Dedaub

Security Technology for Smart Contracts

Armor arShield

Smart Contract Security Assessment

Date: June 21, 2021

Abstract

Dedaub was commissioned to perform a security audit of the Armor ArShield contracts, at commit hash `56f94d805a1dd2760f81ce6d40ead15b6293e644`. The audit is explicitly about the code. Two auditors worked over this codebase over a week.

Setting and Caveats

The audited code base is of average size, at around 2.4KLoC (excluding test and interface code). The audit focused on security, establishing the overall security model and its robustness, and also crypto-economic issues. Functional correctness (e.g., that the calculations are correct) was a secondary priority. 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.

Architecture and Recommendations

Armor arShields are a convenient economic abstraction over standard protocol tokens that encapsulate underlying insurance over the protocol. This insurance is sold by Armor.Fi (through arNFT stakers) and ultimately underwritten by Nexus Mutual.

A user can deposit their token in a supported arShield to create an arToken. Such a token however can be exposed to risk of failure from multiple underlying protocols. For instance, a Curve 3Pool token is exposed to the risk of MakerDAO in addition to Curve itself. In addition, when a curve 3Pool token is used as collateral in another lending protocol, the token that is received is exposed to an additional protocol risk. Because of this each arShield makes use of multiple CoverageBase contracts. These contracts interact with Armor Core to cover arShields. The relation between these is many-to-many. This high level architecture ensures that the protocol can remain relevant in view of the increasing complexity of newer DeFi protocols.

An interesting feature of the protocol is the way in which the insurance premium is paid. This is done through sporadic liquidations of the underlying protocol tokens at the rate supplied by a Chainlink Oracle by external liquidation bots. Thus, the arTokens lose their value with respect to their underlying protocol token, except in cases of hacks, where it is expected that these tokens pay out dividends if a payout is confirmed by Nexus.

Compared to the Armor core protocol (and Nexus Mutual), there are further centralization points in the arShield protocol. The arShield protocol relies heavily on governance actions for important decision making and parameterization. Governance is responsible for accepting/rejecting newly created arShields, unlocking a shield's contract by confirming/rejecting hack notifications, keeping track of minted arToken amounts after a hack notification and banning payouts. In addition, governance can adjust or update important parameters of the protocol such as the bonus amount given to liquidator bots.

In addition to some centralization points, an external keeper bot also needs to regularly update the amount covered and calls functions to update state variables which keep track of the coverage cost-per-time. One of our suggestions is to automate coverage cost-per-time variable updates so as to eliminate the possibility of an attack due to infrequent keeper's actions. The protocol needs to keep track of several fees and costs, for which a number of mathematical formulas with a moderate level of complication need to be checked. The mathematical formulas themselves are easy, but the stateful nature in which these interact with each other can lead to serious issues, some of which we have identified in this audit. Thorough testing for all these formulas is strongly suggested. In addition, refactoring the code such that state changes happen in fewer places, with fewer function calls, will facilitate the understanding and evolution of the code.

Finally, the documentation states that arTokens are envisioned to become a replacement of the protocol tokens themselves and traded on conventional decentralized exchanges such as Uniswap. On the one hand, the stability of the price of these tokens against long tail events such as hacks is a good thing for AMMs, since tokens traded against arTokens in LPs will not be drained if the protocol token suddenly drops in value due to a hack. On the other hand, the sporadic liquidations at a fixed price that is always in favor of the liquidator would increase the arbitrage opportunities of these tokens. This may possibly make them less appealing for use in AMMs.

Vulnerabilities and Functional Issues

This section details issues that affect the functionality of the contract. 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: 1) User or system funds can be lost when third party systems misbehave. 2) DoS, under specific conditions. 3) Part of the functionality becomes unusable due to programming error.
Low	Examples: 1) Breaking important system invariants, but without apparent consequences. 2) Buggy functionality for trusted users where a workaround exists. 3) Security issues which may manifest when the system evolves.

Issue resolution includes "dismissed", by the client, or "resolved", per the auditors.

Critical Severity

[No critical severity issues]

High Severity

Id	Description	Status
H1	**Inaccurate tokens conversion**	**Resolved**

In the following code snippet taken from `arShield::liqAmts` amounts `ethOwed` and `tokensOwed` are supposed to represent equal value.

```
      ethOwed = covBases[_covId].getShieldOwed( address(this) );
      if (ethOwed > 0) tokensOwed = oracle.getTokensOwed(ethOwed,
address(pToken), uTokenLink);

      tokenFees = feesToLiq[_covId];
      tokensOwed += tokenFees;
      require(tokensOwed > 0, "No fees are owed.");

      uint256 ethFees = ethOwed > 0 ?
                            ethOwed
                            * tokenFees
                            / tokensOwed
                        : getEthValue(tokenFees);
      ethOwed += ethFees;
```

However, code line

```
      tokensOwed += tokenFees;
```

is misplaced resulting in an underpriced ethFees computation.
We suggest that it be altered as follows:

```
      ethOwed = covBases[_covId].getShieldOwed( address(this) );
      if (ethOwed > 0) tokensOwed = oracle.getTokensOwed(ethOwed,
address(pToken), uTokenLink);

      tokenFees = feesToLiq[_covId];
      require(tokensOwed + tokenFees > 0, "No fees are owed.");
```

```
        uint256 ethFees = ethOwed > 0 ?
                                ethOwed
                                * tokenFees
                                / tokensOwed
                            : getEthValue(tokenFees);
        ethOwed += ethFees;
        tokensOwed += tokenFees;
```

for accuracy.

H2	Duplicate subtraction of fees amount	Resolved

In `arShield::payAmts` the new ethValue is calculated as follows:

```
        // Ether value of all of the contract minus what we're liquidating.
        ethValue = (pToken.balanceOf( address(this) )
                    // Dedaub: _tokenFees amount is subtracted twice
                    - _tokenFees
                    - totalFeeAmts())
                    * _ethOwed
                    / _tokensOwed
```

totalFeeAmounts() also considers all liquidation fees, resulting in _tokenFees being subtracted twice. This can cause important harm to the protocol, as the total value of coverage purchased is underestimated.

Medium Severity

Id	Description	Status
M1	**Possible non-updated value of `totalCost` when `checkpoint()`ing**	**Dismissed (it is decided that `updateCoverage()` will be called frequently enough)**

Function `CoverageBase::updateShield` updates the `totalEthValue` of the purchased coverage and then calls `checkpoint()` to update the cost-tracking related variables

```
      totalEthValue = totalEthValue
                        - uint256(stats.ethValue)
                        + _newEthValue;

      checkpoint();
```

Where

```
function checkpoint()
      internal
{
      cumCost += costPerEth * (block.timestamp - lastUpdate);
      costPerEth = totalCost
                   * 1 ether
                   / totalEthValue;
      lastUpdate = block.timestamp;
}
```

Variable `totalCost` gets updated by external calls to `updateCoverage()`:

```
function updateCoverage()external
{
      ArmorCore.deposit( address(this).balance );
      ArmorCore.subscribe( protocol, getCoverage() );
      totalCost = getCoverageCost();
      checkpoint();
```

```
}
```

where an external keeper is responsible to call it whenever needed to maintain up-to-date information.
However this gives the ability to an attacker to front-run the updateCoverage transaction so as to have consecutive updateShield() happening without totalCost being updated in the meantime. For example, the attacker could mint arTokens by depositing a large amount of pTokens, then inspect the network for an updateCoverage() transaction so as to front run it with redeem and liquidate transactions. The attacker can benefit by lowering the cost of coverage, resulting in underpaid coverage. This is because totalCost affects prices costPerEth and cumCost. The higher the frequency that updateCoverage() is called the lower the possibility of an effective attack of this kind. If updateCoverage() is called frequently enough then it's most probably an unprofitable attack due to fees (both network fees and protocol fees), but again there is also a dependency on the amount of coverage bought by the user/attacker.
For enhanced security, we suggest that `totalCost` be updated upon each coverage amount change, i.e. the above code snippet from `updateShield()` to become:

```
      totalEthValue = totalEthValue
                      - uint256(stats.ethValue)
                      + _newEthValue;

      checkpoint();
      // Dedaub: update totalCost since totalEthValue changed
      totalCost = getCoverageCost();
```

Low Severity

[No low severity issues]

Other/Advisory Issues

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

Id	Description	Status
A1	**Unintuitive variable name**	**Resolved**

We suggest that variable `totalCost`

```
// Current cost per second for all Ether on contract.
uint256 public totalCost;
```

is renamed to `totalCostPerSec` for clarity.

A2	**Old version of SafeMath library**	**Resolved**

The code of the SafeMath library included is of an old version of compiler (< 0.8.0) being set to pragma solidity 0.8.4. However, compiler versions of 0.8.* revert on overflow or underflow, so this library has no effect. We suggest ArmorCore.sol not use this library and substitute SafeMath operations to normal ones, as well as SafeMath.sol contract be completely removed.

A3	**Misleading comment**	**Resolved**

In arShield.sol function `confirmHack` has a misleading @dev comment:

```
 /**
     * Dedaub: used by governor, not controller
     * @dev Used by controller to confirm that a hack happened, which
then locks the contract in anticipation of claims.
    **/
    function confirmHack(
        uint256 _payoutBlock,
        uint256 _payoutAmt
    )
      external
      isLocked
      onlyGov
```

A4	Extra protection of refunds in `arShield`	Resolved

Function `CoverageBase::DisburseClaim` is called by governance and transfers ETH amount to a selected `_arShield`, that is supposed to be used for claim refunds.

```
    /**
     * @dev Governance may disburse funds from a claim to the chosen
shields.
     * @param _shield Address of the shield to disburse funds to.
     * @param _amount Amount of funds to disburse to the shield.
    **/
    function disburseClaim(
        address payable _shield,
        uint256 _amount
    )
      external
      onlyGov
    {
        require(shieldStats[_shield].lastUpdate > 0, "Shield is not
authorized to use this contract.");
        _shield.transfer(_amount);
    }
```

We suggest that an extra requirement be added, checking that `_shield` is locked. In the opposite case the ETH amount transferred to the arShield contract as refunds can be immediately transferred to the beneficiary. arShield's contract locking/unlocking and `disburseClaim()` are all government-only actions, however this suggestion ensures security in case of false ordering of the governance transactions.

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 status 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, as well as a public bug bounty program.

About Dedaub

Dedaub offers technology and auditing services for smart contract security. The founders, Neville Grech and Yannis Smaragdakis, are top researchers in program analysis. Dedaub's smart contract technology is demonstrated in the contract-library.com service, which decompiles and performs security analyses on the full Ethereum blockchain.