LUSD Chicken Bonds

Smart Contract Security Assessment

Sep. 23, 2022

ABSTRACT

Dedaub was commissioned to audit changes to the LUSD Chicken Bonds protocol. This is a brief audit of changes ("delta audit"), following up on our original audit of the functionality. Context and economic considerations remain unchanged from our earlier audit report. The new functionality concerns incremental improvements, mainly more feature-rich NFTs for bond positions.

SETTING AND CAVEATS

The code and accompanying artifacts (e.g., test suite, documentation) are of excellent quality, developed with high professional standards.

The scope of the audit includes changes to 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
- NFTArtwork/GenerativeEggArtwork.sol
- NFTArtwork/SimpleEggArtwork.sol

The latter two files were reviewed only to ascertain that no security-critical functionality is included. Their functional correctness (in drawing SVG graphics) was not examined at all and is out of scope for this audit. Additionally, the math files, as well as BLUSDToken contain no substantive changes for this delta audit. Therefore, the code changes examined were essentially in BondNFT and ChickenBondManager.

The audit covers changes up to commit 7e32f0bb9b7d5a2b0409f7c30358e034f2e4babb, starting from commit fe32984f920daf50f61ec5544c29282ca6908ef5. The latter is a commit later than our previous audit, but prior to our last review of revisions that address the previous audit's items.

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.

As an audit of changes, some context may be missing: the whole code base and economic model were not revisited. The emphasis was on new functionality and clear security threats.

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:

[No high severity issues]

MEDIUM SEVERITY:

[No medium severity issues]

LOW SEVERITY:

ID	Description	STATUS
L1	NFT generation randomness can be manipulated	**DISMISSED**

NFTs for chicken bonds are generated based on data such as the current permanentLUSD amount (binned in NFT_RANDOMNESS_DIVISOR-sized bins) but also the current block.timestamp:

```
// ChickenBondManager.sol
function createBond(uint256 _lusdAmount) public returns (uint256) {
  ...
      (uint256 bondID, uint80 initialHalfDna) =
        bondNFT.mint(msg.sender, permanentLUSD / NFT_RANDOMNESS_DIVISOR);

// BondNFT.sol
function mint(address _bonder, uint256 _permanentSeed) external returns
(uint256, uint80) {
  ...
  uint256 tokenID = totalSupply() + 1;
  ...
  uint80 initialHalfDna = getHalfDna(tokenID, _permanentSeed);
  ...
  return (tokenID, initialHalfDna);
}

function getHalfDna(uint256 _tokenID, uint256 _permanentSeed) internal
view returns (uint80) {
  return uint80(uint256(keccak256(abi.encode(_tokenID,
                                             block.timestamp,
                                             _permanentSeed))));
}
```

The developers seem concerned about the possibility of a user manipulating the NFT:

```
// This is the minimum amount the permanent bucket needs to be increased
// by an attacker (through previous chicken in or redemption fee),
// in order to manipulate the obtained NFT. If the attacker finds the
// desired outcome at attempt N,
// the permanent increase should be N * NFT_RANDOMNESS_DIVISOR.
// It also means that as long as Permanent doesn't change in that order of
// magnitude, attacker can try to manipulate
// only changing the event date.
uint256 public immutable NFT_RANDOMNESS_DIVISOR;
```

Such manipulation will primarily change the shape of the graphical representation (and underlying binary data) of the NFT. It is, therefore, unlikely that there will be strong motivation to manipulate, for gain. We note, however, that manipulation is certainly possible, in several ways:

- An attacker can wait for the right `block.timestamp`.
- An attacker can minorly influence the `block.timestamp` if they are a validator.
- An attacker can reorder transactions so that their own gets a different `tokenId`, if they are a block creator. (This is more limited but also more serious than just manipulating the extra data, since the `tokenId` itself is a true identity of the NFT and immutable.)

Much of this manipulation depends on having an algorithmic way to recognize “desirable” NFTs or ids.

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	Sub-optimal gas behavior in BondExtraData	**OPEN**

The `BondExtraData` struct is designed to fit in one storage word:

```
struct BondExtraData {
      uint80 initialHalfDna;
      uint80 finalHalfDna;
      uint32 troveSize;  // Debt in LUSD
      uint32 lqtyAmount; // Holding LQTY, staking or deposited into Pickle
      uint32 curveGaugeSlopes;  // For 3CRV and Frax pools combined
}
```

(We note, in passing, that the uint32 amounts are rounded down, so different underlying amounts can map to the same recorded amount. This seems like an extremely minor inaccuracy but it also pertains to issue L1, of NFT manipulation.)

The result of fitting the struct in a single word is that the following code is highly suboptimal, gas-wise, requiring 4 separate SSTOREs, but also SLOADs of values before the SSTORE (so that unaffected bits get preserved):

```
function setFinalExtraData(address _bonder, uint256 _tokenID, uint256
_permanentSeed) external returns (uint80) {
  …
  idToBondExtraData[_tokenID].finalHalfDna = newDna;
```

```
…
idToBondExtraData[_tokenID].troveSize =
   _uint256ToUint32(troveManager.getTroveDebt(_bonder));
…
idToBondExtraData[_tokenID].lqtyAmount =
   _uint256ToUint32(lqtyToken.balanceOf(_bonder) +
                    lqtyStaking.stakes(_bonder) + pickleLQTYAmount);
…
idToBondExtraData[_tokenID].curveGaugeSlopes =
   _uint256ToUint32((curveLUSD3CRVGaugeSlope + curveLUSDFRAXGaugeSlope)
   * CURVE_GAUGE_SLOPES_PRECISION);
```

We recommend using a memory record of the struct, reading its original value from storage, updating the 4 fields in-memory, and storing back to `idToBondExtraData[_tokenID]`.

The Solidity compiler could conceptually optimize the above pattern, but current versions do not even attempt such an optimization in the presence of internal calls, let alone external calls. (We also ascertained that the resulting bytecode is suboptimal under the current build settings of the repo.)

A2	Possibly extraneous check	**INFO**

Under the, relatively reasonable, assumption that `MIN_BOND_AMOUNT` is never zero, the first of the following checks would be extraneous:

```
function createBond(uint256 _lusdAmount) public returns (uint256) {
   _requireNonZeroAmount(_lusdAmount);
   _requireMinBond(_lusdAmount);
```

A3	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.

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Watchdog.

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.