LUSD Chicken Bonds

Smart Contract Security Assessment

13.07.2022

ABSTRACT

Dedaub was commissioned to audit the LUSD Chicken Bonds protocol. The LUSD Chicken Bonds protocol is a specific implementation of the General Chicken Bonds model which aims to acquire protocol-owned liquidity for the LUSD token at no cost while boosting the yield opportunities for end users.

SETTING AND CAVEATS

The code and accompanying artifacts (e.g., test suite, documentation) are of excellent quality, developed with high professional standards. The overall financial logic is described in https://www.chickenbonds.org, with a detailed accompanying whitepaper.

The scope of the audit includes the ChickenBondManager contract as well as its direct dependencies and related tokens (BLUSD and Bond NFTs). Specifically, files:

BLUSDToken.sol
BondNFT.sol
ChickenBondManager.sol
utils/BaseMath.sol
utils/ChickenMath.sol

This audit report covers commit hash `6f32111942acc257e55337cac73f2843a22f15e8`. Two auditors and a mathematician worked over the codebase over 1.5 weeks.

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than regular use of the protocol. Functional correctness (i.e., issues in "regular use") was a secondary consideration, however intensive efforts were made to check the correct application of the mathematical formulae in the reviewed code. Functional correctness relative to low-level calculations (including units, scaling, quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

The scope of the audit also included crypto-economic considerations. A number of checks have been carried out, however the crypto-economic effectiveness of this specific design is novel. Therefore, the financial viability of this protocol in real market conditions cannot be fully established. In terms of protocol completion, the protocol appears to be ready for staging.

The resolution of report items is determined by local inspection of changes, not a full re-audit. Therefore, locally sound fixes may have issues in a broader context. The development team is advised to be especially vigilant with testing the consequences of fixes performed after the initial audit.

PROTOCOL-LEVEL CONSIDERATIONS

There are elements of the design that can have significant financial implications, yet cannot be attributed to a specific part of the code, but to the overall architecture. The main one is the functionality to remove LUSD from the Liquity Stability pool in order to support the peg on the price of LUSD (indirectly, through a Yearn vault) in a Curve meta pool of LUSD+3CRV.

Since the goal is for significant liquidity (in the hundreds of millions, as mentioned in accompanying documentation) through the Chicken Bonds facility, the financial considerations become non-negligible. Namely, LUSD will then become a prop for USDT, USDC, and DAI. For instance, in case of a USDT de-pegging (collapse to much below $1), the Curve pool token will be flooded with USDT. USDT holders will be able to exit to LUSD, through the meta pool (i.e., getting 3CRV tokens and swapping them for LUSD). A rise to the price of LUSD will be countered by the Chicken Bonds “shift” mechanism which allows shifting liquidity from the Stability Pool to add LUSD to Curve, effectively propping up USDT for as long as the ChickenBonds protocol has SP liquidity.

We bring this consideration to the protocol designers' attention, for further discussion. The main threat is to Chicken Bonds investors (specifically, bLUSD holders, not bonders, since bonds are backed only by Stability Pool holdings, not Curve) but there may be wider consequences, such as pressure to LUSD and to Liquity, through large amounts of LUSD seeking to exit quickly.

A general protocol-level threat is that much of the design has been done with the implicit understanding that yield-bearing investments (to the SP or to Curve) never lose value (either on paper or in real terms). For instance, consider the whitepaper invariant "redemption price never decreases". This is not an invariant, i.e., it is not literally true at all times. If there are investment losses, there is not enough LUSD to cover redemptions, so the invariant is not true when combined with the definition of "redemption price":

> "we call the amount of TKN for which each bTKN can be redeemed for the *redemption price*".

We recommend to the development team a critical review of the stated properties with losses considered a possibility, however unlikely.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues that affect the functionality of the contracts. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third party attackers or faulty functionality may block the system or

	cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: -User or system funds can be lost when third party systems misbehave. -DoS, under specific conditions. -Part of the functionality becomes unusable due to programming error.
LOW	Examples: -Breaking important system invariants, but without apparent consequences. -Buggy functionality for trusted users where a workaround exists. -Security issues which may manifest when the system evolves.

Issue resolution includes “dismissed” or “acknowledged” but no action taken, by the client, or “resolved”, per the auditors.

CRITICAL SEVERITY:

[No critical severity issues]

HIGH SEVERITY:

ID	Description	STATUS
H1	Stability Pool deposits can be manipulated for possible gain	**RESOLVED** **(mitigated by commit cf15a5ac: time delay for shifting, limited shift window)**
	The Chicken Bonds protocol gives arbitrary callers the ability to “shift” liquidity out of the stability pool and into Curve, when the price of LUSD (on Curve) is too high. This can be abused for a financial attack if the protocol (i.e., the B.AMMSP) becomes a major shareholder of stability pool liquidity, as expected. Consider a scenario where the attacker notices a large liquidation coming in. Stability pool shareholders stand to gain up to 10%. The attacker wants to eliminate the B.AMMSP share from the stability pool and receive a larger part of the gains. The attacker can tilt the Curve pool (e.g., via flashloan) to get the LUSD price to be outside the acceptable threshold (too high). With a subsequent call of `shiftLUSDFromSPToCurve`, liquidity gets removed from the stability pool.	
H2	An attack tilting the Curve pool before a `redeem` allows the attacker to draw funds from the permanent bucket	**RESOLVED** **(commit 900d481a makes all Curve accounting be in virtual/relative terms)**
	There is a Curve-pool-tilt attack upon a `redeem` operation. The core of the issue is the maintaining between transactions of a storage variable, `permanentLUSDInCurve`:	

```
uint256 private permanentLUSDInCurve; // Yearn Curve LUSD-3CRV vault
…
function shiftLUSDFromSPToCurve(uint256 _maxLUSDToShift) external {
  ...
    uint256 permanentLUSDCurveIncrease = (lusdInCurve -
      lusdInCurveBefore) * ratioPermanentToOwned / 1e18;
    permanentLUSDInCurve += permanentLUSDCurveIncrease;
  ...
}

function shiftLUSDFromCurveToSP(uint256 _maxLUSDToShift) external {
  ...
  uint256 permanentLUSDWithdrawn =
    lusdBalanceDelta * ratioPermanentToOwned / 1e18;
  permanentLUSDInCurve -= permanentLUSDWithdrawn;
  ...
}
```

The problem is that this quantity does not really reflect current amounts of LUSD in Curve, which are subject to fluctuations due to normal swaps or malicious pool manipulation. The `permanentLUSDInCurve` is then used in the computation of acquired LUSD in Curve:

```
function getAcquiredLUSDInCurve() public view returns (uint256) {
  uint256 acquiredLUSDInCurve;

  // Get the LUSD value of the LUSD-3CRV tokens
  uint256 totalLUSDInCurve = getTotalLUSDInCurve();
  if (totalLUSDInCurve > permanentLUSDInCurve) {
    acquiredLUSDInCurve = totalLUSDInCurve - permanentLUSDInCurve;
  }

  return acquiredLUSDInCurve;
}
```

A `redeem` computes the amount to return to the caller using the above function, as a proportion of the acquired LUSD in Curve:

```
function redeem(uint256 _bLUSDToRedeem, uint256 _minLUSDFromBAMMSPVault)
external returns (uint256, uint256) {
  …
  uint256 acquiredLUSDInCurveToRedeem = getAcquiredLUSDInCurve() *
                                        fractionOfBLUSDToRedeem / 1e18;
  uint256 lusdToWithdrawFromCurve = acquiredLUSDInCurveToRedeem *
                                (1e18 - redemptionFeePercentage) / 1e18;
  uint256 acquiredLUSDInCurveFee = acquiredLUSDInCurveToRedeem -
                                   lusdToWithdrawFromCurve;
  yTokensFromCurveVault =
    _calcCorrespondingYTokensInCurveVault(lusdToWithdrawFromCurve);
  if (yTokensFromCurveVault > 0) {
    yearnCurveVault.transfer(msg.sender, yTokensFromCurveVault);
  }
```

As a result, the attack consists of lowering the price of LUSD in Curve, by swapping a lot of LUSD, so that the Curve pool has a much larger amount of LUSD. The `permanentLUSDInCurve` remains as stored from the previous transaction and gets subtracted, so that the acquired LUSD in Curve appears to be much higher. The attacker calls `redeem` and receives a proportion of that amount (minus fees), effectively stealing from the permanent LUSD.

The general recommendation is to not store between transactions any amount reflecting Curve balances (either total or partial). If a partial balance is to be kept, it should be kept in relative terms (i.e., a proportion) not absolute token amounts.

MEDIUM SEVERITY:

[No medium severity issues]

LOW SEVERITY:

ID	Description	STATUS

L1	Unsafe Curve operation (on possibly tilted pool) upon `_firstChickenIn`	**RESOLVED (commit 30c45567: depletion of bLUSD prevented)**

[This issue is rated “low severity” because a firstChickenIn after initial deployment is unlikely. The core threat remains.]

The code of `firstChickenIn` effectively performs an unprotected swap of the full balance of Curve yield at the time (a `remove_liquidity_one_coin` functionally includes a swap):

```
function _firstChickenIn(uint256 _bondStartTime, uint256 _bammLUSDValue,
                         uint256 _lusdInBAMMSPVault) internal returns (uint256) {
  …
  // From Curve Vault
  uint256 lusdFromInitialYieldInCurve = getAcquiredLUSDInCurve();
  uint256 yTokensFromCurveVault =
    _calcCorrespondingYTokensInCurveVault(lusdFromInitialYieldInCurve);

  // withdraw LUSD3CRV from Curve Vault
  if (yTokensFromCurveVault > 0) {
    _withdrawFromCurveVaultAndTransferToRewardsStakingContract(
     yTokensFromCurveVault);
  }
  …
}

function _withdrawFromCurveVaultAndTransferToRewardsStakingContract(
 uint256 _yTokensToSwap) internal {
  uint256 LUSD3CRVBalanceBefore = curvePool.balanceOf(address(this));
  yearnCurveVault.withdraw(_yTokensToSwap);
  uint256 LUSD3CRVBalanceDelta = curvePool.balanceOf(address(this)) -
    LUSD3CRVBalanceBefore;

  // obtain LUSD from Curve
  if (LUSD3CRVBalanceDelta > 0) {
    uint256 lusdBalanceBefore = lusdToken.balanceOf(address(this));
    // Dedaub: attackable!
    curvePool.remove_liquidity_one_coin(LUSD3CRVBalanceDelta,
                                        INDEX_OF_LUSD_TOKEN_IN_CURVE_POOL, 0);

    uint256 lusdBalanceDelta = lusdToken.balanceOf(address(this)) -
                               lusdBalanceBefore;
    _transferToRewardsStakingContract(lusdBalanceDelta);
  }
}
```

If the yield that's being swapped is ever high, an attacker can tilt the Curve pool to make LUSD appear very expensive and make the `remove_liquidity_one_coin` return much lower amounts, effectively making the contract spend its Curve LP tokens for nothing in return. (The attacker will receive the proceeds upon restoring the pool.)

This attack is currently highly unlikely because

- a real first-ever chicken in will find no past yield
- a subsequent `firstChickenIn` is unlikely to occur because there is significant financial motivation for the last bLUSD holders to not redeem, if there is yield. So it is unlikely that the balance of bLUSD will ever again drop to zero.

OTHER/ ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	Simplification in exponentiation	**RESOLVED**

Iterative exponentiation by squaring (ChickenMath::decPow) could be simplified slightly from:

```
      while (n > 1) {
            if (n % 2 == 0) {
            x = decMul(x, x);
            n = n / 2;
            } else { // if (n % 2 != 0)
            y = decMul(x, y);
            x = decMul(x, x);
            n = (n - 1) / 2;
            }
      }
```

to:

```
      while (n > 1) {
            if (n % 2 != 0) {
            y = decMul(x, y);
            }
            x = decMul(x, x);
            n = n / 2;
      }
```

We only recommend this change for reasons of elegance, not impact.

A2	Assert unnecessary	RESOLVED

There is an assert statement in _firstChickenIn that is currently unnecessary.

```
function _firstChickenIn(...) internal returns (uint256) {
      assert(!migration);
```

The function is currently only called under conditions that preclude the assert:

```
   if (bLUSDToken.totalSupply() == 0 && !migration) {
      lusdInBAMMSPVault =
        _firstChickenIn(bond.startTime, bammLUSDValue, lusdInBAMMSPVault);
   }
```

More generally, although there is a long-standing software engineering practice encouraging asserts for “circumstances that should never arise”, we discourage their use in deployed blockchain code, since asserts in the EVM do have a run-time cost.

A3	Unnecessary computation of minimum	DISMISSED, INVALID

		(will remove)

The "minimum" computation in the code below has a pre-determined outcome:

```
function shiftLUSDFromSPToCurve(uint256 _maxLUSDToShift) external {
  …
  (uint256 bammLUSDValue, uint256 lusdInBAMMSPVault) = _updateBAMMDebt();
  uint256 lusdOwnedInBAMMSPVault = bammLUSDValue - pendingLUSD;

  // Make sure pending bucket is not moved to Curve, so it can be
  // withdrawn on chicken out
  uint256 clampedLUSDToShift = Math.min(_maxLUSDToShift,
                                        lusdOwnedInBAMMSPVault);

  // Make sure there's enough LUSD available in B.Protocol
  clampedLUSDToShift = Math.min(clampedLUSDToShift, lusdInBAMMSPVault);
  // Dedaub: the above is unnecessary. _updateBAMMDebt has its first
  //  return value always be <= the second. So, clampedLUSTToShift
  // (which is <= _bammLUSDValue) will always be <= lusdInBAMMSPVault
```

A4	Mistake in README.md	INFO (RESOLVED)

Under the section 'Shifter functions::Spot Price Thresholds', the conditions under which shifts are allowed are incorrect. The correct conditions should read:

- Shifting from the Curve to SP is possible when the spot price is < x, and must not move the spot price above x.
- Shifting from SP to the Curve is possible when the spot price is > y, and must not move the spot price below y.

A5	Compiler bugs	INFO (RESOLVED)

The code is compiled with Solidity 0.8.10 or higher. For deployment, we recommend no floating pragmas, i.e., a specific version, so as to be confident about the baseline guarantees offered by the compiler. Version 0.8.10, in particular, has some known bugs, which we do not believe to affect the correctness of the contracts.

CENTRALIZATION ASPECTS

The design of the protocol is highly decentralized. The creation of bonds, chickening in/out and redemption of bLUSD tokens is all carried out without any intervention from governance. The shifter functions, `ChickenBondManager::shiftLUSDFromSPToCurve` and `ChickenBondManager::shiftLUSDFromCurveToSP`, which move LUSD between the Liquity stability pool and the curve pool are also public and permissionless.

The Yearn Governance address holds control of the protocol's migration mode which prevents the creation of new bonds, among other changes. There is no way to deactivate migration mode. Although new users will not be able to join the protocol, all current users will still be able to retrieve their funds, either through `ChickenBondManager::chickenOut` or `ChickenBondManager::redeem`. Yearn governance decisions are voted on by all YFI token holders and are executed by a 6-of-9 Multisig address.

DISCLAIMER

ABOUT DEDAUB