DEDAUB.COM

Armor RCA

Smart Contract Security Assessment

10.02.2022

ABSTRACT

Dedaub was commissioned to perform an audit on the Armor.fi RCA (reciprocally covered assets). The Armor.fi RCA protocol is a DeFi insurance protocol that has a novel funding model compared to previous versions of Armor. Rather than relying on insurers, or stakers to cover losses, the users of the system collectively cover other users in the event that a single protocol is hacked. Since hacks are long-tailed events it is expected that since there will be many many different pools

No Critical vulnerabilities were found, but two high vulnerabilities were found. The audit did not include OpenZeppelin library code, most notably the Merkle verifier and the Convex Shield. The audit applies to commit hash: `88bfc89d81e85a6c5d010654a4eec1120722984e`.

The audit methodology involved careful review of (currently closed-source) sources, static analysis of compiled contracts and inspection of off-chain systems expected to interact with Armor RCA.

Centralization Aspects

As is common in many new protocols, the maintainers of the smart contracts yield considerable power over the protocol. The power of the governance includes the ability to liquidate all deposits on the protocol by setting the amounts to be liquidated for each shield. A centralized Oracle can set any price it wants for any asset.

The protocol's Governance can set many parameters that affect the user's funds, namely the apr, discount, and withdrawal delay. In addition, the "guardian" of the protocol can set the amount of funds that are paused (to be used following a hack) that directly affects withdrawals (see L3). When this happens, whoever withdraws their tokens loses paused% (no limits in code prevent it from reaching 100%) of the underlying tokens, but these are distributed to all other users after funds are un-paused.

Security Opinion

The audit was conducted on an in-development version of Armor RCA. This has allowed Dedaub to make suggestions at an early stage, whilst the system is still under development. On the other hand, at this stage of development the system is not yet fully functional, lacks documentation or a complete test suite. Although a security audit may reveal a number of functional issues, it is more intended on finding security issues in an adversarial environment. In order to find functional issues, further testing of the application under many different settings needs to be conducted. There is also an additional risk that new bugs may be introduced before the final project is deployed. Further audits for the rest of the project after testing and staging, but prior to final deployment, will considerably lower these risks.

The main economic threat with the architecture of Armor RCA (H2) is the centralized Oracle that is expected to become operational whenever either a hack is detected or fees are levied, and liquidation is initiated. Once this is invoked, a small percentage of all assets covered by the system will be able to be liquidated. Considering the Armor RCA is expected to cover hundreds of derivative assets (e.g., staked Yearn assets), maintaining a reasonable price of all assets will require a substantial amount of (centralized) infrastructure, and is hard to get right. The other main issue is with “Zaps” (H1), which once implemented could potentially allow attackers to steal some withdrawals through classic flashloan attacks.

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“) is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e., full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often trusted the code’s calculations and interactions, in the absence of any other specification. Functional correctness relative to low-level calculations (including units, scaling, quantities returned from external protocols) is generally most effectively done through thorough testing.

VULNERABILITIES & 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: -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", by the client, or "resolved", per the auditors.

CRITICAL SEVERITY:

[No critical severity issues]

HIGH SEVERITY:

ID	Description	STATUS
H1	Withdrawn tokens may be lost through flash loan attacks	**CLOSED**

The function `RcaController::redeemFinalize()` allows any unauthorized user to “zap“ withdrawn tokens using any available zapper contract. It is presumed that, once implemented, some of these zapper contracts would swap on an AMM or stake / unstake from an LP. This allows attackers to atomically tilt respective pools used by the zapper contract to extract value out of these withdrawals.

Note: Upon our re-review of the code, the described flash loan attack is fixed. However, it should be noted that the user is responsible for calling a zapper contract with the appropriate parameters in order to avoid MEV attacks.

ID	Description	STATUS
H2	Updates can be DoS’ed via front-running	**CLOSED**

The function `RcaController::_update()` is invoked every time a shield calls the controller (on every user interaction with the shield), to ensure it has fresh/correct parameters:

```
function _update(...) internal {
  IRcaShield shield = IRcaShield(msg.sender);
  uint32 lastUpdate = uint32(lastShieldUpdate[msg.sender]);

  // Seems kinda messy but not too bad on gas.
  SystemUpdates memory updates = systemUpdates;
  if (lastUpdate < updates.treasuryUpdate)
shield.setTreasury(treasury);
  ...
  // Update shield here to account for interim period where APR was
changed but shield had not updated.
  if (lastUpdate < updates.aprUpdate) {
     shield.controllerUpdate(apr, uint256(updates.aprUpdate));
     shield.setApr(apr);
  }
  ...
  lastShieldUpdate[msg.sender] = uint32(block.timestamp);
}
```

However, its current implementation allows an attacker to front-run an update transaction, by placing a transaction interacting with the shield before the controller's update. In this case, the next time `_update()` is called, the relevant update action will not take place as `lastShieldUpdate[msg.sender]` will be equal to the respective field in the `SystemUpdates` variable.

MEDIUM SEVERITY:

ID	Description	STATUS
M1	Insufficient capacity check	**CLOSED**

In RcaShieldBase::mintTo, the controller is called to verify that the shield has the required capacity before minting the tokens. However, the following check:

```
require(_uAmount < _capacity...)
```

Does not take into consideration the currently "active" underlying tokens. Thus, with the current checks in place, a user can bypass this capacity limit by calling mintTo() multiple times.

M2	Liquidatable amounts cannot be lowered due to underflow	**CLOSED**

In RcaShieldBase::setLiq, if the amount to be liquidated needs to be lowered (for instance because the hacked amount was overvalued) the contract call reverts due to an underflow:

```
function setLiq(
   uint256 _newCumLiq,
   uint256 _updateTime
) external onlyController
{
   ...
   // Do this here rather than on controller for slight savings.
   uint256 addForSale = _newCumLiq - cumLiq;
   amtForSale += addForSale;
   cumLiq = _newCumLiq;
}
```

M3	Use of centralized oracle can be problematic	**DISMISSED**

The system maintains the root of a merkle tree that stores the prices of all the covered assets in eth. These prices are verified in the `purchase()` method of the `RcaController` which is used when the `purchaseU()` and `purchaseRca()` methods of the RcaShieldBase contract are called.

Given the volatile nature of many of the covered assets, stale price values can deviate significantly from their actual prices, allowing arbitrageurs to make profit by reducing the Shield's profits. This loss is bound by the fact that the available amount is the shield's cumulative liquidity (amount lost in hacks) and the protocol's profits (if apr > 0).

LOW SEVERITY:

ID	Description	STATUS
L1	Real APR hard to calculate with liquidations	DISMISSED

The protocol's fees are collected by setting a small percentage of tokens to be liquidated every time each shield is utilized. We believe this will be inefficient. Since these amounts will be low, arbitrage bots will only liquidate these tokens when there's a significant upside, and only when the price is generous. For this reason the real APR is expected to be different to the advertised one.

ID	Description	STATUS
L2	New system parameter values are not sanity checked	CLOSED

The RcaShieldBase contract contains a number of setter functions, callable only by the RcaController, setting some important system parameters. Many of these functions (setPercentPaused(), setDiscount(), setApr()) accept/set values that represent percentages and should be checked to not be over 100% (10000) in order to avoid having a mistake in the controller damaging the system's state. In addition, the value set in setWithdrawalDelay() could have some upper and lower bounds as well to highlight its intended use.

ID	Description	STATUS
L3	Incorrect comment regarding the stored capacities	CLOSED

The comment on the RcaController mentions that capacities are stored in USD.

```
//Merkle root of the amount of capacity available for each ... (in USD)
bytes32 public capacitiesRoot;
```

However the actual values stored are in the native underlying token. This can be verified by looking at the `RcaController::mint` method, which is called by the respective shield to verify that Armor RCA has the required capacity to take on new deposits, the amount that is deposited is compared to a respective amount from a merkle tree containing all the capacities for all shields:

```
require(_uAmount < _capacity, "Not enough capacity available.");
```

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	`RcaController` modifiers can be merged.	**CLOSED**
	The modifiers `RcaController::onlyShield` and `RcaController::update` can be merged into a single modifier.	
A2	Unused arguments in RcaController::redeemFinalize	**CLOSED**
	The function RcaController::redeemFinalize returns whether a contract is a zapper contract. However, this currently only uses a single argument out of the 5 passed to determine this.	
A3	Users lose funds when withdrawing while shield is paused	**DISMISSED**
	There are two mechanisms in the shield contracts that calculate the amount of underlying token (`_uValue()`) or the amount of rca tokens (`_rcaValue()`) when withdrawing or depositing, respectively. The calculations however are inconsistent with each other, at the expense of the user since a lower amount of underlying tokens is returned when withdrawing, if the shield is paused. This is a concern, especially if the protocol is going to be paused for a week for every single hack. The assumption here is that the amount of tokens to be paused is conservative, higher than the amount required to cover the hacked shields.	
A4	Compiler known issues	**INFO**

The contracts were compiled with the Solidity compiler 0.8.11 which, at the time of writing, has no known bugs.

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, 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.