Aragon Court

Security Assessment & Code Review

November 16th, 2019

ARAGON
COURT

Prepared for:
Jorge Izquierdo | Aragon One, CEO
jorge@aragon.one

Prepared By:
Georgios Konstantopoulos | Independent Consultant
me@gakonst.com

Table of Contents

Executive Summary

The Aragon Court is a set of Solidity smart contracts which implement a dispute resolution protocol. It handles disputes which cannot be objectively evaluated by autonomous smart contracts. To perform that task, it leverages a token used for voting along with a set of “jurors” who vote on possible outcomes of a dispute.

The Aragon Court codebase was assessed at commit [0205c2fcafa00e64ecdafad4aec5d7cd1d1e0b27] between November 1st and November 15th 2019.

The architecture [as explained in the documentation] is centered around a `Controller`, which has administrative privileges to all of the smart contracts. It is of paramount importance that the keys controlling the “governor” addresses of the system to be properly managed.

During the first week of the audit, we familiarize ourselves with the smart contracts used for the Aragon Court, and performed an initial review of the `Controller`, `JurorsRegistry`, `Voting` and `Treasury, DisputeManager` contracts.

In the second week of the assessment, we evaluated the low-level libraries (attention was focused on the `JurorsTreeSortition` and `HexSumTree` contracts). In addition, we reviewed the `Subscriptions` module, and the `Court` which puts everything together.

We ran the code against the state of the art open source security auditing tools like Slither, Echidna, Manticore & Myth, which gave mainly informational output and identified no exploitable issues.

The codebase is well documented. The system is modularized in components with clear separation of concerns. The test suite is very extensive and gives high guarantees about the correctness of the code. Inline assembly usage is restricted to well-known helper libraries which are imported from the AragonOS.

Several code improvement suggestions were provided, which however were not related to the security of the system. The identified attacks can be performed either by the privileged users of the system (the governors), or miners. While hard to perform, they should be clearly documented and in the long-term addressed.

Finally, it is recommended that the Aragon team investigates the economic payouts to jurors and other actors of the system, as well as takes into account potential liquidity attacks by actors who repeatedly create low-cost disputes.

Scope & Coverage

A security review of all of the smart contracts, tests included in the aragon-court git repository as of commit 0205c2fcafa00e64ecdafad4aec5d7cd1d1e0b27 was conducted.

The scope did not include any code which is part of the aragonOS repository or any of the other dependencies. We do not review which disputes arrive to the court, neither how they are created. That process is described in a separate document.

The implementation is abstracted over the ERC20 token which is used for voting. We do not make any assumptions about its behavior or correct implementation beyond the specified logic in the standard.

Given that the "Trust Assumptions" documentation are TODO at the specified commit, high difficulty attacks (e.g. miners manipulating block.hash to affect randomness) are also investigated, and suggestions are provided to minimize the impact of such compromises happening.

The questions we seek to answer in this engagement are the following:

1. Is it possible for participants to lose tokens?
2. Can juror tokens be locked forever?
3. Are jurors drafted to proposals in a sufficiently random way?
4. Does the voting schelling game result in an equilibrium and incentivize jurors to always vote on the "correct" outcome?
5. Is the proposed mechanism correctly designed and sufficiently documented?
6. Does the implementation of the system match the specification?
7. Is it possible for jurors to earn fees disproportionately to the tokens they are staking?
8. Is the court always able to come to a decision regarding a proposal?
9. Given the trusted nature of the *governor* addresses, how can we minimize the damage done by them, if their keys are compromised or if they become malicious?
10. Can participants escalate their proposals indefinitely?
11. Can participants create proposals without paying fees?

Audited Components

- **Aragon Court, Controller and its libraries:** The court contains the main access control component business logic of the court. In master-slave terms, the `Controller` is the "master", and each contract inheriting "`Controlled`", acts as the "slave" and points to its controller. In addition, contracts which hold funds inherit "`ControlledRecoverable`", giving permission to the controller to drain all funds stored in the contract. It also contains a "`Clock`" which defines "terms" as a unit of time, and is used as an epoch across the codebase. It contains configuration contracts, which are used to set fees, number of

escalations per dispute, along with durations constants of each phase of the protocol.The `AragonCourt` contract acts as the single entry-point to the system and is what users interface with. It enforces all access control and restricts the interface to simple create/dispute/rule functionalities. When a dispute is created, it can only be drafted after there has been time for system participants to submit evidence. Once evidence is submitted to the arbitrable, the arbitrable may call the court to end the evidence submission period and start the draft.
    - `aragon-court/contracts/court/controller/`
    - `aragon-court/contracts/court/clock/`
    - `aragon-court/contracts/court/config/`
    - `aragon-court/contracts/court/AragonCourt.sol`
- **Arbitration:** Contracts which would like to be arbitrated by the Aragon Court must implement the Arbitrable interface and must also be ERC165 compliant. The Aragon Court implements the Arbitrator interface, and allows it to act as the single entry point to the system.
    - `aragon-court/contracts/arbitration/Arbitrable.sol`
    - `aragon-court/contracts/arbitration/IArbitrable.sol`
    - `aragon-court/contracts/arbitration/IArbitrator.sol`
- **Lib:** Contains libraries which are used throughout the codebase. The `Checkpointing` library allows having variables which maintain history as they get updated. This is useful for tracking the jurors' balances as terms progress. The `HexSumTree` contract implements a hexary sum tree which stores all the active balances of the jurors. `JurorsTreeSortition` is used to sample lists of jurors based on their active balances and a provided pseudorandom seed from the `HexSumTree`. `BytesHelpers` and the libraries under `os/` are commonly seen helpers across Solidity repositories. `PctHelpers` provides utilities for calculating the percentages of `uint256` numbers.
    - `aragon-court/contracts/lib/os/`
    - `aragon-court/contracts/lib/BytesHelpers.sol`
    - `aragon-court/contracts/lib/PctHelpers.sol`
    - `aragon-court/contracts/lib/HexSumTree.sol`
    - `aragon-court/contracts/lib/JurorsTreeSortition.sol`
    - `aragon-court/contracts/lib/Checkpointing.sol`
- **Voting:** The voting module contains an implementation of a Commit / Reveal `ICRVoting` interface, as well as a `CRVoting` contract which implements the interface. Votes on a topic can only be initiated by the court. A party can either commit, leak or reveal a value. Each of these operation calls the `Court` (which implements the `ICRVotingOwner` interface) to ensure that the caller is authorized (i.e. has enough active tokens and is during the proper term) to make the call.
    - `aragon-court/contracts/voting/CRVoting.sol`
    - `aragon-court/contracts/voting/ICRVoting.sol`
    - `aragon-court/contracts/voting/ICRVotingOwner.sol`
- **Treasury:** The treasury's `ITreasury` interface exposes 2 simple functionalities: assignment and withdrawal. The `CourtTreasury` implementation is connected to the `Court` via the controller (and since it is `ControlledRecoverable`, the governor is able to drain any

funds stored in it). Any fees which get deposited to the court during dispute or appeal creation are forwarded to the treasury. Funds from the treasury are allocated to users as rewards for interacting with the protocol and triggering drafts, settlements of appeals, penalties etc.
    - `aragon-court/contracts/treasury/ITreasury.sol`
    - `aragon-court/contracts/treasury/CourtTreasury.sol`
- **Registry:** The registry module contains the `IJurorsRegistry` interface and the `JurorsRegistry` implementation (which also conforms to ERC900 for staking). Users can deposit any amount of the staking token to the registry. Users can signal their desire to become jurors by "activating" any amount of their deposited balance, which locks up their funds and adds an entry with their id and balance to the hexary sumtree (or updates their existing data if they are not performing this action for the first time). Finally, the registry exposes a draft function which returns a list of jurors for the provided input parameters, which eventually gets consumed by the court.
    - `aragon-court/contracts/registry/IJurorsRegistry.sol`
    - `aragon-court/contracts/registry/JurorsRegistry.sol`
- **Disputes:** All of the above components get put together in the `DisputeManager` contract, which implements the full Aragon Court protocol. The flow of interacting with the protocol starts at dispute creation. After a dispute is created, drafts may be performed, until enough jurors are elected for that dispute. The jurors proceed to engage in a commit/reveal round, where they vote (weighted by the amount of tokens they have earlier activated in the registry) on the result of the dispute. After the reveal phase is over, users who are dissatisfied with the result may appeal. If there exists any party willing to take confirm the appeal, a new draft with more jurors begins. If no party confirms the appeal within a time bound, then the appeal's result is considered to be the final result. There exists a maximum number of appeals (configured at deployment time and later modifiable by the governor) which can happen, after which there exists a "final round" where all jurors with more than the minimum required voting amount in the registry can opt-in and vote. Creating a dispute, appealing, and confirming appeals incurs fees which get paid out to the jurors. The fees for appealing and confirming appeals increase with each round. The majority of the jurors who voted on a topic get rewarded with the fees plus any tokens from the minority of the jurors who disagreed with them (the minority effectively gets slashed).
    - `aragon-court/contracts/disputes/DisputeManager.sol`
    - `aragon-court/contracts/disputes/IDisputeManager.sol`
- **Subscriptions:** In order for a dispute to be created, the contract complying to the `IArbitrable` interface is required to have an active "subscription". Subscriptions are counted in "periods" (not to be confused with juror "terms" from the Clock contract, a court term can have many subscription periods). There are three types of periods: regular, delayed and resumed. The governor earns a share of all fees gathered by subscriptions. A user of the system initially subscribes for a number of periods by paying a fee proportional to the number of periods. If their subscription expires, they need to renew it in order to be able to interact with the court. A subscriber cannot prepay more than a certain number of periods, in order to avoid cases where the buy many periods up-front to avoid fee fluctuations. In addition, jurors

only receive fees when a subscription is updated, so the cap ensures that all subscriptions are eventually pre-paid so that fees are distributed to jurors.
    - `aragon-court/contracts/subscriptions/CourtSubscriptions.sol`
    - `aragon-court/contracts/subscriptions/ISubscriptions.sol`

For the purposes of the audit, the repository was forked and detailed comments for the functionality of each Contract have been added in-line. We recommend the authors of the code to refer to these notes as complementary material alongside the report.

Findings

1. The governors are single points of failure

Severity: Informational
Difficulty: High
Type: Access Control
Target: Controlled / Controller

Description

The Controller contract contains the addresses of all modules as well as the configuration options of the Aragon Court. These options can be modified via “governor” accounts. Modules of the Aragon Court inherit from the `Controlled` contract, which specifies a controller at deployment time. Any cross-contract call fetches the address of other modules via the controller. Governors also have privileged access to certain functions of the modules which allow setting module-specific parameters. Contracts which handle funds additionally inherit from the `ControlledRecoverable` contract, which allows the funds governor to withdraw funds at any given point in time (this is added in order to migrate funds during module upgrades). It can be easily seen that the governors have superuser privileges to the system, and can be a single point of failure if compromised.

Exploit Scenario

Aragon gets hacked and the keys of the governors are compromised, or a rogue employee gets access to the governor keys. They proceed to drain the funds from all contracts, change the court’s configuration state variables to have long timeouts and high fees. They then proceed to eject the governors or set them to keys which they control, rendering the system useless. Alternatively, they could swap out one of the existing modules with one of their choices and attempt to interfere with the operations of the court.

Recommendation

Ensure that the governor keys are properly secured in hardware wallets. Instead of the governors being EOAs, make them multisig accounts with keys which are owned by multiple parties. Consider adding a delay in any configuration change call, along with an option to cancel any change proposal, which can only be called by a separate “CancelConfigChange” governor. Ensure that different governors are controlled by different people, so that compromise of 1 governor does not result in full compromise of all governors.

Amendment
This was acknowledged and is documented in PR238.

2. Court may get stuck on disputes where the clock is not updated in time

Severity: High
Difficulty: Medium
Type: Denial of Service
Target: Court / Clock

Description
The Aragon Court's unit of time is a "term" and is implemented via the `CourtClock` contract. The clock advances via a `heartbeat` function. At each heartbeat, the next block's number is stored (before it is mined). The Court requires sampling a set of jurors for a term from the larger juror list via randomness fetched via the `clock.ensureTermRandomness` function. This function fetches the block number stored at the corresponding term's heartbeat, and returns its hash via `blockhash(term.randomnessBN)`. Blockhash is only able to return the block's hash, if the block is within the 255 most recent blocks. As a result, `ensureTermRandomness` for a term will always revert if it is not called within 255 blocks after the term's `randomnessBN` is set, meaning the draft for a dispute will always fail, making the dispute stuck in the `PreDraft` stage.

Exploit Scenario
A new term starts and a dispute is created via `createDispute` at block N. Dispute creator is unable to call draft for 255 blocks due to on-chain congestion or because a miner censored them. Nobody calls `clock.ensureTermRandomness` during that time. User attempts to draft a juror set for their dispute, which reverts since the block randomness cannot be calculated. The dispute is stuck at the `PreDraft` state, with no way for it to be resolved.

Recommendation
Short term, run a daemon which makes a call to `ensureTermRandomness` as soon as a block is mined after each heartbeat. Make sure to set the gas price to a high amount such that the transaction gets confirmed in less than 256 blocks even during chain congestion. Long term, do not rely on blockhash for on-chain randomness.

Amendment
This was acknowledged and is addressed in PR#231.

3. Usage of blockhash for randomness is unsafe against miners who are also jurors

Severity: Medium
Difficulty High
Type: Weak randomness
Target: Court / JurorsRegistry

Description
When a dispute is created, a number of jurors who have a balance over a minimum amount in the next term get drafted to vote on the result of the dispute. The draft is based on randomness fetched from a block's hash. A block's hash is manipulable by miners.

Exploit Scenario
A miner who is a juror and wants to interfere with that dispute, can choose to discard a block that calls the heartbeat function, if that block would not result in them being drafted for the dispute. The miner may lose the block reward for that block, but it may be worth it for them if the dispute is relevant to their interests.

Recommendation
Short term, clearly document that this is a potential attack vector. Long term, switch to a more robust form of generating randomness such as Verifiable Delay Functions or Threshold Relays.

Amendment
This was acknowledged and is documented in PR#239

Code Quality Recommendations

1. Rename variables which shadow function names

The Controlled contract defines a `_config()` method which fetches the config submodule. The Court contract is `Controlled`, but has multiple functions which take a _config argument. This disallows calling the `_config()` method in any of these functions. Consider renaming the function arguments or the function such that this type of behavior is not possible. Note that the compiler is able to detect this when the shadowing occurs in the same files, but does not detect it when the shadowing happens across inherited files. We present a list of the detected areas where shadowing occurred:

- _config:
    - https://github.com/aragon/aragon-court/blob/86de1b45500923a62f62c5924686e0d9317b679d/contracts/controller/Controlled.sol#L139
    - https://github.com/aragon/aragon-court/blob/86de1b45500923a62f62c5924686e0d9317b679d/contracts/court/Court.sol#L798
    - https://github.com/aragon/aragon-court/blob/86de1b45500923a62f62c5924686e0d9317b679d/contracts/court/Court.sol#L813
    - https://github.com/aragon/aragon-court/blob/86de1b45500923a62f62c5924686e0d9317b679d/contracts/court/Court.sol#L910
    - https://github.com/aragon/aragon-court/blob/86de1b45500923a62f62c5924686e0d9317b679d/contracts/court/Court.sol#L928
    - https://github.com/aragon/aragon-court/blob/86de1b45500923a62f62c5924686e0d9317b679d/contracts/court/Court.sol#L1008
    - https://github.com/aragon/aragon-court/blob/86de1b45500923a62f62c5924686e0d9317b679d/contracts/court/Court.sol#L1060
    - https://github.com/aragon/aragon-court/blob/86de1b45500923a62f62c5924686e0d9317b679d/contracts/court/Court.sol#L1079
    - https://github.com/aragon/aragon-court/blob/86de1b45500923a62f62c5924686e0d9317b679d/contracts/court/Court.sol#L1226
    - https://github.com/aragon/aragon-court/blob/86de1b45500923a62f62c5924686e0d9317b679d/contracts/court/Court.sol#L1246
    - https://github.com/aragon/aragon-court/blob/86de1b45500923a62f62c5924686e0d9317b679d/contracts/court/Court.sol#L1263
    - https://github.com/aragon/aragon-court/blob/86de1b45500923a62f62c5924686e0d9317b679d/contracts/court/Court.sol#L1305
    -
- _currentTermId:
    - https://github.com/aragon/aragon-court/blob/86de1b45500923a62f62c5924686e0d9317b679d/contracts/controller/clock/CourtClock.sol#L243

    - https://github.com/aragon/aragon-court/blob/86de1b45500923a62f62c5924686e0d9317b679d/contracts/controller/clock/CourtClock.sol#L226
    - https://github.com/aragon/aragon-court/blob/86de1b45500923a62f62c5924686e0d9317b679d/contracts/controller/config/CourtConfig.sol#L125
    - https://github.com/aragon/aragon-court/blob/86de1b45500923a62f62c5924686e0d9317b679d/contracts/controller/config/CourtConfig.sol#L163
    - https://github.com/aragon/aragon-court/blob/86de1b45500923a62f62c5924686e0d9317b679d/contracts/controller/Controller.sol#L445
    - https://github.com/aragon/aragon-court/blob/86de1b45500923a62f62c5924686e0d9317b679d/contracts/controller/config/CourtConfig.sol#L163
    - https://github.com/aragon/aragon-court/blob/86de1b45500923a62f62c5924686e0d9317b679d/contracts/court/Court.sol#L1232
- _treasury:
    - https://github.com/aragon/aragon-court/blob/86de1b45500923a62f62c5924686e0d9317b679d/contracts/controller/Controlled.sol#L91
    - https://github.com/aragon/aragon-court/blob/86de1b45500923a62f62c5924686e0d9317b679d/contracts/court/Court.sol#L1310

2. Use SafeMath wherever possible

Throughout the codebase the SafeMath library is used to ensure there is no overflow or underflow in mathematical operations. In certain cases (51), the usage of SafeMath is skipped, in order to reduce gas consumption in the consistency check. This is possible because previous checks already ensure that overflow/underflow is not possible. If any of these assumptions end up being incorrect, then there may be arithmetic errors resulting in undefined behavior. Erring on the side of caution, consider using SafeMath everywhere, even if it results in higher gas costs. Run randomized tests on functions which contain such logic, to increase confidence in these operations being safe. Ensure there is no undocumented code where SafeMath is skipped.

3. Fix inconsistent function naming

There are 2 types of functions which can be repeatedly found across the codebase:

- `check*` internal view functions which fetch some value and potentially check invariants about it
- `ensure*` functions perform everything that `check*` does, but are not `view` because they also “ensure” that a variable has been set to its latest state (e.g. randomness for a term), before returning the value.

While generally consistent, the `_checkAdjudicationState` function is not `view`, because it internally makes calls to an `ensure*` function which is inconsistent with the above naming convention. Consider renaming the `checkAdjudicationState` to `ensureAdjudicationState`, or refactor the `ensureCurrentTerm` function out of it so that it can be marked as `view`.

4. Inline over-abstracted helper functions

Across the codebase there are multiple places where external functions are defined as `foo()`. Access is performed there, and the rest of the execution is forwarded to `_foo()`. While this is a generally accepted pattern, sometimes the `_foo()` function proceeds to perform a simple operation (e.g. emit an event), and forwards its execution to another internal helper function, presumably for code reuse. If the internal functions being delegated to are not present more than once (i.e. offer no code reuse benefits), consider inlining them to the body of the function which is calling them.

Examples:

- `_stake` in JurorsRegistry.sol calls `_deposit` and emits a Staked event but `_deposit` is not called anywhere else.
- `_checkValidSalt` calls `encryptVote`, `encryptVote` is not called anywhere else

5. Rename `encryptVote` since it does not perform encryption

`encryptVote` simply hashes its arguments along with a salt. It is by no means encryption and should not be referred as such as it may be misleading. If you choose to inline its functionality as written in the previous suggestion, this issue should be resolved as well, otherwise rename it to `hashVote` or `commitVote`

6. Update documentation for Appeal Collateral Factor and Appeal Confirm Collateral factor

Across the documentation, the collateral factors are referred to be a “multiple of juror fees”. This is not true, as they are a multiple of the “total fees” (ie juror fees + draft fees + settle fees).

7. Improve callback pattern

In `CourtClock`, Instead of using an empty implementation of `onTermTransitioned`, consider making it an unimplemented function. That way, a contract inheriting it must explicitly opt-in implement it as an empty function, rather than implicitly leaving it empty.

8. CourtConfig does not implement IConfig

The `IConfig` interface specifies a method `getConfig`. `CourtConfig` does not implement that method. Compilation only succeeds because `Controller` inherits from `CourtConfig` and it itself implements `getConfig`. Consider moving the `IConfig` interface restriction to `Controller`, rather than having it on `CourtConfig` or removing the `getConfig` function from the interface.

9. Consider enriching inline documentation

The audited fork contains a lot of comments which were written down while the authors of the report were familiarizing with the codebase. Consider adopting some or all of them.

10. Deactivation requests are not processed when staking (vs when activating)

When activating tokens, any existing deactivation requests are cleared. This ensures that the active balance of the juror will not be less than the minimum active balance, in case there was a pending deactivation request. When staking, there is no call to clear deactivation requests. Consider adding a call to `_processDeactivationRequest` in the deposit function, in the conditional branch which would activate the deposited tokens.

11. Clearly document the pass by reference when drafting

The draft function in the `JurorsRegistry` makes a pass by reference call to `_draft`, with jurors which have been initialized inside `draft`. Even though there is 1 return argument from _draft, the returned jurors from draft are populated, due to the pass by reference (instead of passing by value). Clearly document this, as this is a rarely-seen technique in Solidity, which in this case is used to work around stack limit errors.

12. Remove unused variables

There are a few unused error strings in the contracts which should be removed:

- `Checkpointing.MAX_UINT64` (lib/Checkpointing.sol#8)
- `CourtClock.ERROR_TERM_OUTDATED` (controller/clock/CourtClock.sol#9)
- `Controller.ERROR_SENDER_NOT_COURT_MODULE` (controller/Controller.sol#12)
- `Controller.ERROR_ZERO_IMPLEMENTATION_OWNER` (controller/Controller.sol#14)
- `HexSumTreeGasProfiler.BASE_KEY` (test/lib/HexSumTreeGasProfiler.sol#10)
- `HexSumTreeGasProfiler.CHILDREN` (test/lib/HexSumTreeGasProfiler.sol#11)
- `HexSumTreeGasProfiler.ITEMS_LEVEL` (test/lib/HexSumTreeGasProfiler.sol#12)
- `SafeERC20.ERROR_TOKEN_BALANCE_REVERTED` (lib/os/SafeERC20.sol#14)
- `SafeERC20.ERROR_TOKEN_ALLOWANCE_REVERTED` (lib/os/SafeERC20.sol#15)
- `CRVoting.ERROR_OWNER_NOT_CONTRACT` (voting/CRVoting.sol#13)
- `CRVoting.ERROR_COMMIT_DENIED_BY_OWNER` (voting/CRVoting.sol#14)
- `CRVoting.ERROR_REVEAL_DENIED_BY_OWNER` (voting/CRVoting.sol#15)

Amendments

The above recommendations were acknowledged and applied in PR#233