOT SUPPORT THIS PROTOCOL
| | | |   ↳ [Revert] DO NOT SUPPORT THIS PROTOCOL
```

Mitigation

Consider whether you should support a no-rehypothecation version or remove the option from the Enum

Q-15 `lltv` is validated only when first adding a collateral

Impact

`updateCollateralTokenInfo` looks as follows:

[https://github.com/spine-finance/resupplied-](https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingRouter.sol#L171-L199)

[sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingRouter.sol#L171-L199](https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingRouter.sol#L171-L199)

```
function updateCollateralTokenInfo(
    address _poolAddress,
    address _token,
    CollateralTokenData memory _data
) public onlyAdmin(_poolAddress) {
    PoolData memory poolData = pools[_poolAddress];

    if (_token == poolData.tokenAddress) { /// @audit WHY?
        collateralTokenInfo[_poolAddress][_token].lltv = 0;
        collateralTokenInfo[_poolAddress][_token].ltv = 0;
    } else if (collateralTokenInfo[_poolAddress][_token].lltv == 0) {
        require(
            _data.lltv > 0 &&
            _data.lltv <= TEN_THOUSANDS &&
            _data.ltv <= _data.lltv,
            "Invalid liquidation ratio"
        );
        // new collateral token
        require(
            listCollateralAssets[_poolAddress].length <
            MAX_COLLATERAL_PER_POOL,
            "Collateral list too long"
        );
        listCollateralAssets[_poolAddress].push(_token);
    }
    collateralTokenInfo[_poolAddress][_token] = _data;
    ICollateralVault(poolData.collateralVault).approveToken(_token);
    emit CollateralTokenSet(_poolAddress, _token, _data.ltv, _data.lltv);
}
```

A key soundness check for LTV and LLTV is:

```
require(
    _data.lltv > 0 &&
    _data.lltv <= TEN_THOUSANDS &&
    _data.ltv <= _data.lltv,
    "Invalid liquidation ratio"
);
```

Because this is performed in:

```
} else if (collateralTokenInfo[_poolAddress][_token].lltv == 0) {
```

It will only be performed the first time, possibly allowing the admin, by mistake or willingly, to set unsound values

Mitigation

Simplify the function to perform the checks at all time

Q-16 AMM Fees should be rounded up as they are consistently undercharged

Impact

The fees are computed with a high precision and then divided by `1e18`, which causes them to be rounded down.

Because the config allows a `minAmount` in many cases fees will always be non-zero.

However it's worth noting that:

If you set `basedFee` to `1` bps it will take 53 weeks before the fee is `1`

26 weeks + 1 week if you set it to `2`

I believe it's probably best to use a higher precision as otherwise fees won't be charged unless the AMM is selling very long maturities

```
uint256 constant ONE = 1e18;

function test_getFee_ud() public {
    uint256 t = 52 weeks + 1 weeks;
    uint256 basedFee = 1; // 1 / 1e4 = 0.0001
    console2.log("getFee", _getFee_ud(t, basedFee));
}

uint256 constant YEAR_SECONDS = 31536000;

function getTimeToMaturity(
    uint256 _seconds
) internal view returns (UD60x18) {
    return ud(_seconds).div(ud(YEAR_SECONDS)); /// @audit TODO: Check this cause it looks odd
}

function _getFee_ud(uint256 _seconds, uint256 basedFee) private view returns (uint256) {
    UD60x18 t = getTimeToMaturity(_seconds);
    console2.log("t", t.intoUint256());
    UD60x18 intermediary_result = (t * ud(basedFee * ONE));
    console2.log("intermediary_result", intermediary_result.intoUint256());
    uint256 fee = intermediary_result.intoUint256() / ONE;

    return fee;
}
```

```
[PASS] test_getFee_ud() (gas: 11700)
Logs:
t 1016438356164383561
intermediary_result 1016438356164383561
getFee 1
```

Q-17 Euler deposits can be forced to revert to ensure that they do not earn yield

Impact

The code for `handleCashIn` looks as follows:

<https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/lib/Router/ActionHelper.sol#L73-L87>

```
    } else if (_pool.restakingType == RESTAKING_POOL_TYPE.EULER) {
        IERC20(tokenAddress).approve(_pool.stakingTokenAddress, 0);
        IERC20(tokenAddress).approve(
            _pool.stakingTokenAddress,
            _tokenAmountIn
        );
        try
            IEulerVault(_pool.stakingTokenAddress).deposit(
                _tokenAmountIn,
                _poolAddress
            )
        {} catch {
            // If deposit fails, we transfer the tokens directly to the pool address
            IERC20(tokenAddress).safeTransfer(_poolAddress, _tokenAmountIn);
        }
    }
```

The EVK allows flashloans:

<https://github.com/euler-xyz/euler-vault-kit/blob/5b98b42048ba11ae82fb62dfec06d1010c8e41e6/src/EVault/modules/Borrowing.sol#L145-L158>

```
/// @inheritdoc IBorrowing
function flashLoan(uint256 amount, bytes calldata data) public virtual nonReentrant {
    address account = EVCAuthenticate();
    callHook(vaultStorage.hookedOps, OP_FLASHLOAN, account);

    (IERC20 asset,,) = ProxyUtils.metadata();

    uint256 origBalance = asset.balanceOf(address(this));

    asset.safeTransfer(account, amount);

    IFlashLoan(account).onFlashLoan(data);

    if (asset.balanceOf(address(this)) < origBalance) revert E_FlashLoanNotRepaid();
}
```

<https://github.com/euler-xyz/euler-vault-kit/blob/5b98b42048ba11ae82fb62dfec06d1010c8e41e6/src/EVault/modules/Vault.sol#L124-L136>

```

function deposit(uint256 amount, address receiver) public virtual nonReentrant returns (uint256) {
    (VaultCache memory vaultCache, address account) = initOperation(OP_DEPOSIT, CHECKACCOUNT_NONE);

    Assets assets = amount == type(uint256).max ? vaultCache.asset.balanceOf(account).toAssets() :
amount.toAssets();
    if (assets.isZero()) return 0;

    Shares shares = assets.toSharesDown(vaultCache);
    if (shares.isZero()) revert E_ZeroShares();

    finalizeDeposit(vaultCache, assets, shares, account, receiver);

    return shares.toUint();
}

```

This means that a deposit can take a flashloan, trigger the `nonReentrant` guard, then cause the code in the `handleCashIn` to revert, causing all deposits to be performed in the underlying asset

Mitigation

Consider whether you need the `try/catch` at all, as ultimately you may want to deprecate a market when the rehypothecation target is deprecated or paused

Q-18 CEI Breaking Code

Impact

The codebase is meant to be used with arbitrary tokens

Whenever that is the case we should be mindful of tokens that have hooks on transfer

CEI (Check Effect Interactions) is a pattern that makes the code safe against hooks as ultimately you are possibly giving out control to another contract only after all relevant state changes have been performed

There are scenarios in which more than one external call is performed, for those cases, we can reorder operations based on "trust", e.g. a contract that we deploy is more trusted than a token.

Following this principle, we can reorganize some parts of the code to minimize external integration risks:

Instances

<https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingRouter.sol#L652-L662>

```
tokenAmountOut = IRestakingBondMM(_poolAddress).withdrawLiquidity(
    msg.sender,
    _shares
);
_handleCashOut(_poolAddress, tokenAmountOut, msg.sender);

if (tokenAmountOut > pools[_poolAddress].lpDepositAmount) {
    pools[_poolAddress].lpDepositAmount = 0;
} else {
    pools[_poolAddress].lpDepositAmount -= tokenAmountOut;
}
```

-> Should update the `lpDepositAmount` before `_handleCashOut`

<https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingRouter.sol#L783-L786>

```
_handleCashIn(_poolAddress, totalPaid); /// NOTE: Pay in token
uint256 totalPaidWithLiquidationFee = (totalPaid *
    (TEN_THOUSANDS + poolData.liquidatedFee)) / TEN_THOUSANDS;
```

`_handleCashIn` should happen after all storage changes

<https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingRouter.sol#L872-L876>

```
_handleCashIn(_poolAddress, _amount);
IRestakingBondMM(_poolAddress).repay(_amount, _maturity);
userBorrowed[userBorrowedKey] -= _amount;
emit PositionRepaid(_account, _poolAddress, _amount, _maturity);
return _amount;
```

<https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbf218839b6a2a5f4e9e/contracts/RestakingRouter.sol#L872-L874>

```
_handleCashIn(_poolAddress, _amount);
IRestakingBondMM(_poolAddress).repay(_amount, _maturity);
userBorrowed[userBorrowedKey] -= _amount;
```

`_handleCashIn` should happen last as it uses a token that may not be as trusted as the other contracts

Q-19 Off by one error in `handlePaymentForSingleMaturity`

Impact

The function `handlePaymentForSingleMaturity` looks as follows:

<https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/lib/Router/ActionHelper.sol#L154-L175>

```
function handlePaymentForSingleMaturity(
    address _poolAddress,
    uint256 _maturity,
    uint256 _borrowedAmount
) public returns (uint tokenAmount) {
    if (_borrowedAmount > 0) {
        if (block.timestamp <= _maturity) { /// @audit tokenAmount >=
            tokenAmount = IRestakingBondMM(_poolAddress)
                .swapQuoteTokenForExactBond(
                    _borrowedAmount,
                    _maturity,
                    ACTION.CB
                );
        } else {
            tokenAmount = _borrowedAmount;
            IRestakingBondMM(_poolAddress).repay(
                _borrowedAmount,
                _maturity
            );
        }
    }
}
```

At maturity (`block.timestamp - maturity == 0`), the function will use `swapQuoteTokenForExactBond` using the same price as in `repay` but charging fees

This will impact the ratio between the value of bonds and the cash

It will also cause the liquidator to pay an additional fee, possibly making the liquidation unprofitable or less desirable

Mitigation

Use `if (block.timestamp < _maturity) {`

Also see: <https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingBondMM.sol#L314-L324>

```

function repay(uint256 _cashIn, uint256 _maturity) external onlyRouter {
    require(block.timestamp > _maturity, "Can't repay before maturity");
    uint256 _X = X.intoUint256();
    X = ud((_X * (y + _cashIn)) / y); /// @audit Repay X vs cX
    y += _cashIn;
    if (loanData[_maturity].b > _cashIn) {
        loanData[_maturity].b -= _cashIn;
    } else {
        loanData[_maturity].b = 0;
    }
}

```

<https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingBondMM.sol#L298-L312>

```

function redeem(
    address _account,
    uint256 _maturity
) external onlyRouter returns (uint256 cashOut) {
    require(block.timestamp > _maturity, "Can't redeem before maturity");
    cashOut = bond.balanceOf(_account, _maturity);
    uint256 _X = X.intoUint256();
    X = ud((_X * (y - cashOut)) / y); /// @audit Borrower / Lender receives exact
    y -= cashOut;
    if (loanData[_maturity].l > cashOut) {
        loanData[_maturity].l -= cashOut;
    } else {
        loanData[_maturity].l = 0; /// @audit INVARIANT TEST to ensure we don't have >
        `loanData[_maturity].l`
    }
}

```

<https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingBondMM.sol#L75-L81>

```

modifier ValidMaturity(uint256 _maturity) {
    require(
        block.timestamp <= _maturity && matureAt[_maturity],
        "This maturity is not exists"
    );
    _;
}

```

Q-20 `LoanCalculator` should use feeds with 8 decimals

Impact

The code in `calcCollateralAmountToToken` looks as follows:

<https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/lib/Router/LoanCalculator.sol#L40-L45>

```
collateralAmountToToken =
    (_collateralAmount *
     collateralTokenPrice *
     10 ** (borrowedTokenDecimals + borrowedTokenPriceDecimals)) /
    (borrowedTokenPrice *
     10 ** (collateralTokenDecimals + collateralTokenPriceDecimals));
```

Let's look only at the collateral portion which has one additional multiplication:

```
(_collateralAmount *
 collateralTokenPrice *
 10 ** (borrowedTokenDecimals + borrowedTokenPriceDecimals))
```

Whenever we have tokens and feed with 18 decimals the values will roughly reach:

$1e18 * 1e18 * (10 ** (18 + 18))$

This leads us to $1e72$

Just 5 orders of magnitude away from the `type(uint256).max`

This can be very dangerous for any substantial size (10_000 units of the asset would cause an overflow)

Mitigation

Use feeds with 8 decimals, and fuzz test any integration with higher decimals as it will risk reaching the limit

TODO: Consider refactoring to a simplified formula that eliminates a few decimals

Q-21 Discrepancy between view functions and actual swap functions

QA / MED - Asymmetric Fee math in
`estimateBondAmountForExactQuoteToken` vs
`swapBondForExactQuoteToken`

The formula used are inconsistent

```
_amountOut = (_amountOut * TEN_THOUSANDS) / (TEN_THOUSANDS - fee);  
  
uint256 swapFee = (_amountOut * fee) / TEN_THOUSANDS;
```

See Python example:

```
>>> fee = 100  
>>> MAX_BPS = 10_000  
>>> 123 * fee / MAX_BPS + 123  
124.23  
>>> 123 * MAX_BPS / (MAX_BPS - fee)  
124.24242424242425  
>>> 12345 * fee / MAX_BPS + 12345  
12468.45  
>>> 12345 * MAX_BPS / (MAX_BPS - fee)  
12469.69696969697  
>>>
```

QA / MED - Asymmetric Fee math in
`estimateQuoteTokenAmountForExactBond` vs
`swapQuoteTokenForExactBond`

```
uint256 swapFee = (uint_delta_y * fee) / TEN_THOUSANDS;  
  
amountIn = (uint_delta_y * (TEN_THOUSANDS - fee)) / TEN_THOUSANDS;
```

Client

This function is no longer support: <https://github.com/spine-finance/resupplied-sm/pull/5>

Q-22 `addMaturity` allows adding maturities in the past

Fixed here: <https://github.com/spine-finance/resupplied-sm/pull/7>

Q-23 Gotcha: Collateral Withdrawal can be Blocked by Cross-Collateral Solvency Check

Impact

The `CollateralLTVProtected` modifier in `withdrawCollateral` function at <https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingRouter.sol#L327> prevents users from withdrawing unused collateral when they have debt backed by different collateral assets. This creates capital inefficiency where users cannot access their free collateral even when it's not securing any debt.

The issue affects:

- Capital efficiency for users with multiple collateral types
- User experience when managing collateral positions
- Liquidity management for borrowers

POC

1. User deposits collateral X (e.g., ETH)
2. User deposits collateral Y (e.g., USDC)
3. User mints debt against collateral X only
4. User attempts to withdraw collateral Y (which has no debt against it)
5. Transaction fails due to `CollateralLTVProtected` modifier checking solvency across all collateral for the specific collateral token being withdrawn

The modifier at <https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingRouter.sol#L45-L70> checks `getCollateralAmountToToken` and `getTotalBorrowedToken` for the specific collateral token, but the solvency check considers the overall position health rather than collateral-specific debt.

Mitigation

Document this gotcha and possible use only one collateral at a time per deployment

Q-24 Early return true on 0 borrows

Impact

The modifier `CollateralLTVProtected` is meant to ensure that the borrow / coll rate is safe

<https://github.com/spine-finance/resupplied-sm/blob/ae4a9cdadc0cf96ae6267a2797afe717b52e60d7/contracts/RestakingRouter.sol#L45-L49>

```
modifier CollateralLTVProtected(  
    address _account,  
    address _poolAddress,  
    address _collateralToken  
) {
```

This requires performing various calculations to compare the debt to the collateral value

When no borrows have happened, we can skip the check

Refactoring

<https://github.com/spine-finance/resupplied-sm/blob/ae4a9cdadc0cf96ae6267a2797afe717b52e60d7/contracts/RestakingRouter.sol#L59-L63>

```
uint256 borrowedToken = getTotalBorrowedToken(  
    _account,  
    _poolAddress,  
    _collateralToken  
);
```

You can skip the logic if borrowed is zero, and return true

Q-25 No Event on Liquidation

The function `liquidate` performs a key operation but doesn't emit any event

Q-26 Wrong event params on `LiquidityAdded`

Shares minted are scaled by decimals

[https://github.com/spine-finance/resupplied-](https://github.com/spine-finance/resupplied-sm/blob/ae4a9cdadc0cf96ae6267a2797afe717b52e60d7/contracts/RestakingBondMM.sol#L214-L220)

[sm/blob/ae4a9cdadc0cf96ae6267a2797afe717b52e60d7/contracts/RestakingBondMM.sol#L214-L220](https://github.com/spine-finance/resupplied-sm/blob/ae4a9cdadc0cf96ae6267a2797afe717b52e60d7/contracts/RestakingBondMM.sol#L214-L220)

```
vault = _vault;
poolFee = _fee;
if (underlyingDecimals <= 18) {
    _mint(_creator, _initLPShares * 10 ** (18 - underlyingDecimals));
} else {
    _mint(_creator, _initLPShares * 10 ** (underlyingDecimals - 18));
} /// @audit TODO: Are these scaling OK???
```

Event is using the amount: [https://github.com/spine-finance/resupplied-](https://github.com/spine-finance/resupplied-sm/blob/ae4a9cdadc0cf96ae6267a2797afe717b52e60d7/contracts/RestakingRouter.sol#L162-L167)

[sm/blob/ae4a9cdadc0cf96ae6267a2797afe717b52e60d7/contracts/RestakingRouter.sol#L162-L167](https://github.com/spine-finance/resupplied-sm/blob/ae4a9cdadc0cf96ae6267a2797afe717b52e60d7/contracts/RestakingRouter.sol#L162-L167)

```
emit LiquidityAdded(
    msg.sender,
    poolAddress,
    _tokenAmount,
    _tokenAmount
);
```

[https://github.com/spine-finance/resupplied-](https://github.com/spine-finance/resupplied-sm/blob/ae4a9cdadc0cf96ae6267a2797afe717b52e60d7/contracts/interfaces/IRestakingRouter.sol#L76-L81)

[sm/blob/ae4a9cdadc0cf96ae6267a2797afe717b52e60d7/contracts/interfaces/IRestakingRouter.sol#L76-L81](https://github.com/spine-finance/resupplied-sm/blob/ae4a9cdadc0cf96ae6267a2797afe717b52e60d7/contracts/interfaces/IRestakingRouter.sol#L76-L81)

```
event LiquidityAdded(
    address user,
    address pool,
    uint256 amountIn,
    uint256 shareAmount
);
```

A-01 Suggested Next Steps

Executive Summary

Spine Finance introduces a novel mechanism for pricing Fixed Rate loans.

This is ambitious and (as with any new technology) dangerous.

As it stands, I am fairly confident that the math is sound on paper.

In practice, especially due to integer operations not belonging to a mathematical group, rounding errors and edge cases could break the invariants and formulas for specific values.

In this short solo review I was able to identify numerous economic, business logic and implementation issues.

The codebase was sent without any significant unit test, and due to this some low hanging fruits were left.

The highly specialized codebase, paired with the lack of code maturity leads me to recommend that the code as it stands is not deployed to production.

During the review I have built a [Chimera](#) based [Invariant Testing Suite](#), I highly recommend that you review the suite, extend it and use it as part of your testing strategy. I also believe you need to write unit and integration tests for all parts of the code, and in the future should do that in parallel with development.

The codebase will benefit by:

- A rewrite of some of the math formulas to be fully stateless Libraries, that are Formally Verified
- The codebase needs to be fully tested
- The Invariant testing suite should be extended, and reviewed to make sure more properties of the system are tested

Due to:

- The novelty of the math
- The intricacies that derive from having multiple level of external integrations

I recommend that the codebase goes through a Security Contest before being deployed, as it's highly unlikely that one person is able to identify every issue in this codebase at all levels (economic, integration, logic, etc..)

If you need to deploy to hit investor milestones, change the code to only allow KYCd entities to interact with it

Ideal Roadmap

- 100% Test Coverage
- Invariant Tests
- Modelling of Adversarial attacks (e.g. flashloans to alter rate)
- Partial FV of libraries (ideally all math formulas are FVd)

- Audit with another firm
- Security Contest before launch
- Guarded launch with Bug Bounty

Note on Governance Risk

In order to massively reduce PK leak risks I recommend you implement the following Governance structure:

- The owner of the contracts should be a Timelock
- The Timelock should be owned by a Multisig

Due to this it's probably best to grant `pause` permissions to a `GUARDIAN` whereas unpausing and setting the guardian should be at the discretion of the admin

Consult this document for additional recommendations:

https://book.getrecon.xyz/opsec/op_sec.html

Notes on lack of thorough testing

I cannot over emphasize how important tests are in ensuring the code behaves as intended

Many projects have been exploited in DeFi, and the vast majority of them had extremely thorough testing suites

They still had bugs left in spite of the enormous effort put by the developers

Without a thorough set of tests, the code is very likely to have low hanging fruit issues and each new code change can bring new bugs.

We specialize in Invariant Testing which is a more advanced way of testing, I highly recommend you fork the repo I wrote over this engagement and maintain it.

That said, your code needs unit, and integration tests, without them, as well as more security reviews, you should not be confident to make these contracts available to the wider public.

Concerns for future reviewers

Here's a few idea I highly recommend a follow up reviewer explores

- Can I profitably flash lend to reduce the borrow rate and in aggregate pay less?
- Can I flash borrow to make `closeBorrowingPositionEarly` more expensive?
- Can I ever pay more than 100% on a early close?
- Is the flow of callbacks always safe, or are there conditions in which it's no longer safe? (Note custom pendle swapper, assume collateral and asset with hooks)
- Can I somehow sandwich deposits in the ERC4626 vault and cause losses to the BondMM?
- Is there any way to cause permanent DOS / Funds being stuck due to overflows?

Note on mitigation review

I have provided a complimentary mitigation review

I've done a review of each change

Given the amount of changes, as well as the lack of testing, you should seek an additional reviewer, as ultimately the volume of changes, paired with the lack of tests raises the likelihood that we missed some possible risk

A-02 Governance Mistake / PK Leak Risks

Pause and liquidate risk

Remove Coll configuration risk

Removing a collateral token will theoretically put a user health at 0

In practice this function will revert when calling a non-existing oracle

<https://github.com/spine-finance/resupplied->

[sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingRouter.sol#L351-L378](https://github.com/spine-finance/resupplied-sm/blob/4fc85b1944f8c4f901bbbfe218839b6a2a5f4e9e/contracts/RestakingRouter.sol#L351-L378)

```
function getCollateralAmountToToken(
    address _borrower,
    address _poolAddress,
    address _collateralToken
) public view returns (uint256 collateralAmountToToken) {
    PoolData memory poolData = pools[_poolAddress];
    CollateralTokenData memory collateralTokenData = collateralTokenInfo[
        _poolAddress
    ][_collateralToken];
    CollateralTokenData memory tokenPriceFeedData = collateralTokenInfo[
        _poolAddress
    ][poolData.tokenAddress];
    uint256 collateralAmount = userDeposit[
        LoanCalculator.getCollateralKey(
            _borrower,
            _poolAddress,
            _collateralToken
        )
    ];
    return
        LoanCalculator.calcCollateralAmountToToken(
            poolData.stakingTokenAddress,
            tokenPriceFeedData,
            _collateralToken,
            collateralTokenData,
            collateralAmount
        );
}
```

Change adapter and lose assets risk

A malicious adapter could steal all funds

If the PendleOnSpineStrategy is made upgradeable

Then all accounts that delegated could lose funds

E-01 LTV vs LLTV needs to be conscious of compounded Oracle Drift

Impact

Oracles are inherently inaccurate, the Deviation Threshold is the underlying price change at which a new round will start.

I use the term Oracle Drift for the difference between the underlying price and the price reported by the oracle.

Note that due to delays the Realized Oracle Drift can be higher than the Deviation Threshold, especially during liquidations and high volatility.

Having the result of two inaccurate oracles inherently compounds the inaccuracy.

Due to this, it's necessary that the LLTV is sufficiently low to ensure that liquidations are healthy in spite of the oracle inaccuracies

Math

Using ETH/USD (50 BPS Dev Threshold) and USDC / USD (25 BPS) we get


```

const MAX_BPS = 10_000;

export function applyDrift(
  price: number,
  driftBPS: number,
  isUp: boolean
): number {
  let newPrice = price;
  if (isUp) {
    newPrice = (price * (MAX_BPS + driftBPS - 1 / 1e18)) / MAX_BPS;
  } else {
    newPrice = (price * (MAX_BPS - driftBPS + 1 / 1e18)) / MAX_BPS;
  }

  return newPrice;
}

const getEBTCPrice = (coll: number, debt: number) => {
  return (coll * debt);
};

function doubleDrift(
  ethBtcPrice: number,
  ethBtcDriftBPS: number,
  stEthPrice: number,
  stEthDriftBPS: number
): { max: number; min: number; spot: number } {
  const spot = getEBTCPrice(ethBtcPrice, stEthPrice);

  let max = 0;
  let min = 0;

  // 4 possible prices
  const lowBtc = applyDrift(ethBtcPrice, ethBtcDriftBPS, false);
  const lowstEth = applyDrift(stEthPrice, stEthDriftBPS, false);
  const highBtc = applyDrift(ethBtcPrice, ethBtcDriftBPS, true);
  const highstEth = applyDrift(stEthPrice, stEthDriftBPS, true);

  const ll = getEBTCPrice(lowBtc, lowstEth);
  const lh = getEBTCPrice(lowBtc, highstEth);
  const hl = getEBTCPrice(highBtc, lowstEth);
  const hh = getEBTCPrice(highBtc, highstEth);

  min = Math.min(ll, lh, hl, hh);
  max = Math.max(ll, lh, hl, hh);

  return {
    spot,
    max,
    min,
  };
}

const drifted = doubleDrift(358189942348, 25, 99989574, 50);

console.log("doubleDrift", drifted);

console.log("max delta", (drifted.max / drifted.min) * 100);
console.log("max up", (drifted.max / drifted.spot) * 100);
console.log("max down", (drifted.min / drifted.spot) * 100);

```

```
doubleDrift {  
  spot: 35815259746461080000,  
  max: 36084321885306370000,  
  min: 35547092989109453000  
}  
max delta 101.51131597838818  
max up 100.75125  
max down 99.25124999999998
```

For estimating the risk of insolvency we can just look at max up / max down which is around 75 BPS

This means that the current formula can be inaccurate up to around 75 BPS and as such you should use intervals greater than that

Mitigation

Compute the doubly compounded threshold and ensure the LLTV is below that

Please note that this is a technical analysis, real LLTV may need to be even lower based on the volatility of the correlation of the two assets

When it comes to LLTV vs LTV make sure that there's a sufficient spread between the two (of at least 1%)

When it comes to the liquidation premium, make sure that the premium is above the compounded Oracle Drift as otherwise liquidators may not want to perform liquidations due to them not being profitable

Resources

Use this tool to help you calculate the oracle drift: <https://getrecon.xyz/tools/oracle-drift>

I-01 Additional Properties

X/Y only when X and Y are the same

<https://github.com/spine-finance/resupplied-sm/blob/ae4a9cdadc0cf96ae6267a2797afe717b52e60d7/contracts/RestakingBondMM.sol#L202-L206>

```
    } else if (_r0 > _r_star) {  
        X = (ud(_r0 - _r_star) / ud(_k0)).exp() * udCashIn; /// @audit Why?  
    } else {  
        X = udCashIn / ((ud(_r_star - _r0) / ud(_k0)).exp()); /// @audit Why?  
    }
```

These 2 conditions break the invariant due to truncation

Never 1

<https://github.com/spine-finance/resupplied-sm/blob/ae4a9cdadc0cf96ae6267a2797afe717b52e60d7/contracts/RestakingRouter.sol#L554-L555>

```
(uint256 sign, ) = getRate(_poolAddress);  
require(sign == 0, "Negative rate"); /// @audit Invariant Test
```

Overflow Scenarios / Solvency Math

<https://github.com/spine-finance/resupplied-sm/blob/ae4a9cdadc0cf96ae6267a2797afe717b52e60d7/contracts/RestakingBondMM.sol#L298-L325>

```

function redeem(
    address _account,
    uint256 _maturity
) external onlyRouter returns (uint256 cashOut) {
    require(block.timestamp > _maturity, "Can't redeem before maturity");
    cashOut = bond.balanceOf(_account, _maturity);
    uint256 _X = X.intoUint256();
    X = ud((_X * (y - cashOut)) / y);
    y -= cashOut;
    if (loanData[_maturity].l > cashOut) { /// @audit Invariant Test
        loanData[_maturity].l -= cashOut;
    } else {
        loanData[_maturity].l = 0;
    }
}

function repay(uint256 _cashIn, uint256 _maturity) external onlyRouter {
    require(block.timestamp > _maturity, "Can't repay before maturity");
    uint256 _X = X.intoUint256();
    X = ud((_X * (y + _cashIn)) / y);
    y += _cashIn;
    if (loanData[_maturity].b > _cashIn) { /// @audit Invariant Test
        loanData[_maturity].b -= _cashIn;
    } else {
        loanData[_maturity].b = 0;
    }
}

```

I-02 Invariant Testing

Latest Run

<https://getrecon.xyz/shares/6949c593-3ec0-4a4f-acc3-9be4cb474c50>

Code

<https://github.com/GalloDaSballo/spine-invariants/>

Restaking Bond Properties

- X/y is constant on liquidity addition and removal as well as yield
- `getBondPrice` matches a python implementation
- `getBondPrice` increases as we get closer to maturity (`vm.warp`)
- We MUST never have 0 Equity and non zero total supply as that disables the minting of shares
- Doomsday Arbitrage check: I should not be able to open \rightarrow close and have any gain (even with 0 fees)
- Doomsday Arbitrage Rate alteration: I should not be able to open \rightarrow close and have any alteration on the rate

Optimization

- X/y change max value

Fuzz

- `getRate` is the same for `UD60x18` and `SD59x18`

X/Y only when X and Y are the same

<https://github.com/spine-finance/resupplied-sm/blob/ae4a9cdadc0cf96ae6267a2797afe717b52e60d7/contracts/RestakingBondMM.sol#L202-L206>

```
    } else if (_r0 > _r_star) {
        X = (ud(_r0 - _r_star) / ud(_k0)).exp() * udCashIn; /// @audit Why?
    } else {
        X = udCashIn / ((ud(_r_star - _r0) / ud(_k0)).exp()); /// @audit Why?
    }
```

These 2 conditions break the invariant due to truncation

Never 1

<https://github.com/spine-finance/resupplied-sm/blob/ae4a9cdadc0cf96ae6267a2797afe717b52e60d7/contracts/RestakingRouter.sol#L554-L555>

```
(uint256 sign, ) = getRate(_poolAddress);
require(sign == 0, "Negative rate"); /// @audit Invariant Test
```

Overflow Scenarios / Solvency Math

<https://github.com/spine-finance/resupplied->

[sm/blob/ae4a9cdadc0cf96ae6267a2797afe717b52e60d7/contracts/RestakingBondMM.sol#L298-L325](https://github.com/spine-finance/resupplied-sm/blob/ae4a9cdadc0cf96ae6267a2797afe717b52e60d7/contracts/RestakingBondMM.sol#L298-L325)

```
function redeem(
    address _account,
    uint256 _maturity
) external onlyRouter returns (uint256 cashOut) {
    require(block.timestamp > _maturity, "Can't redeem before maturity");
    cashOut = bond.balanceOf(_account, _maturity);
    uint256 _X = X.intoUint256();
    X = ud((_X * (y - cashOut)) / y);
    y -= cashOut;
    if (loanData[_maturity].l > cashOut) { /// @audit Invariant Test
        loanData[_maturity].l -= cashOut;
    } else {
        loanData[_maturity].l = 0;
    }
}

function repay(uint256 _cashIn, uint256 _maturity) external onlyRouter {
    require(block.timestamp > _maturity, "Can't repay before maturity");
    uint256 _X = X.intoUint256();
    X = ud((_X * (y + _cashIn)) / y);
    y += _cashIn;
    if (loanData[_maturity].b > _cashIn) { /// @audit Invariant Test
        loanData[_maturity].b -= _cashIn;
    } else {
        loanData[_maturity].b = 0;
    }
}
```

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.