Liquity ChickenBond

Smart Contract Audit

coinspect

Liquity

ChickenBond

Smart Contract Audit

V220610

Prepared for Liquity • May 2022

1. Executive Summary

In **May 2022**, **Liquity** engaged Coinspect to perform a source code review of **ChickenBond**. The objective of the project was to evaluate the security of the smart contracts.

The project smart contracts follow security best practices and are extensively documented both with comments in the source code and documentation detailing known risks and potential threats. The test suite is comprehensive. At the moment of this audit, the code is still being developed, and the source code contains many `TODO` comments regarding open issues that are not fully defined yet.

This audit focused on the correctness of this specific implementation of the system described in the ChickenBonds whitepaper.

The following issues were identified during the initial assessment:

High Risk	Medium Risk	Low Risk
0	3	1
Fixed	Fixed	Fixed
0	0	0

The three medium risk issues are about the lack of guarantees that bond holders can always chicken out and get back the full principal amount (see **LCB-1**), and risks due to allowing anybody to trigger shifts of funds from Yearn's SP Vault to the Curve pool and back (see **LCB-5** and **LCB-9**).

It is recommended to address or mitigate **LCB-1**, and to remove public shift functions at least until the Curve pool behavior is fully understood and a worse case scenario loss for **LCB-5** and **LCB-9** is established.

2. Assessment and Scope

The audit started on **May 30, 2022** and was conducted on the **main** branch of the git repository at https://github.com/liquity/ChickenBond as of commit **`f9c22bb8d6bae0b82a2b80d735a5e871f133a591`** of **May 30, 2022**.

The audited files have the following sha256sum hash:

```
73bbde06271eda40045db5738c0d59b390f3ad69c043f65ff3982f5852b8d9dd  BLUSDToken.sol
f539b1e3f95f7b9b62a0d5709460d428ba6d539fa342afd2facc03d03f99612f  BondNFT.sol
148481746bcdb76d9fe3d14fd121ed3ea91ad7366d5585cf112d57ec48210a8f  ChickenBondManager.sol
d04be169247e8814b0dc0bed1cf5610a6a7bab37f41df072c4865efae5099759  LUSDSilo.sol
486deaf93c75e6da1ebf10b54887cd440d53e80d171d4cc45fc7419e489ab619  ExternalContracts/MockCurveLiquidityGaugeV4.sol
65b07744c4f1cb95c36a237afe1b7c8c30156882ac2d281373161e59ab5b262e  ExternalContracts/MockCurvePool.sol
5f6d5c6c964cdddb9ada98342b4abfa8bbba9e22aaaeb068d2ff52f5dc92ec91  ExternalContracts/MockLUSDToken.sol
5de512dc41a6287cd4c68a6fbc537deac310e043a723f6befbad3f481dfa124d  ExternalContracts/MockYearnRegistry.sol
70463f2ba7988e6a905ae26abf85f93bd866163f6334bfa26e40250e0fb76aa4  ExternalContracts/MockYearnVault.sol
a2979fb37ed51708293e07ec00d274abe17f5fd94b7eec62e9beadc4f9ec2331  Interfaces/IBLUSDToken.sol
8aea8df2a8ac86b2ce835ad017d228b9710ab68f19d0cc158b77c186d5785051  Interfaces/IBondNFT.sol
f0f8fc166960bf59b9c936a1e1dc1dd9c44d32e74b0ad2d9c7d0d8d2a1eb939c  Interfaces/IChickenBondManager.sol
c8813e186b5d0a7e9f09b66af976ed6992f7887481cd6e40f3b3d1978bdd1d50  Interfaces/ICurveLiquidityGaugeV4.sol
187e9a863a11d78bd2c8277edd881d878f1f1c8ad00dcc70e2e4575ecabe9dcf  Interfaces/ICurvePool.sol
8410326508cc0296bafd4482ef56a54ddde701d0e5903f205cb799023122e0bc  Interfaces/ILUSDToken.sol
d70b12499cce02cb162203879531274e1d1509c32cc2c7aed7f83569891f46dc  Interfaces/IYearnRegistry.sol
b85a5aa5b27a94e13476f2a92c28a0dc55267f9be816c3735949b8b431953608  Interfaces/IYearnVault.sol
6c0218058b30032d1aec1e32491f71ecdb5d431eeed6f5a636b12e72ff77c9bd  Interfaces/StrategyAPI.sol
1214cf1a20bb8bc10b9a2c56e3b5317f6eef3abde8ad89da51d9570ab41df453  Proxy/ChickenBondOperationsScript.sol
21d2c25d5296d74543316bdc6d58112c4e63ed29c09c8271430259bf75ef9c62  utils/BaseMath.sol
d5c440af881bcb0e03599cca708cd0d27b61f87eeeed8106904e555026297211  utils/ChickenMath.sol
827d5d6c3e54dfcb69de11e77f48e99213ae891ea2f04f4291f5847139dc9fe4  utils/console.sol
```

The ChickenBond contracts interact and trust user funds to Yearn Finance and Curve contracts. These third-party contracts were out of the assessment's scope and considered as trusted.

As explained in the whitepaper, *ChickenBonds is an autonomous self-bootstrapping treasury system that acquires yield-bearing tokens through a novel bonding mechanism: users may bond a token TKN and start accruing a "boosted" derivative token bTKN in return*. In this implementation users may bond LUSD and start accruing a "boosted" derivative token bLUSD. Bond holders should be able at any time to retrieve their LUSD principal foregoing the accrued amount ("chicken out"), or trade it in against the accrued bLUSD ("chicken in").

All contracts are specified to be compiled with Solidity version at least 0.8.10, and during the assessment they were compiled with Solidity 0.8.14.

The contracts are not upgradable. Some of the contracts are `Ownable`, but the owner renounces ownership after the contracts are initialized during deployment and they become completely autonomous. The contracts are well-written and very clear. They make use of the *immutable* modifier for all state variables that don't change. And they contain useful and clear comments. However, the contracts contain many `TODO` comments, and some parts of the implementation might change after the audit.

The repository includes a set of tests run on a dev network, as well as a second set of tests that are run on a mainnet fork. The dev test suite includes 81 tests and all pass. The mainnet test suite comprises 46 tests, with all passing except one test that failed (`testFirstChickenInAfterRedemptionDepletionAndCurveHarvestTransfersToRewardsContract`). Also, developers disabled a test from the mainnet suite because it sometimes fails (see **LCB-6**) and another test is incorrect (see **LCB-7**).

The main contract of the project is `ChickenBondManager`. There's also `BLUSDToken` that implements the boosted token (bLUSD), `BondNFT` that implements the bonds as NFTs, `LUSDSilo` that is just used as an address to hold tokens when in migration mode, and `ChickenBondOperationsScript` that is a contract that users can use to interact with the `ChickenBondManager` in a more convenient way.

The contract `BLUSDToken` contains an import of `console.sol`, probably a leftover from debugging (see **LCB-2**). It is recommended to remove this unused import.

The contract `ChickenBondManager` unnecessarily inherits from `Ownable`. The contract doesn't contain any `onlyOwner` function, and it immediately renounces ownership in the constructor (see **LCB-3**).

The contract `ChickenBondManager` makes one-time infinite LUSD approvals to Yearn and Curve in the constructor. As noted in a comment, this is a trade-off of security for lower gas costs.

The contract `ChickenBondOperationsScript` has a minor mistake in the constructor that calls `Address.isContract()` with the address passed to the constructor as the address of the `ChickenBondManager` contract but ignores the return value. It is recommended to use `require` and revert if

`Address.isContract()` returns false, as it was clearly intended by the programmer (see **LCB-4**).

The ChickenBondManager contract provides functions to shift LUSD from Yearn's SP Vault to the Curve pool (function `shiftLUSDFromSPToCurve()`), and back from the Curve pool to the SP Vault (function `shiftLUSDFromCurveToSP()`). These functions can be called by any address.

The function `shiftLUSDFromSPToCurve()` does not handle the scenario where the total LUSD in Yearn's SP Vault is less than the protocol's outstanding pending balance. In those scenarios the function reverts because of an unhandled overflow while subtracting these two values (see **LCB-8**).

The shift functions have checks on the change of the spot price, and revert if the spot price moves unfavorably. However, there are currently no checks to prevent arbitrary losses. In some circumstances these functions could be abused to drain protocol funds by triggering unfavorable swaps in a manipulated unbalanced Curve liquidity pool (see **LCB-5**). It is recommended to remove the public shifting functionality until the Curve pool behavior is fully understood and a worse case scenario loss is established.

During the assessment, Coinspect found that when bond holders decide to convert their bonds back to LUSD ("chicken out"), they are not guaranteed to recover the full amount of LUSD that was originally deposited, contradicting the ChickenBonds whitepaper. In a scenario where many users chicken out in a short period of time, this issue unfairly affects users that chicken out last (see **LCB-1**). This issue is aggravated by the possibility of front-running calls to `chickenOut()` with a shift of funds of the permanent bucket from the Yearn's SP Vault to the Curve pool (see **LCB-9**). It is advisable to consider options to address or mitigate this issue, for example by letting users choose a minimum amount to get back when performing a chicken out, or by implementing partial chicken outs that don't burn the bonds but instead reduce the amount of principal LUSD, or by distributing the loss among all bond holders.

3. Summary of Findings

Id	Title	Total Risk	Fixed
LCB-1	Bond is burned when full LUSD amount is not available	Medium	✘
LCB-2	Unused console import	Info	✘
LCB-3	Ownable not needed	Info	✘
LCB-4	Missing require call in constructor parameter verification	Info	✘
LCB-5	External pools manipulation could lead to loss of funds	Medium	✘
LCB-6	Critical test disabled	Low	✘
LCB-7	Incorrect test or comment	Info	✘
LCB-8	shiftLUSDFromSPToCurve unexpected revert scenario	Info	✘
LCB-9	Chicken outs can be front-ran to harm users and profit	Medium	✘

4. Detailed Findings

LCB-1	Bond is burned when full LUSD amount is not available	
Total Risk Medium	Impact High	Location `ChickenBondManager.sol`
Fixed ✘	Likelihood Low	

Description

When bond holders decide to convert their bonds back to LUSD ("chicken out"), they are not guaranteed to recover the full amount of LUSD they had deposited. This contradicts the ChickenBonds whitepaper.

The `chickenOut()` function pays back the minimum between the LUSD entitled to the bond and the currently available LUSD in the vault. However, the full bond is burned, no matter how much the user has recovered from the original deposit.

```
            lusdToWithdraw = Math.min(bond.lusdAmount, lusdToken.balanceOf(lusdSiloAddress));
            [...]
            lusdToken.transferFrom(lusdSiloAddress, msg.sender, lusdToWithdraw);
        }
        [...]
        bondNFT.burn(_bondID);
```

The potential conditions that could lead to the scenario where the pending bucket is not fully backed are detailed in a `TODO` comment in the contract's source code. As the comment in `chickenOut()` stands, there are edge cases that could result in `totalPendingLUSD` not being fully backed:

1. Heavy liquidations, and before yield has been converted
2. Heavy loss-making liquidations, i.e. at <100% CR
3. SP or Yearn Vault hack that drains LUSD

In any of these edge cases, bond holders calling `chickenOut()` could end up with their bonds burned in exchange for less than their original principal. **This problem unfairly affects the bond holders that chicken out last.**

The protocol's invariants specification states the following regarding the `chickenOut()` function, which does not exactly match the implementation:

Refunds full bonded amount to bonder	Occasionally may withdraw slightly less due to rounding error. Occasionally may revert temporarily, if the available LUSD in the Yearn SP vault is insufficient, due to Liquity liquidations that have had not had their ETH gain harvested and converted back to LUSD.

Moreover, the ChickenBonds whitepaper states there is no risk for bond holders:

> *Risk-free bonding for users. Bond holders may "chicken out" and withdraw their deposited principal at any time, in lieu of converting it to derivative tokens. Therefore bonding is fully reversible and bond holders are not locked into any commitment to the protocol.*

Recommendation

Bond holders should be able to provide a minimum amount of LUSD they expect to recover when burning their bonds. This would allow users to protect themselves in changing conditions.

Another possibility is to implement partial burns, so the user is able to claim the remaining LUSD when the funds become available.

The issue could also be mitigated by making the chicken out proportional to the actual amount of LUSD available (only in case that `totalPendingLUSD` < LUSD in vault or silo). In this way bond holders would be all equally affected, instead of making the bond holders that chicken out last absorb the loss.

Another possible mitigation, to some extent, is to ensure that all LUSD that can be shifted from the curve to the SP Vault is shifted first, so that the balance of the vault is maximized at the time of computing the amount to return during a chicken out.

In any case, it is recommended to clearly document this possibility and make users aware of the risks.

Status

Open.

LCB-2 Unused console import

Total Risk **Info**	Impact -	Location `BLUSDToken.sol`
Fixed ✘	Likelihood -	

Description

The following import is not used by the contract:

```
import "./utils/console.sol";
```

Recommendation

Remove the unused import.

Status

Open.

LCB-3	Ownable not needed	
Total Risk Info	Impact -	Location `ChickenBondManager.sol`
Fixed ✘	Likelihood -	

Description

The contract constructor calls `renounceOwnership()` so deriving from `Ownable` seems unnecessary.

Probably this contract was originally initialized in another function protected with the `onlyOwner` modifier, and later it was changed to a constructor.

Recommendation

Do not inherit from `Ownable` unless necessary.

Status

Open.

LCB-4 Missing require call in constructor parameter verification

Total Risk	Impact	Location
Info	-	`ChickenBondsOperationsScript.sol`
Fixed ✘	Likelihood -	

Description

The following code in the contract constructor has no effects and can be removed or should be used in a require call:

```
constructor(IChickenBondManager _chickenBondManager) {
    Address.isContract(address(_chickenBondManager));
```

Recommendation

Add the missing require call or remove the unused line of code.

Status

Open.

LCB-5 External pools manipulation could lead to loss of funds

Total Risk	Impact	Location
Medium	High	`ChickenBondsManager.sol`
Fixed	**Likelihood**	
✘	Low	

Description

The public `shiftLUSDFromSPToCurve()` and `shiftLUSDFromCurveToSP()` functions can be called by anybody by design. This could be abused to drain protocol funds by triggering unfavorable swaps in a manipulated unbalanced Curve liquidity pool. There are currently no checks in the code to prevent arbitrary losses from occurring.

It is generally dangerous to allow any account to trigger interactions with the external pools, which could themselves be subject to manipulation. Note the unbalanced pool scenario could occur naturally as well as a consequence of regular pool operations without needing the attacker's intervention. These scenarios could be only temporal (as arbitrageurs would be incentivized to get involved), but shifts can be invoked whenever the conditions in the pool are most beneficial for the callers, and as many times as desired.

In a worst case scenario, the only cost for attackers is the gas used by the shift transaction. This cost is low compared to losing up to 1% of the shifted amount. According to Liquity's estimations, relative loss is between [0.1%,1%] and up to 1% when shifting from the SP Vault to the Curve pool. This relative loss is calculated between the deposit and withdrawal of the same amount of funds in Curve pool, without further transactions in the middle. This value was obtained by simulating transactions with different parameters in a mainnet fork.

The ability to manipulate the Curve pool involved will depend on the available liquidity at a given time.

The price validation enforced in the last require check in both shift functions allows shifts that move the price in the correct direction, close to 1, without crossing this value. This limits the amount that can be shifted. **However, the relative loss is not measured nor limited by these functions.**

A minimum expected value is not provided to the Curve pool `add_liquidity()` and `remove_liquidity_one_coin()` functions, as clearly noted in a `TODO` comment in the source code. **This means any amount of Curve pool shares would be accepted in return for the amount of LUSD added or removed.**

Lost funds would benefit Curve liquidity providers, and this could serve as incentive to abuse the shift mechanism.

This is the relevant code in the `shiftLUSDFromSPToCurve()` function:

```
        /* TODO: Determine if we should pass a minimum amount of LP tokens to receive here. Seems
infeasible to determinine the mininum on-chain from
        * Curve spot price / quantities, which are manipulable. */
        curvePool.add_liquidity([lusdBalanceDelta, 0], 0);
        uint25
```

And this is the relevant code in the `shiftLUSDFromCurveToSP()` function:

```
        // Withdraw LUSD from Curve
        uint256 lusdBalanceBefore = lusdToken.balanceOf(address(this));
        /* TODO: Determine if we should pass a minimum amount of LUSD to receive here. Seems
infeasible to determinine the mininum on-chain from
        * Curve spot price / quantities, which are manipulable. */
        curvePool.remove_liquidity_one_coin(lusd3CRVBalanceDelta, INDEX_OF_LUSD_TOKEN_IN_CURVE_POOL,
0);
```

A full cost/benefit analysis of this issue has not been performed and requires fully analyzing Curve's pool behavior in order to quantify the maximum loss that the protocol would suffer in each shift scenario. At the moment of writing this report it is not clear if the benefits of moving the peg closer to 1 would cancel the losses caused by the pool operations.

This analysis focused on Curve pool operations. According to Yearn documentation, the V2 Vaults fee scheme would not make this attack scenario possible. Coinspect recommends performing simulations and/or reading the code to make sure the attack is not possible.

Recommendation

Coinspect recommends removing the public shifting functionality until a worse case scenario loss is established. Allowing funds swaps to be triggered by non-trusted agents, involving external contracts subject to manipulation, without any safeguard is not recommended.

Coinspect suggests investigating if it is possible to simulate a liquidity removal call right after the deposit, for example by using Curve's `calc_withdraw_one_coin()` function, and limit the calculated loss with a maximum allowed loss. This would result in a guaranteed loss cap that will be enforced no matter the scenario and all possible context changes in the future. This would also enable calculation of the profitability of the shift functions.

The following additional potential improvements are not perfect and might have undesired consequences for the protocol's design goals (e.g., introducing trusted off-chain components):

1. Consider limiting the shift functions access to a trusted account and calculate a minimum expected value for the swaps off-chain.
2. Consider limiting the shift functions access to users with a minimum amount of protocol shares.

Status

This issue is related to an issue noticed by the team and shared with Coinspect when the audit started: the team observed a slight loss of funds when depositing to and withdrawing from Curve. There are tests related to this issue that expose the potential loss that the protocol could incur when shifts take place.

Further research is recommended in order to fully understand the concrete impact of this issue.

LCB-6 Critical test disabled

Total Risk	Impact	Location
Low	Low	`ChickenBondManagerMainnetOnlyTest.sol`
Fixed	**Likelihood**	
✘	Low	

Description

There is a critical test disabled, signaling either lack of understanding of Curve interactions or a non effective check present in the code. This could result in loss of funds.

The `testShiftLUSDFromCurveToSPRevertsWhenShiftWouldRaiseCurvePriceAbove1` test is currently disabled. According to the developer's comment, the test doesn't always pass, depending on the forked mainnet block used. This test is intended to verify that the last `require` call in the `shiftLUSDFromCurveToSP()` function correctly prevents the price from crossing 1:

```
    // TODO: refactor this test to be more robust to specific Curve mainnet state. Currently
sometimes fails.
    // function testShiftLUSDFromCurveToSPRevertsWhenShiftWouldRaiseCurvePriceAbove1() public {
```

However, it appears that sometimes this check fails and that the issue is not yet fully understood.

Note that this test is related to the shift functionality which is considered risky as described in finding **LCB-5**.

Recommendation

Enable the test and make sure it passess in many different scenarios.

Status

Open.

LCB-7 Incorrect test or comment

Total Risk	Impact	Location
Info	-	`ChickenBondManagerMainnetOnlyTest.sol`
Fixed ✘	Likelihood -	

Description

Coinspect observed a mismatch between the code and its related comment in the `testCurveImmediateLUSDDepositAndWithdrawalLossIsBounded` test.

An unnoticed incorrectly implemented test (or its misleading comments) could lead to undiscovered flaws.

In the `testCurveImmediateLUSDDepositAndWithdrawalLossIsBounded` test, 1e15 should be 1e16 instead:

```
        // Check that simple Curve LUSD deposit->withdraw loses between [0.01%, 1%] of initial
deposit.
        assertLt(curveRelativeDepositLoss, 1e15);
        assertGt(curveRelativeDepositLoss, 1e14);
```

Note that this test is related to the shift functionality which is considered risky as described in finding **LCB-5**.

Recommendation

Update the comment or change the code to match.

Status

Open.

LCB-8 shiftLUSDFromSPToCurve unexpected revert scenario

Total Risk	Impact	Location
Info	-	`ChickenBondManager.sol`
Fixed ✘	Likelihood -	

Description

The `shiftLUSDFromSPToCurve()` function does not handle the scenario where the total LUSD in Yearn's SP Vault is less than the protocol's outstanding pending balance. In those scenarios the function reverts because of an unhandled overflow while subtracting these two values:

```
    function shiftLUSDFromSPToCurve(uint256 _lusdToShift) external {
        _requireNonZeroAmount(_lusdToShift);
        _requireMigrationNotActive();

        uint256 initialCurveSpotPrice = _getCurveLUSDSpotPrice();
        require(initialCurveSpotPrice > 1e18, "CBM: Curve spot must be > 1.0 before SP->Curve
shift");

        uint256 lusdInSP = calcTotalYearnSPVaultShareValue();

        /* Calculate and record the portion of LUSD withdrawn from the permanent Yearn LUSD bucket,
        assuming that burning yTokens decreases both the permanent and acquired Yearn LUSD buckets by
the same factor. */
        uint256 lusdOwnedLUSDVault = lusdInSP - totalPendingLUSD;
```

This scenario is handled in the `chickenOut()` function (though the consequences in that case might not be ideal, as described in issue **LCB-1**):

```
            uint256 lusdInLUSDVault = calcTotalYearnSPVaultShareValue();
            lusdToWithdraw = Math.min(bond.lusdAmount, lusdInLUSDVault);
```

Because a reverted shift operation does not represent a direct risk to user funds and the situation is highly unlikely, this issue is considered informative and addressing it would be only a minor improvement.

Recommendation

The shift operation can continue when there is yet no owned funds in the vault. Alternatively, consider reverting with an appropriate error string.

Status

Open.

LCB-9 Chicken outs can be front-ran to harm users and profit

Total Risk	Impact	Location
Medium	Medium	`ChickenBondsManager.sol`
Fixed ✘	Likelihood Medium	

Description

In certain circumstances, calls to `chickenOut()` can be front-ran with a call to function `shiftLUSDFromSPToCurve()` to perform a shift of funds of the permanent bucket from the Yearn SP Vault to the Curve pool. As a result, users trying to recover their funds might end up with less LUSD than expected (almost zero LUSD is possible) and their bond burned.

The funds from the burned bond will remain in the platform, and will be redistributed as yield to the other users. As a result, users are incentivized to front-ran chicken outs.

Funds in the Yearn SP Vault are used to pay back the LUSD deposited to create the bond. Those funds were originally deposited in the vault when the bond was created. When the user decides to chicken out, only the funds in this vault are used. However, the protocol allows transferring all funds from this vault to the Curve pool. If that happens, the chicken out transaction could end up with two results:

1. The users receive less than originally bonded and their full bond is burned, without any possibility to protect themselves.
2. If nothing is left in the vault, the transaction reverts when division by zero is attempted in the `_calcCorrespondingYTokens()` function.

```
   // Returns the yTokens needed to make a partial withdrawal of the CBM's total vault deposit
   function _calcCorrespondingYTokens(IYearnVault _yearnVault, uint256 _wantedTokenAmount, uint256
_CBMTotalVaultDeposit) internal view returns (uint256) {
        uint256 yTokensHeldByCBM = _yearnVault.balanceOf(address(this));
        uint256 yTokensToBurn = yTokensHeldByCBM * _wantedTokenAmount / _CBMTotalVaultDeposit;
        return yTokensToBurn;
   }
```

Because of the reason explained above, this finding can be seen as an attack vector for issue **LCB-1**.

Mitigating factors: the ability to shift most of the pending bucket funds from the SP Vault to the Curve Pool is limited by the current spot price and its deviation from the 1 peg, as enforced in the last require call in the shift function. However, this issue could be more easily exploitable during the protocol bootstrap when the amount of funds in the SP Vault is smaller. In that case, users attempting to chicken out could be harmed, and the price of bonds in external markets could be affected.

The following log excerpt was obtained from a proof of concept test developed during the audit:

```
100000000000000000000,  LUSD deposit by userA
100000000000000000000,  LUSD deposit by userB
199999999999999999858,  LUSD in SP before harvest
200000125349336796553,  LUSD in SP after harvest
200000125349336796552,  LUSD will be shifted from SP to Curve
shifted all_LUSD_in_SP-1 to Curve
userA chickens out
1,  LUSD only received by userA
userB attempts to chicken out and reverts

[FAIL. Reason: Division or modulo by 0]
         testUserGetsNothingAfterShiftToCurve() (gas: 2774098)
```

The log shows two scenarios:

1. userA deposits 100e18 LUSD but only receives 1 LUSD when chickens out
2. userB chicken out reverts

Proof of concept

```
function testUserGetsNothingAfterShiftToCurve() public {
    // A, B create bond
    uint256 bondAmount = 100e18;

    createBondForUser(A, bondAmount);
    // Get A's bondID
    uint256 A_bondID = bondNFT.totalMinted();
    console.log(bondAmount, " LUSD deposit by userA");

    createBondForUser(B, bondAmount);
    // Get B's bondID
    uint256 B_bondID = bondNFT.totalMinted();
    console.log(bondAmount, " LUSD deposit by userB");


    uint256 lusdInSPBefore = chickenBondManager.calcTotalYearnSPVaultShareValue();
    console.log(lusdInSPBefore," LUSD in SP before harvest");
    _spHarvestAndFastForward();
    uint256 lusdInSPAfter = chickenBondManager.calcTotalYearnSPVaultShareValue();
    console.log(lusdInSPAfter," LUSD in SP after harvest");
```

```
        // Shift all LUSD in SP
        makeCurveSpotPriceAbove1(200_000_000e18);
        uint256 lusdToShift = lusdInSPAfter-1;
        console.log(lusdToShift," LUSD will be shifted from SP to Curve");
        chickenBondManager.shiftLUSDFromSPToCurve(lusdToShift);
        console.log("shifted all_LUSD_in_SP-1 to Curve");

        // A chickens out
        vm.startPrank(A);
        console.log("userA chickens out");
        uint256 userABalanceBefore = lusdToken.balanceOf(address(A));
        chickenBondManager.chickenOut(A_bondID);
        uint256 userABalanceAfter = lusdToken.balanceOf(address(A));
        vm.stopPrank();
        console.log(userABalanceAfter-userABalanceBefore, " LUSD only received by userA");

        uint256 totalPendingLUSD = chickenBondManager.totalPendingLUSD();
        console.log(totalPendingLUSD," totalPendingLUSD");

        // B chickens out
        vm.startPrank(B);
        console.log("userB attempts to chicken out and reverts");
        chickenBondManager.chickenOut(B_bondID);
        vm.stopPrank();
    }
```

Recommendation

Coinspect suggests evaluating if it is convenient to prevent the `shiftLUSDFromSPToCurve()` function from moving the funds in the pending bucket to the Curve pool, or if those funds should be always available in the Yearn's SP Vault.

Consider the case when the `_CBMTotalVaultDeposit` parameter is 0.

Also, see recommendations in finding **LCB-1**.

Status

Open.

5. Disclaimer

The information presented in this document is provided "as is" and without warranty. The present security audit does not cover any off-chain systems or frontends that communicate with the contracts, nor the general operational security of the organization that developed the code.