Aragon Contract Audit

Introduction

This audit report was undertaken by @adamdossa for the purpose of providing feedback to Aragon. It has been written without any express or implied warranty.

The review focussed on the Solidity contract code in these repos, and covered:
- possible attack vectors
- mismatches between logic and documentation
- possible code improvements
- confusing functionality or naming

The review was performed using the code tagged with `audit1` at the following commits.

`aragon-core`:
https://github.com/aragon/aragon-core/tree/audit1/contracts

`aragon-apps`:
https://github.com/aragon/aragon-apps/tree/audit1/apps

We used RocketChat to coordinate the review and help resolve questions during the review process.

Overview

Overall the code structure was readable, well-documented and structured coherently and consistently.

Functions largely conform to the best practices at:
http://solidity.readthedocs.io/en/develop/style-guide.html
making the structure and code easy to parse and understand.

The codebase also included comprehensive testing suites, using the Truffle testing framework. The Aragon team also employs a coverage framework and linter process to ensure their code remains consistent and testing comprehensive.

The team was responsive, measured and prompt in their responses and feedback during the process.

General Comments

- There were a few functions that were missing explicit function visibility modifiers. This has been reviewed and corrected (including linter updates) via:
  https://github.com/aragon/aragon-apps/pull/33
- In addition to the below there are a few miscellaneous fixes and improvements that have been

made, including additional testing coverage and minor fixes. These include:
https://github.com/aragon/aragon-apps/commit/49253097831a6f979c68c309ae0c6c00c325af4d
https://github.com/aragon/aragon-apps/commit/c239847065eebb2f2b3ea2dbc264b8a25cd788cd
as well as many other incremental improvements which have been made during and after this audit process.

aragon-core/contracts/kernel

These contracts form the core Kernel implementation. The Kernel manages app and Kernel permissions and provides a layer of indirection to access apps (allowing apps, including the Kernel, to be upgraded as necessary).

Issues

There were a few areas of the documentation which were either out-dated, or possibly confusing. These included:

- description of `Grant Permission` which has now been amended to make the conditions of this operation clearer.
- making the separation between instantiation and initialisation clearer.
- correcting some minor mistakes in the example code for `Create Permission`

These issues have now been resolved in a number of pull requests:
https://github.com/aragon/aragon-wiki/commit/32b6f1cfb396c8974a2be4e925111e33dbe9b275
https://github.com/aragon/aragon-wiki/commit/b2cde71d3faa8212a3124813901b8d5280d93c76
https://github.com/aragon/aragon-wiki/commit/c3a734c651a04052cea0dc80753079a4a6a9ab82

Possible Improvements

Some possible future improvements were identified, specifically:

- events being triggered when permissions were used.
- managing the instantiation / initialisation of contracts (see below section for more details).
- continuing to work on documentation to make the permission framework as clear as possible.

aragon-core/contracts/apps and aragon-core/contracts/common

These contracts contains the logic required for both proxy contracts (which allow the underlying "business logic" contracts to be upgraded) and forwarding (which allows entities to forward transaction scripts to other entities for execution).

Issues

- Pre-Byzantium logic could now be removed from `DelegateProxy` as `returndatasize` opcodes are now available. This has now been resolved via:

https://github.com/aragon/aragon-core/pull/145

- The use of `delegatecall` to execute underlying App logic is a clever way of preserving state and Ether value in the `DelegateProxy` whilst using the code / business logic of an upgradable `App` contract. The below represent areas that could be clarified in the documentation, rather than bugs in the code.
  - Since the `App` code is expected to have state / storage, this imposes a limitation that any new upgraded version of the `App` has a superset of the previous contract which it is replacing.
  - Since state is maintained in the `DelegateProxy` contract, not the underlying `App` contract, this may cause some practical issues with third-party apps such as exchanges or etherscan.io (for example I don't believe the "Read Contract" functionality on etherscan.io would work as the ABI wouldn't have the relevant functions on the `DelegateProxy` contract, and the state on the `App` contract wouldn't reflect updates).
- There were a few areas of documentation which use inconsistent terminology (e.g. scaling vs. forwarding) which have been resolved via: https://github.com/aragon/aragon-wiki/commit/c3a734c651a04052cea0dc80753079a4a6a9ab82
- One of the identified issues related to the instantiation and initialisation of `AppProxy` contracts not being atomic. In the expected use-cases, where `AppProxy` contracts are instantiated and initialised through a factory process (within a single atomic transaction) this is not problematic, but there is some potential for this structure to be abused in unexpected use-cases, allowing a malicious user to front-run `AppProxy` contracts. The Aragon team discussed various options internally and an approach was implemented to allow an optional `_initializePayload` to be passed to the `AppProxy` constructor, which can be used to initialise the contract as part of its instantiation. https://github.com/aragon/aragon-core/pull/148

Possible Improvements

- Given the use of `delegatecall`, the documentation could clarify the intention as to whether `App` contracts are designed to be used as factory contracts, or redeployed for each Aragon organisation.
- It may be sensible to make the `10000` gas in `AppProxy` a variable that can be modified to account for any future changes to opcode gas costs for `delegatecall`.

aragon-apps/apps/voting/contracts

This contract allows votes to take place on arbitrary actions, using the MiniMeToken to determine voting power of token holders.

Issues

- The `VoterState` of `Absent` doesn't seem to be used anywhere - not sure if this is in for some future purpose perhaps, if not it could be removed.
- The `Voting` app automatically executed a vote payload once the outcome of the vote was beyond doubt. Whilst this is useful functionality, it made it hard for a voter to estimate gas requirements and the effect of their vote transaction deterministically (as the outcome depended on whether or not the vote tipped the balance). A boolean parameter has now been introduced to allow a voter

to specify whether or not they want their vote to automatically execute the vote payload (if appropriate) via: https://github.com/aragon/aragon-apps/commit/ae8a92e05c9bb90892da24e09ba1354fe49514ac

Possible improvements

- Currently the duration of a vote is fixed on initialisation of the `Voting` contract - it may be worth considering making this a per-Vote variable instead to allow for both long and short-dated votes. Since the `_minAcceptQuorumPct` can be modified during the lifetime of the contract, it may be that users also want to change `voteTime` as well.
- The event `CastVote` should include the number of votes being cast to allow easier off-chain tracking of vote status. This has now been resolved via: https://github.com/aragon/aragon-apps/commit/8ea8cda2d190a695d2a7ec0d7680c4c8bafd5054
- When modifying the `_minAcceptQuorumPct` through `changeMinAcceptQuorumPct` it may be worth emitting an event for this change. This has now been resolved via: https://github.com/aragon/aragon-apps/commit/03f26441f60a4dedd042749972218fb6dadaa9de
- When creating a new vote through the `forward` method, it may be worth putting a non-blank `_metadata` in to make it clearer to track where the vote has come from. This could incorporate the forwarding entities address.

aragon-apps/apps/token-manager/contracts

This contract manages token actions through the MiniMeToken contract (either wrapping another token, or natively).

Issues

- `canForward` allows holders of tokens which have not yet vested to forward request - this may be intentional, but worth clarifying in the documentation.
- It is technically possible for a single `TokenManager` to be set as the controller for multiple `MiniMeToken` contracts. It may be worth asserting in the `TokenController` functions (`onApprove`, `onTransfer`, `proxyPayment`) that `msg.sender` matches the expected `token` value, to avoid any misconfiguration.

Possible Improvements

- The vesting model follows broadly the Zeppelin approach. In my experience clients have often wanted a simpler model with a single cliff date, upon which all tokens vest. This is not possible to achieve with the implemented approach as the tokens must vest linearly between the start and vesting dates (with nothing vesting before the cliff date).
- It may be worth mentioning in the documentation that it is important to set the `controller` in the `MiniMeToken` to be the `TokenManager` (i.e. that it isn't sufficient to initialise the `TokenManager` with the `MiniMeToken` but that the link needs to go both ways).

aragon-apps/apps/group/contracts

This contract acts as a simple way to permission a group of entities as a single entity.

Issues

- In the documentation, there is a minor grammatical mistake - the phrase "This is needed for identification purposes, given that the only other context the group possesses." should perhaps be "This is needed for identification purposes, given that it is the only context the group possesses.".

aragon-apps/apps/fundraising/contracts

This contract is responsible for raising funds (in any ERC20 token, which could include wrapped Ether) in return for a new ERC20 token.

Issues

- In line 297, it should be `(period.finalPrice != initialPrice)` rather than `(period.finalPrice != 0)`. From the documentation, it says:
  "A period has an initial and final price. If they are not the same, price for a given timestamp is linearly interpolated in function of time." This has now been resolved via:
  https://github.com/aragon/aragon-apps/commit/49253097831a6f979c68c309ae0c6c00c325af4d
- In function `_buy` the calculation of `returnAmount` is subject to rounding errors I believe. i.e. in the `isInversePrice` case, if dividing by `pricePrecision` leads to rounding, then re-multiplying by `pricePrecision` may not yield the expected value. This is an issue that is worth being aware of, but is unlikely to lead to any genuine problems.
- The `initialize` function could check that the `TokenManager` is running in native (rather than wrapped) mode by checking the public variable `_tokenManager.wrappedToken == 0`.
- Having `getCurrentPrice` behave differently on a `call` / `sendTransaction` may be confusing. You could refactor this so that it is genuinely a `constant` function instead. I agree that functionally this works as `transitionSalePeriodIfNeeded` is always called before any other state modifying functions execute.
- `calculatePrice` has an implicit dependency on `transitionSalePeriodIfNeeded` being called before it is used. This could be enforced in the function (e.g. by checking that the current timestamp is between `sale.periodStartTime` and `sale[currentPeriod].periodEnds`. This is redundant in the current code (since `transitionSalePeriodIfNeeded` is always called before `calculatePrice` but would make it harder to make errors in the future, or in derived contracts). This has now been resolved via:
  https://github.com/aragon/aragon-apps/commit/c239847065eebb2f2b3ea2dbc264b8a25cd788cd

Possible Improvements

- Adding a token purchase function which allows a beneficiary address (rather than `msg.sender`) to receive tokens may be useful.

- Having the ability to associate a script to be called on a sale closing may be useful. This script could for example ensure that no more tokens could be minted (effectively capping the issued tokens supply) by revoking `MINT_ROLE` permissions from the `TokenManager`, or allow the `FundRaising` contract to transfer `CREATE_SALES_ROLE` permission to a different entity (i.e. a `VotingApp` associated with the token being issued).
- Maybe worth making the distinction between tokens being raised, and tokens being issued clearer through consistent variable naming. Tracking which variables correspond to `raisedToken` and which correspond to tokens being issued was confusing (maybe consistently call this `raisedToken` and `issuedToken` or similar). i.e. `uint256 returnTokens = ...` could become `uint256 returnRaisedTokens = ...`, `_payedTokens` could become `_paidRaisedTokens`, `_value` in `tokenFallback` could become `_paidRaisedToken`, `sale.minBuy` would become `sale.minRaised` and so on.
- I'm not sure what the motivation to have multiple (potentially concurrent) ongoing sales is? I guess if you wanted to raise from a mix of tokens that would make sense (as each `Sale` could have a different `raisedToken`), but in this case you wouldn't have a consolidated cap etc., which would make it confusing. I can see that it may be required to have a follow-on sale, but wonder whether having each sale being represented by a new FundRaisingProxy may be simpler / clearer.

aragon-apps/apps/finance/contracts

This contract is responsible for tracking transactions (payments or deposits) across an organisations token balances, and managing scheduled payments.

Given the importance of this contract and the value there would be in exploiting it, it may be worth having this specific contract be audited separately by multiple individuals and / or placing a bounty within a live deployment to encourage investigation.

Issues

- Any reason why the minimum payment duration is 2, rather than 1:
  `require(_periodDuration > 1);`
  I can see that you use `settings.periodDuration - 1` as the duration length, but not clear why you don't just use `settings.periodDuration` (with the requirement that `settings.periodDuration > 0`);
- In both `tokenFallback`, and `deposit` the `_token.transferFrom` and `token.transfer` methods could potentially re-enter the `Finance` app. I can't immediately see how this can cause an issue, but worth being aware of.
- It may be worth mentioning in the documentation that payments can be "back-dated", in which case the receiver will be eligible to execute all "back-dated" payments. This has now been resolved via:
  https://github.com/aragon/aragon-wiki/commit/b2cde71d3faa8212a3124813901b8d5280d93c76
- If the budget for a token is reduced during a period (via `setBudget`) it is possible that `_getRemainingBudget` may throw if the amount already spent (`periods[currentPeriodId()].tokenStatement[_token].expenses`) is larger than the new budget

(in which case `settings.budgets[_token].sub(spent)` will throw). It may be better to return 0 rather than throwing in this case.
- `_makePaymentTransaction` could record the payment reference in the corresponding transaction reference.

Possible Improvements

- There are no restrictions on what ERC20 tokens can be used for deposits. This would allow an attacker to create an ERC20 token which always returns `true` on `_token.transferFrom` without actually depositing any tokens. Whilst this isn't really an issue it would be possible to bloat the `transactions` array. Not sure there is really an easy solution to this, although having an optional "whitelist" of allowed token addresses that can be deposited is an option.

aragon-apps/apps/vault/contracts

This contract is used to store an organisations tokens (and wrapped Ether). Any actual balances will be held in the corresponding `AppProxy` contract.

This is a nice and simple contract, with no obvious issues.

Possible Improvements

- Potentially integrating this contract with the `Finance` contracts budget functionality. In other words setting a budget in the `Finance` app could require a corresponding allowance in the `Vault` app for the given token.