PUBLIC

Security Audit

of POA NETWORK's Smart Contracts

September 11, 2018

Produced for

by

Table Of Content

Foreword

We first and foremost thank POA NETWORK for giving us the opportunity to audit their smart contracts. This documents outlines our methodology, limitations, and results.

– ChainSecurity

Executive Summary

The POA NETWORK smart contracts have been analyzed under different aspects, with a variety of tools for automated security analysis of Ethereum smart contracts as well as expert manual review.

Overall, CHAINSECURITY found that POA NETWORK employs very good coding practices and has clean, well-documented code, although the documentation could be updated. We still succeeded in uncovering several vulnerabilities in the code base which were fixed by POA NETWORK in a timely and very professional manner. CHAINSECURITY also reported more minor issues, recommendations and suggestions which were understood and implemented by the development team, leading to an improved and more secure state of the project.

Scope

POA NETWORK requested a precisely scoped audit, meant to assess the technical foundation of POA NETWORK's project in its current state. To define this scope, CHAINSECURITY listed potential points of failure and agreed with POA NETWORK upon them.

Issues that have been encountered while verifying this specification have been listed, even when they were not explicitly mentioned in the specification. However, this list should not be considered exhaustive with respect to the security of POA NETWORK's smart contracts. CHAINSECURITY strove to verify the points listed here, to provide a report which contents could serve as potential guidelines in the future.

The main specification sections are listed below and a detailed description of the reviewed properties can be found in the issues section.

Included in the scope

- I. Initialization and Key distribution
- II. Validator Set
- III. Upgradeability
- IV. Voting and Governance
- V. Reward System
- VI. Miscellaneous

Out of scope

- Non-smart contract components (e.g. user interfaces of dApps)
- APIs (Contract, Bank, SMS) and sub-protocols (PoPA, PoBA PoSN, PoPN)
- Communications between validators and master of ceremony happening off-chain
- Parity implementation: we assume that the parity client has no bug impacting POA NETWORK affecting the POA network)
- Aura consensus and its finality
- Correctness of genesis block

Audit Overview

Scope of the Audit

The scope of the audit is limited to the following source code files. All of these source code files were received on August 2, 2018[1], and the last updated version introducing the fixes on August 30, 2018[2]:

File	SHA-256 checksum
contracts/BallotsStorage.sol	c9c185077ed33705e5d799b614c9c3ca0079bb37782c3f1976513c482d574b9b
contracts/EmissionFunds.sol	498cd74bea3e3ddf423515f07e121ac0c980fa0e1fbc3dc06293998cdc024c84
contracts/KeysManager.sol	dd7e838c3021be9bc571adcdbac3059e859c59e8b0bad8c469c151f93c6434cf
contracts/PoaNetworkConsensus.sol	102ea7f223d11295bfe621ab19eb517b4cc94644980466ed2ce877ef748947ec
contracts/ProxyStorage.sol	8b056b3d0d3f4ff5942006ec94e305fe6f362808d41a0ae359fef424e8f20bdd
contracts/RewardByBlock.sol	1d16052f69918933454c6bbd7e149ee8a8e4f9cdfb74bf33249e84eb923b2f14
contracts/RewardByTime.sol	e6ab9d0207b469910de34f3ca386cd5a6408c628434dab2dc16303b9d5c48d94
contracts/ValidatorMetadata.sol	906c204a948ff56a3f39d9d3de39afefb53ad0029920ed8af04bf59b631f04e3
contracts/VotingToChangeKeys.sol	89ba7739422aba13f72e98c91f3e07abc11f513d2b26dee32a62b766c14b5517
contracts/VotingToChangeMinThreshold.sol	3559a534cbed654538890211cc55b166fbb5f7bbcbd404c555ec53fb3eb1891d
contracts/VotingToChangeProxyAddress.sol	b80444d6bee5abbb99868989866c954501d3d64d0ddf79a525d97c82de1984b2
contracts/VotingToManageEmissionFunds.sol	3662a883500dd6e034e7c62b051f5eae5efd1f1390e587baaa3199d78108e7ef
contracts/abstracts/VotingTo.sol	4c232a45065a0476cca64118cade1ee6bedae269e39afdc72c2f7f1f855a7652
contracts/abstracts/VotingToChange.sol	51c402ace9ede45be11eeaa41421d67a5475a1f6956e699749f02f8524f26c59
contracts/eternal-storage/EternalStorage.sol	422ffabb744940de9f2c647ec16b0cc5c859e489797e67f5551b93fc075451d6
contracts/eternal-storage/EternalStorageProxy.sol	41d29bcc13f2dce6321dd1742e4d0ca5555a749698e4e32d11295e7e25ededfb
contracts/interfaces/IBallotsStorage.sol	3993e6b10b16935c8a676d1a32c117576bbf28fb37bfff12d55c50fdef56c0e3
contracts/interfaces/IEmissionFunds.sol	d404425c85b18d63e6b7a165348de6f21435c18995d7f9ae72f640ef37fc9a4d
contracts/interfaces/IEternalStorageProxy.sol	10ec8b6fce3562f5584a193e45f44f49e58413443239a18403f75bad4f8ee233
contracts/interfaces/IKeysManager.sol	c86b24472674f8e901c14337b52f0d5cdbf55c07bc673ff6490b31d99213e443
contracts/interfaces/IPoaNetworkConsensus.sol	db467f5991a3038b921b47cd6bb8106ddbf3836434018114bd4efbb3f547c065
contracts/interfaces/IProxyStorage.sol	54bfce44fbc3d318c7da29edbc54a37f088dd3ac397ee4b7537a87aedbfd81b6
contracts/interfaces/IRewardByBlock.sol	803967c46964831af6a645bde9ca756c041c8924208fcf2457480822daf0e6c9
contracts/interfaces/IRewardByTime.sol	debac2d8360dfa5ee5c7c89a399a203d0794171de9375cd282252c8a94ffa5e7
contracts/interfaces/IVotingToChange.sol	47610414a6af876d52895e74639007ed220a1b96a0a77a47598dc706298bb048
contracts/interfaces/IVotingToChangeKeys.sol	a994f33643e00c1a564c25a20b7536814e663c6f02e92d85bd9383f19674e690
contracts/interfaces/IVotingToChangeMinThreshold.sol	ab93ab01ea64dd57dd1302373a50e3e0be55f88cbf2441dee71911b62e1a95db
contracts/interfaces/IVotingToChangeProxyAddress.sol	aeda559976432e629ba0bfb9e7eab164c2b7988ed39328ff8caf0766cfefe216
contracts/libs/SafeMath.sol	a42770d734a449ce0e4448eed3cdef8599314556da4062440d44f859a670742f

Depth of Audit

The scope of the security audit conducted by CHAINSECURITY was restricted to:

- Scan the contracts listed above for generic security issues using automated systems and manually inspect the results.
- Manual audit of the contracts listed above for security issues.

[1] https://github.com/poanetwork/poa-network-consensus-contracts/tree/daa6feed5c7b456d62d486ae0e78c5f333212536
[2] https://github.com/poanetwork/poa-network-consensus-contracts/tree/0c175cb98dac52201342f4e5e617f89a184dd467

Terminology

For the purpose of this audit, we adopt the following terminology. For security vulnerabilities, we specify the *likelihood*, *impact* and *severity* (inspired by the OWASP risk rating methodology[3]).

Likelihood represents the likelihood of a security vulnerability to be encountered or exploited in the wild.

Impact specifies the technical and business related consequences of an exploit.

Severity is derived based on the likelihood and the impact calculated previously.

We categorize the findings into 4 distinct categories, depending on their severities:

- L Low: can be considered as less important
- M Medium: should be fixed
- H High: we strongly suggest to fix it before release
- C Critical: needs to be fixed before release

These severities are derived from the likelihood and the impact using the following table, following a standard approach in risk assessment.

	IMPACT		
LIKELIHOOD	**High**	**Medium**	**Low**
High	C	H	M
Medium	H	M	L
Low	M	L	L

During the audit concerns might arise or tools might flag certain security issues. If our careful inspection reveals no security impact, we label it as ✓ No Issue. If during the course of the audit process, an issue has been addressed technically, we label it as ✓ Fixed, while if it has been addressed otherwise by improving documentation or further specification, we label it as ✓ Addressed. Finally, if an issue is meant to be fixed in the future without immediate changes to the code, we label it as ✓ Acknowledged.

Findings that are labeled as either ✓ Fixed or ✓ Addressed are resolved and therefore pose no security threat. Their severity is still listed, but just to give the reader a quick overview what kind of issues were found during the audit.

[3] https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology

Limitations

Security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a secure smart contract. However, auditing allows to discover vulnerabilities that were overlooked during development and areas where additional security measures are necessary.

In most cases, applications are either fully protected against a certain type of attack, or they lack protection against it completely. Some of the issues may affect the entire smart contract application, while some lack protection only in certain areas. We therefore carry out a source code review trying to determine all locations that need to be fixed. Within the customer-determined time frame, CHAINSECURITY has performed auditing in order to discover as many vulnerabilities as possible.

Best Practices in POA NETWORK's project

Projects of good quality follow best practices. In doing so, they make audits more meaningful, by allowing efforts to be focused on subtle and project-specific issues rather than the fulfillment of general guidelines.

Avoiding code duplication is a good example of a good engineering practice which increases the potential of any security audit.

We now list a few points that should be enforced in any good project that aims to be deployed on the Ethereum blockchain. The corresponding box is ticked when POA NETWORK's project fitted the criterion when the audit started.

- ☑ The code is provided as a Git repository, which allows reviews of potential audit-related code changes.
- ☑ Code duplication is minimal or justified and documented.
- ☑ The code compiles.
- ☑ There are no compiler warnings, or warnings are documented.
- ☑ There are tests.
- ☑ The tests are easy to run.
- ☑ The test coverage is available or can be obtained easily.
- ☑ The output of the build process (including possible flattened files) is not committed to the Git repository.
- ☑ The project only contains audit-related files, or, if not possible, a meaningful separation is made between modules that have to be audited and modules that CHAINSECURITY should consider correct.
- ☑ There are migration scripts.
- ☑ The tests are related to the migration script.
- ☑ The code is well documented.
- ☑ The high-level specification is thorough.
- ☑ There are no getter functions for public variables, or the reason why these getters are in the code is given.
- ☑ Function are grouped together according either to the Solidity guidelines[4], or to their functionality.
- ☑ There is no dead code.

POA NETWORK managed to implement all these points, but the following good practices were also implemented:

- ☑ The default value of all enum types is `Invalid`, which is best practice.
- ☑ No array length can underflow.
- ☑ There is one call to the function send. It is caused after all state changes and it handles failures explicitely.

[4] http://solidity.readthedocs.io/en/v0.4.24/style-guide.html#order-of-functions

Security Issues

In the following, we discuss our investigation into security issues. Therefore, we highlight whenever we found specific issues but also mention what vulnerability classes do not appear, if relevant.

Initialization and Key distribution

1.1 Basic specification ✓ No Issue

- Master of ceremony generates the initial keys, which are written to `KeysManager` contract by him

  Indeed, the `initiateKeys` checks that only the Master of Ceremony is allowed to do so.

- Validators will be required to replace their initial keys with a set of three keys (mining, voting, payout) and these can be only replaced with (successful) ballot proposals
  - No further action can be taken without the keys being replaced by the ones that the master of ceremony does not have
  - We assume that there is a secure channel off-chain to prove that new keys have been generated

  The smart contracts ensure that, without only the initial key, validators cannot fulfill their roles. Thus this initial key has to be replaced by three new keys corresponding to the roles. Once these new keys have been registered, the initial key no longer has any power, thanks to `_setInitialKeyStatus(msg.sender, uint8(InitialKeyState.Deactivated));`.

- Successful removal of the master of ceremony is only possible in case of correct initialization

  The Master of Ceremony is removed only when `_isMoCRemovedPending` is true which is made only by the `PoaNetworkConsensus.removeValidator` function if validators are voting to remove the MoC with the `VotingToChangeKeys` contract. However, the MoC cannot be removed unless all of the 12 initial keys were used.

  The `createKeys` function calls

  `IPoaNetworkConsensus(poaNetworkConsensus()).addValidator(_miningKey, true);` which emits `InitiateChange(blockhash(block.number - 1), pendingList);`.

  This event is picked up by the system and `finalizeChange` is called. This function includes

```
if (_mocPending != address(0)) {
        _moc = _mocPending;
        _mocPending = address(0);
    }
    if (_isMoCRemovedPending) {
        _isMoCRemoved = true;
        _isMoCRemovedPending = false;
    }
```

  Thus, it is called before all 12 validators did register their keys and therefore before the network is completely set. We thus assume that all validators will register.

- 3 validators are needed to start a fully functional network

1.2 No guarantees on 12 initial validators L ✓ Acknowledged

During the verification of section I., CHAINSECURITY found that the property

- 12 keys are generated by the master of ceremony and received by validators.

does not hold, since the only check on the number of initial keys is made in the `initiateKeys` function of the `KeysManager` contract. However, a new `PoaNetworkConsensus` can be initialized with more than 12 validators through the constructor. Hence, no hard on-chain guarantee can be given.

Likelihood: Low
Impact: Low

Fixed: POA NETWORK clarified that the initialization of validators through the constructor will only be used for migration purposes.

Validator Set

2.1 Basic specification ✓ No Issue

- There are 12 initial keys for validators, including the master of ceremony (Ceremony stage)
  - The master of ceremony does not have voting rights
  - Later, more validators can be added (up to 2000)

  The number of 12 validators is only an upper bound, as evidenced by the following snippets:

```
function maxNumberOfInitialKeys() public pure returns(uint256) {
  return 12;
}
```

KeysManager.sol

```
function initiateKeys(address _initialKey) public {
    require(msg.sender == masterOfCeremony());
    require(_initialKey != address(0));
    require(initialKeys(_initialKey) == uint8(InitialKeyState.Invalid));
    require(_initialKey != masterOfCeremony());
    uint256 _initialKeysCount = initialKeysCount();
    require(_initialKeysCount < maxNumberOfInitialKeys());
```

KeysManager.sol

  This second snippet also shows that the master of ceremony's initial key cannot be initiated. This in turn prevents the master of ceremony from having voting rights, given that the initial key has to be activated for a validator to be enfranchised, according to:

```
modifier onlyValidVotingKey(address _votingKey) {
    IKeysManager keysManager = IKeysManager(getKeysManager());
    require(keysManager.isVotingActive(_votingKey));
    _;
}
```

VotingTo.sol

  New keys can be added through `addMiningKey`, `addVotingKey` and `addPayoutKey`, with the caveat that issues will arise if too many keys are used.

- The list of validators can be obtained by users at any time Since `currentValidators` is public in `PoaNetworkConsensus.sol`, its member list can be fetched by anyone.
- There is no noticeable change before `finalizeChange` has been called and no partial change. Through the use of the variable `pendingList`, which contains potential validators up to the call of `finalizeChange`, non-atomic changes are avoided: `finalizeChange` enables all changes in one transaction, provided that the pending list is not too big.
- Only the first submitted change has an impact, until it has been either approved or rejected (no collision between proposed changes).

  Any proposal receives a unique monotically increasing id:

```
    _setNextBallotId(ballotId.add(1));
```

VotingTo.sol

  and proposal interactions only happen through this id, hence the absence of collisions.

2.2 Minority of dishonest validators colluding can be harmful L ✓ Acknowledged

Whether or not a set of validators can have an impact on the system is determined by its cardinal: there is no impact as long as this number is under `getMinThresholdOfVoters`, while any change is accepted beyond.

Given that this threshold is not computed depending on the number of validators, but is voted on, it can be lower or higher than 51% of the validators.

Fixed: POA NETWORK is aware of this limitation.

Upgradability

3.1 Basic Specification ✓ No Issue

CHAINSECURITY verified the following properties related to the upgradeability of POA NETWORK's smart contracts.

- Upgradable contracts consists of two parts: Proxy and Implementation
- The upgradable contract's state will be stored in the proxy
  - This holds. When called through the proxy, implementations can only read and write the state of the proxy. Anyone could call the implementations directly and change their own storage, but that is irrelevant to the behavior of the proxy contracts. The only relevant state of the implementations is their mere existence. There are no calls to `suicide` or `selfdestruct` anywhere.
- The code will be inside the implementation
  - Yes, except for the functions defined in EternalStorage and EternalStorageProxy that cannot be overridden: `version`, `implementation`, `getOwner`, `getProxyStorage`, `renounceOwnership`, `transferOwnership`, `upgradeTo`. Indeed, none of these is reimplemented in implementation contracts.
- An upgrade to a correctly-designed contract results in no loss of information
  - The storage layout is deterministic and documented[5]. Upgradable POA NETWORK contracts and `EternalStorageProxy` all inherit from the same `EternalStorage` which defines the storage fields. No other storage fields are defined in other contracts. New implementations that do the same, and are compiled with a compatible compiler, will be able to use the existing storage easily.
- The proxy will forward all calls directly to the implementation
- After each upgrade the version number inside the proxy will be incremented
  - Yes, although CHAINSECURITY notes that a future implementation could change the version to any value.
- Exceptions inside the implementation will be propagated by the proxy
  - Yes. Reverts and return values of any size are propagated, including in nested calls.
- The proxy is controlled by the central `ProxyStorage`, meaning:
  - Only the `ProxyStorage` can update the implementation
    - Yes. In the special case of the `ProxyStorage`, it can update itself.
  - The `ProxyStorage` will only update when instructed by the Governance Dapp
    - Yes, after the governing contract has been set by `initializeAddresses()`. It cannot change after initialization in this implementation. There are two ways the governance can change: with a new implementation of `VotingToChangeProxyAddress`, or a new implementation of `ProxyStorage` that switches to another governance contract.

[5] `http://solidity.readthedocs.io/en/v0.4.24/miscellaneous.html#layout-of-state-variables-in-storage`

3.2 Correctness of the Eternal Storage Proxy pattern ✓ No Issue

The POA contracts make heavy use of the eternal storage pattern. A drawback of this technique is that it bypasses the Solidity type system, given that storage is accessed either by key strings, or by hashes of arguments. This opens multiple avenues for errors.

To compensate for the lack of type-safety, we checked all accesses to storage manually.

- There are no typos in key strings.
- The correct `*Storage` variables from `EternalStorage` are used.
- There are no implicit type castings in keys or values.
- There cannot be key collisions for the following reasons:
  - All keys start with a constant, and use the same pattern for a given constant.
  - No variable-sized inputs are used in `keccak256` calls.
  - All keys were checked for issues of common-prefix or -suffix.
  - Some keys are in fact a prefix of others, such as `affectedKey*`, `miningKey*`, `minThreshold*`, and `proposedValue`.
  - They are all used in different contexts and do not collide.

CHAINSECURITY also verified the following:

- The delegation code is correct.
- No contract has any storage slot of its own.
- No usage of `selfdestruct`.
- The addresses of the implementations are not used anywhere but in the proxy.
- There are no constructors and initialization is done in various init functions. Only the owner can call these and only once.
- No functions of `EternalStorageProxy` are redefined in implementations.

We conclude that the usage of the eternal storage pattern is correct.

3.3 `ProxyStorage` link can be changed after initialization L ✓ Fixed

During the verification of section III., CHAINSECURITY found that the property

- The link from proxy to `ProxyStorage` can only be initialized in the beginning

does not hold, since

- `ValidatorMetadata` can change its `ProxyStorage`, using a logic similar to a multisig wallet with 3 confirmations.
- Future implementations could change the `ProxyStorage` to any other address.

Likelihood: Medium
Impact: Low

Fixed: POA NETWORK fully removed the functionality allowing to change the `ProxyStorage` address after initialization.

Voting and Governance

4.1 Basic Specification ✓ No Issue

CHAINSECURITY verified the following properties related to the Voting and Governance system of POA NETWORK's smart contracts.

- Once a consensus has been reached, the decision can be fetched by external actors without manipulation
  - This can be done through the `getQuorumState` function. Each voting contract has its own `QuorumStates` as an enum type, which contains the possible outputs of the vote, along with the other states of the ballot.
- Ballots can only be created, voted upon and finalized from valid voting keys
  - All `finalize`, `vote` and `createBallot` functions are guarded with `onlyValidVotingKey` modifiers.
- A validator can vote at most once (even when changing its key)
  - The `isValidVote` function is checked in all `vote` calls, which in turn calls `areOldMiningKeysVoted` and validates this for up to 25 past keys.
- Once a validator's vote has been taken into account, it cannot be removed
  - If the validator is removed, the vote is still taken into account if it took place before the removal
- A validator has all the information needed to vote: there is no information outside of the chain for which an external actor has to be trusted
  - A validator has access to the addresses of other validators, the current voting state, (read) access to all keys, the emission fund address, past votes and thresholds.
- The order in which votes occur does not matter for the result
- A ballot has these properties (independent of type, additional type properties can be present):
  - Start Time (has to be in the future)
  - End Time (has to be bigger than start time)
  - Consensus threshold (immutable once a ballot is created)
    - Default threshold depends on the number of validators at ballot creation time
  - Finalized Marker
  - Quorum State
    - VotingToChange: Invalid, InProgress, Accepted, Rejected
    - VotingToManageEmissionFunds: Invalid, InProgress, Sent, Burnt, Frozen
  - Creator
  - Memo (Metadata String)
- The Ballot will only accept votes between start and end time
  - `isActive` check needs to return `true` for any successful `vote` call.
- The Ballot will only accept votes if it is not finalized
  - `require(!getIsFinalized(_id))` is checked in all voting functions.
- The Ballot can be finalized when all possible validators voted OR the time for this ballot runs out
  - In `VotingToEmissionFunds` finalization is triggered after all validators have voted automatically, otherwise `finalize` can be called when it fulfills the `_canBeFinalizedNow` requirements.
- Master of ceremony does not possess any voting rights
- The following decisions are possible (excluding any other)
  - Voting to add validator keys (validator management ballot)
  - Voting to change validator keys (validator management ballot)

  - Voting to remove validator keys (validator management ballot)
  - Voting to change proxy address (modify proxy ballot)
  - Voting to change minimal threshold needed to pass a vote (consensus threshold ballot)
  - Voting to manage emission funds
- Past votes of changed keys are not revoked
- Smart-contracts are defended from validators' spam attacks (spamming the network with nonsense Ballots)

4.2 Contract Owner Controls Voting H ✓ Fixed

The following item was part of the specification list above but did not hold:

- Consensus has always to be reached for a decision to be taken
  - The function `finalize` can be called by any validator. When this function is called, the number of positive votes is matched against the minimum threshold which is set initially.
  - The result can be obtained through the `getQuorumState` function, which returns an integer that can be cast to an enum type indicating the vote's state. State changes occur through the `finalize` function.
  - However, the `owner` has power to override votes, as described below.

The owners of the `VotingToChangeMinThreshold`, `VotingToChangeKeys`, and `VotingToChangeProxyAddress` can manipulate the votes contrary to the majority opinion. Using the `_migrateBasicOne` function the contract owner can manipulate ongoing or completed votes. This is possible through the following implementation. Here the owner can:

```
254     function _migrateBasicOne(
255         uint256 _id,
256         address _prevVotingToChange,
257         uint8 _quorumState,
258         uint256 _index,
259         address _creator,
260         string _memo,
261         address[] _voters
262     ) internal onlyOwner {
263         require(_prevVotingToChange != address(0));
264         require(!migrateDisabled());
265         IVotingToChange prev = IVotingToChange(_prevVotingToChange);
266         _setStartTime(_id, prev.getStartTime(_id));
267         _setEndTime(_id, prev.getEndTime(_id));
268         _setTotalVoters(_id, prev.getTotalVoters(_id));
269         _setProgress(_id, prev.getProgress(_id));
270         _setIsFinalized(_id, prev.getIsFinalized(_id));
271         _setQuorumState(_id, _quorumState);
272         _setIndex(_id, _index);
273         _setMinThresholdOfVoters(_id, prev.getMinThresholdOfVoters(_id));
274         for (uint256 i = 0; i < _voters.length; i++) {
275             address miningKey = _voters[i];
276             _votersAdd(_id, miningKey);
277         }
278         _setCreator(_id, _creator);
279         _setMemo(_id, _memo);
280     }
```

VotingToChange.sol

- Block certain voters from voting by registering that they already voted.
- Reset the quorum state of a finished vote, e.g. from rejected to accepted.

- Remove voters from the list and thereby allow them to vote again.

As the owner can add new validators or remove honest validators through these votes, it can take over the majority of validators through this vulnerability.
Likelihood: Medium
Impact: High

Fixed: The function `_migrateBasicOne` was removed and the voters, progress and quorum state are now retrieved from the previous ballots or events instead of being passed as arguments. Additionally, the migration function can now only be called before any voting operation, meaning that before any vote is made migration and initialization must be completed.
CHAINSECURITY remarks that this issue was present in contracts not yet deployed and was successfully mitigated before the upcoming deployment.

4.3 Any ballot type is accepted L ✓ Addressed

During the verification of section IV., CHAINSECURITY found that the property

- Each ballot has a type, which has to be in the enum type

does not hold.
In the `VotingTo` contract, a ballot is defined as an `enum`, representing one of the seven possible ballot types[6]. However, when the function `_createBallot` is called to create a new ballot, no checks are performed on the type of the proposed ballot, accepting any value.
Further examples of missing input validation can be found in the `EmissionFunds` contract, where the functions `sendFundsTo`, `burnFunds` and `freezeFunds` do not perform any checks on the `_amount` parameter.
Likelihood: Low
Impact: Low

Fixed: POA NETWORK remarks that `_createBallot` is an internal function, so that it is assumed that the type is checked before a call to it. Indeed, all current voting contracts either pass hard-coded arguments or sanitize the input.

4.4 Any validator can freeze the `EmissionFund` L ✓ Acknowledged

The following item was part of the specification list above but did not hold:

- A deadlock due to unreachable consensus is unlikely, only if many validators are removed before voting while a ballot is open

In `VotingToManageEmissionFunds`, a validator can create a ballot with `_endTime = startTime + 1`, such that it is unlikely that any validator will manage to vote within this short timeframe. Afterwards, the validator can call `finalize`, reaching the branch `if (getTotalVoters(_id)< getMinThresholdOfVoters(_id))` where the `QuorumState` will be set to `Frozen`. Since currently the `Frozen` state does not have any consequences, this is not critical.
Likelihood: Low
Impact: Low

Fixed: POA NETWORK notes that if the validator does that maliciously, other validators can create a ballot to remove that validator from the list of validators or to remove his voting key.

4.5 Theoretical Reentrancy during Ballot Creation M ✓ Fixed

During ballot creation, the following code is executed:

```
202     function _createBallot(
203         uint256 _ballotType,
204         uint256 _startTime,
205         uint256 _endTime,
206         string _memo,
```

[6] `Invalid`, `KeyAdding`, `KeyRemoval`, `KeySwap`, `MinThreshold`, `ProxyAddress` and `ManageEmissionFunds`.

```
207         uint8 _quorumState,
208         address _creatorMiningKey
209     ) internal returns(uint256) {
210         uint256 ballotId = nextBallotId();
211         _setStartTime(ballotId, _startTime);
212         _setEndTime(ballotId, _endTime);
213         _setIsFinalized(ballotId, false);
214         _setQuorumState(ballotId, _quorumState);
215         _setMinThresholdOfVoters(ballotId, getGlobalMinThresholdOfVoters());
216         _setCreator(ballotId, _creatorMiningKey);
217         _setMemo(ballotId, _memo);
218         _setNextBallotId(ballotId.add(1));
```

VotingTo.sol

A fresh ballot ID is chosen in the beginning. The ballot ID is the central identifier for this ballot. To ensure uniqueness, the identifier for the next ballot is incremented in line 218.

However, in line 215 an external contract is invoked. This contract could in theory trigger another (reentrant) call to `_createBallot`. While not possible in the current version, future upgrades might enable such a reentrancy, which would create two ballots with the same ID and thereby violate the uniqueness of ballot IDs.

Likelihood: Low
Impact: High

Fixed: The `ballotId` is now incremented and set before the potentially reentrant call.

Reward System ✓ No Issue

CHAINSECURITY checked the following points related to the POA NETWORK reward system. The subpoints listed below are comments on the fulfillment of the specification.

- Only the "system" (parity's special address) can reward them
  - Both in the `RewardByTime` and `RewardByBlock`, the `reward` function is guarded by the `onlySystem` modifier, such that only the special parity system address can successfully call it.
- External actors know well in advance what the reward will be and can thus make an informed choice about whether they want to become validators
  - Before HF on Nov 2018 the reward is 1 coin, send to the validator who created that block (total 1 coin per block)
    - CHAINSECURITY notes that for each block the validator and the `emissionFund` are rewarded, never just the validator.
  - After HF on Nov 2018 the reward is 1 coin, send to the validator who created that block, and an additional coin for self sustainability (total 2 coins per block), send to a dedicated address
    - While no statements about the fork and how it will proceed can be made, the `RewardByBlock` contract implements the reward distribution correctly, according to the interface described in the parity documentation[7].
- Events informing about a call to the reward function are emitted in correct places
  - In both `RewardBy` contracts, the event `Rewarded(receivers, rewards)` is emitted if the call executes successfully[8].
- The reward of one validator is only impacted by the actions of other validators insofar as it can be removed from the validator pool by the majority
  - The reward amount for a validator is fixed in the constant `blockRewardAmount`. Further, the `reward` function can only be called from the parity special address and the engine does so after a block is produced, rewarding only the block author (i.e. validator). Hence, under the assumption of a bug-free parity engine, the reward for a validator can't be impacted by other validator's actions.

[7] https://wiki.parity.io/Block-Reward-Contract.html
[8] The following limitation holds: https://wiki.parity.io/Block-Reward-Contract.html#limitations

- Reward by time contract only rewards validators, and the reward can be only triggered by the parity engine
  - At least the system is always rewarded.
  - The function `_refreshPayoutKeys` can only retrieve validator keys, which are initialized upon creation of a `PoaNetworkConsensus` or added through voting later.
  - CHAINSECURITY notes that the following calculation is made to compute the number of keys to obtain a reward:

$$k = \frac{t_c - t_{lb}}{\tau} + 1 \tag{1}$$

    where t_c is the current timestamp, t_{lb} the timestamp of the last block and τ a constant threshold parameter. The +1 stems from the fact that the `EmissionFund` is always rewarded. According to the Aura consensus documentation[9], each validator gets an assigned time slot in which they can release a block, so a validator can't artificially stretch $(t_c - t_{lb})$ to increase the number of validators that get a reward with him. Also, if τ is the size of the timeslot described in Aura, only the block producing validator and the `EmissionFund` will be rewarded, with no potential for abuse. Concluding, incentive alignment in `RewardByTime` is not trivial and choice of parameters should be carefully considered before deployment.

Miscellaneous ✓ No Issue

CHAINSECURITY considered the issues listed below while investigating the overall system.

- Keys in the Eternal Storage (one per proxy contract) cannot be chosen freely
  - Keys meant as the computation output of the hash operations
  - Meaning that there can be no collision between independent variables
- Link with data stored outside of the blockchain can be verified (signed)
  - With the exception of the initial keys, all data submitted from off-chain to on-chain storage is passed through the dApps (Validators- and Governance-dApp), by signed transactions. Transactions writing to storage only succeed when in compliance with modifiers guarding the setter functions. Hence, an observer can review all transactions writing to storage, check their sender and verify the data.
- Short address attacks
- Smart-contracts are not accessible for changing state by unauthorized person (not owner, not validator and not other smart-contract from governance scope)

Locked ETH and Locked Tokens L ✓ Fixed

In the current version of the contracts, ETH and ERC-20 tokens can be sent to the contracts. While all contracts can receive tokens, all proxied contracts can receive ETH through a generic fallback function:

```
53     function() public payable {
```

EternalStorageProxy.sol

However, the current contracts have no capabilities to forward or spend such funds. Hence, the funds will be locked inside the contracts. To unlock them, a code upgrade would become necessary.
Likelihood: Medium
Impact: Low

Fixed: The `payable` keyword was removed.

[9] https://wiki.parity.io/Aura.html

Trust Issues

The issues described in this section are not security issues but describe functionality which is not fixed inside the smart contract and hence requires additional trust into POA NETWORK, including in POA NETWORK's ability to deal with such powers appropriately.

Trust into Validators M ✓ Acknowledged

In the design of POA NETWORK, validators jointly have a lot of power. They can potentially misuse this power in various ways. We will elude to some of those issues:

1.1 Mining

In their role as validators, they control the block generation and hence the state of the blockchain. Here, as in other blockchain systems, a simple majority controls the long-term blockchain state. Note, that the size of this majority might be smaller than 50% if more than 2 diverging views exist about the desirable blockchain state.

During mining, validators should act according to the protocol and not favor certain actors (including themselves) while also not disregarding specific other actors.

1.2 Technical competence when upgrading

As the design of POA NETWORK allows multiple central components, such as voting procedures, to be upgraded. In case of an upgrade the new smart contracts have to be carefully evaluated to make sure that no functionality breaks, no vulnerabilities are contained and no additional permissions are granted.

In practice, every validator should have the technical skills to review the proposed upgrades independently, without relying on third-party explanations. Given the upgradability features of POA NETWORK's project, there is no real bound on how complex future upgrades will be, which in turn implies that the minimum required technical skills of validators is unknown as of now.

1.3 Collusion Resistance

A subgroup of validators can collude to damage other validators or to obtain other benefits. For example, they can ignore blocks generated by one validator or vote to remove him, in order to improve their rewards. Even if such a group is not a majority, other validators might be economically incentivized to collaborate with them and would hence indirectly support them.

Additionally, validators can collude during the emission fund voting process to collect emissions. Here, smaller groups might be sufficient due to the structure of the voting process.

Fixed: POA NETWORK is aware of the current trust model and its implications.

Design Issues

The points listed here are general recommendations about the design and style of POA NETWORK's project. They highlight possible ways for POA NETWORK to further improve the code.

Assigning to function parameters L ✓ Fixed

Assigning to function parameters should be avoided. Several occurrences of this can be found in POA NETWORK's code:

- `VotingTo` contract: in the `areOldMiningKeysVoted` function, the argument `_miningKey` is written to.
- `VotingToChangeKeys` contract: In the `checkIfMiningExisted` function the address `_currentKey` is overwritten.

Fixed: New variables were introduced and function parameters are not written to anymore.

Silent failures in case of enum mismatches M ✓ Fixed

As the POA NETWORK contracts are upgradable, current dependencies might be broken in the future. As an example, we highlight the `setContractAddress` function, which iterates over all elements of the `ContractTypes` enum:

```
203         } else if (_contractType == uint8(ContractTypes.ValidatorMetadata)) {
204             IEternalStorageProxy(
205                 getValidatorMetadata()
206             ).upgradeTo(_contractAddress);
207         } else if (_contractType == uint8(ContractTypes.ProxyStorage)) {
208             IEternalStorageProxy(this).upgradeTo(_contractAddress);
209         }
210         emit AddressSet(_contractType, _contractAddress);
211     }
```

ProxyStorage.sol

If however, the supplied `_contractType`, due to an upgrade, does not match any of the tested enum values, the contract just silently fails. A reverting `else` clause, might be more appropriate in this case.
Likelihood: Medium
Impact: Medium

Fixed: POA NETWORK modified the code such that a `bool` variable is returned indicating whether the fields were valid or not, hence silent failures are not possible anymore.

`migrateBasicOne` can exceed block gas limit L ✓ Acknowledged

During the verification of section VI., the property item

- No cases of reaching block gas limit in methods' execution

was violated by the following issue:
The function `migrateBasicOne` is implemented in all three voting contracts: `VotingToChangeProxyAddress`, `VotingToChangeKeys`, and `VotingToChangeMinThreshold`.
While it can only be successfully called by the owner, a provided parameter which is not checked is the `address[] _voters` array. A loop iterates over this array and once its length reaches 340, the function will exceed the block gas limit of 8 million.

Fixed: POA NETWORK notes that the `migrateBasicOne` function will be used to migrate ballots from old contracts to new upgradable contracts until the hard fork. The current ballots do not contain a lot of voters because there are 25 validators for the Core network, so the assumption is that the block gas limit will not be exceeded. For future possible migrations, gas limitations will be taken into account.

`finalizeChange` can exceed block gas limit L ✓ Addressed

During the verification of section VI., the property item

- No unbounded array iterations resulting in a deadlocked state

was violated by the following issue:

If the `finalizeChange` function in the `PoaNetworkConsensus` contract is called while more than 1100 validators are present, the function call will exceed the current block gas limit of 8 million and not succeed. While this number is high, it is below the `maxLimitValidators` limit of 2000 and can lead to a change never being finalized.

Fixed: POA NETWORK explains that the corresponding function is called by the Parity engine which uses "system" transactions for its calls, meaning that the block gas limit for such transactions is equal to 0xFFFF...FFFF (256 bits).

Initializing with 75 validators exceeds block gas limit L ✓ Acknowledged

During the verification of section VI., the property item

- No cases of reaching block gas limit in methods' execution

was violated by the following issue:

If a `PoaNetworkConsensus` is created with 75 validators, the deployment costs is above 8 million, therefore exceeding the current block gas limit. Since a new consensus contract is planned to only be initialized with 12 validators, this is not critical but should be noted to avoid potential deployment failures.

Fixed: POA NETWORK is aware of these limitations.

`RewardByTime` will not `reward` with more than 108 validators M ✓ Addressed

During the verification of section VI., the property item

- No unbounded array iterations resulting in a deadlocked state

was violated by the following issue:

While adding validators can only be done on initialization or after a voting procedure, having 109 current validators already will result in the reward function exceeding the block gas limit of 8 million in the `RewardByTime` contract. This would lead to validators not receiving their reward.

Fixed: POA NETWORK explains that the corresponding function is called by the Parity engine which uses "system" transactions for its calls, meaning that the block gas limit for such transactions is equal to 0xFFFF...FFFF (256 bits).

Eternal Storage Pattern could be more efficient M ✓ Fixed

As part of the used eternal storage pattern, POA NETWORK defines string constants as follows:

```
24      string internal constant VOTERS = "voters";
25      string internal constant VOTING_STATE = "votingState";
```

VotingTo.sol

These constants are then used to calculate the storage offset, as in this example:

```
152     function hasMiningKeyAlreadyVoted(uint256 _id, address _miningKey)
153         public
154         view
155         returns(bool)
156     {
157         return boolStorage[
158             keccak256(abi.encodePacked(VOTING_STATE, _id, VOTERS, _miningKey))
159         ];
160     }
```

VotingTo.sol

However, as the constants are declared as `string`, this computation is relatively inefficient. Instead, the constants could be declared as `bytes32`. According to our measurements, such a redeclaration would save 257 gas per storage access in simple cases with one constant. Additionally, the redeclaration would save a significant amount of gas during deployment.

Fixed: POA NETWORK adopted the proposed changes.

The `IEternalStorageProxy` Interface is not explicitly implemented *L* ✓ Fixed

The `IEternalStorageProxy` interface is not implemented by any of the contracts. Many contracts are, however, cast to this interface inside the `ProxyStorage` contract:

```
181         if (_contractType == uint8(ContractTypes.KeysManager)) {
182             IEternalStorageProxy(
183                 getKeysManager()
184             ).upgradeTo(_contractAddress);
185         } else if (_contractType == uint8(ContractTypes.VotingToChangeKeys)) {
186             IEternalStorageProxy(
187                 getVotingToChangeKeys()
188             ).upgradeTo(_contractAddress);
189         } else if (_contractType == uint8(ContractTypes.
                VotingToChangeMinThreshold)) {
190             IEternalStorageProxy(
191                 getVotingToChangeMinThreshold()
192             ).upgradeTo(_contractAddress);
193         } else if (_contractType == uint8(ContractTypes.VotingToChangeProxy))
                {
194             IEternalStorageProxy(
195                 getVotingToChangeProxy()
196             ).upgradeTo(_contractAddress);
```

ProxyStorage.sol

It seems that `EternalStorageProxy` is expected to implement the `IEternalStorageProxy` interface, which should be made explicit in this case.

Fixed: The `EternalStorageProxy` now correctly implements the `IEternalStorageProxy` interface.

`setProxyStorage` can be called by validators *M* ✓ Fixed

The `setProxyStorage` function in the `PoaNetworkConsensus` contract allows to set the address of the proxy storage, which can be done by any validator or the contract owner. However, a malicious validator can set the `ProxyStorage` address to a contract he controls, hence being able to manipulate the voting process and storage. We recommend to restrict the ability to set this address to only the Master of Ceremony or contract owner in order to avoid such attacks. Additionally, we recommend renaming the `isMasterOfCeremonyInitialized` flag, since it really indicates whether the `ProxyStorage` was initialized or not.

Fixed: Only the Master of Ceremony or an owner can now call `setProxyStorage` and the flag was renamed.

More than 200 active ballots possible L ✓ Acknowledged

The following code suggests that at most 200 active ballots can exist inside the system:

```
103     function getMaxLimitBallot() public view returns(uint256) {
104         return 200;
105     }
```

BallotsStorage.sol

However, this is not quite true. As the creation of new ballots is limited per validator, a validator that recently changed its key can create additional ballots. Due to the specific setting this is, however, highly unlikely to occur.

Fixed: POA NETWORK is aware of these limitations.

Confusing validator variables L ✓ Fixed

The `PoaNetworkConsensus` has two modifiers (`isNewValidator`, `isNotNewValidator`) and one function (`isValidator`) accessing the same variable `validatorsState[_someone].isValidator`. Additionally, all of these carry a very similar naming, which is confusing.

CHAINSECURITY notes that the relation between those can be expressed as: `!isNewValidator(a)== isValidator(a)== isNotNewValidator(a)`, meaning that having only one of them would be enough in POA NETWORK's case.

Fixed: The modifiers `isNewValidator`, `isNotNewValidator` were removed.

Incentive for delayed voting L ✓ Acknowledged

Due to the design of the Aura consensus, validators have economic incentives to delay their voting, when adding and removing validators. This is because exact the time of removal influences their reward payouts. As an example, we will consider the following scenario:

- There are 12 registered validators
- A ballot exists to remove a validator
- This ballot only needs one additional vote to succeed
- Validator 5 wants to vote for the removal, but at the optimal time
- The current Aura step is 125

If validator 5 acts purely short-term focussed, he finds the following situation:

- In the Aura step 125, validator 5 would be the primary in case the validator set remains unchanged (because `125 mod 12 = 5`)
- If validator 5 votes for removal of another validator within step 125, validator 4 becomes the primary due to the changed validator set (because `125 mod 11 = 4`)
- However, if validator 5 votes in the next step, it will be the primary for steps 125 and 126 (because `125 mod 12 = 5` and `126 mod 11 = 5`)
- Therefore, validator 5 has an incentive to delay his vote, which might drag out voting procedures. The incentive, however, is relatively small. It is one block reward.

Above, we have shown that incentives exist to delay voting on removing validators. Analogously, these incentives exist when adding validators.

Likelihood: Low
Impact: Low

Fixed: POA NETWORK is aware of this scenario and considers it unlikely.

Risk of forks due to updated list of validators M ✓ Acknowledged

Due to the design of the Aura consensus, unforeseen forks might exist. As an example we will consider, the following scenario:

- There are 12 registered validators
- A ballot exists to remove a validator 3
- This ballot only needs one additional vote to succeed
- The current Aura step is 125

The first branch of the fork is generated by validator 5:

- In the Aura step 125, validator 5 would be the primary in case the validator set remains unchanged (because `125 mod 12 = 5`)
- Hence, validator 5 creates a valid block for step 125

The second branch of the fork is generated by validator 4:

- Validator 4 votes for the removal ballot and the validator 3 is removed
- Now, in step 125, validator 4 is the primary (because `125 mod 11 = 4`)
- Hence, validator 4 creates a valid block for step 125

As we have shown, two valid blocks exist for step 125. This might occur accidentally, as validator 5 might have not yet received the voting transaction by validator 4. However, it might also occur intentionally, as both validators have an economic incentive to behave this way in order to receive the block reward.
Likelihood: Medium
Impact: Medium

Fixed: POA NETWORK is aware of this scenario and opened a corresponding issue in the Aura repository[10].

[10] https://github.com/paritytech/parity-ethereum/issues/9425

Recommendations / Suggestions

Recommendations regarding the Eternal Storage Proxy pattern

We recommend the following regarding the eternal storage proxy pattern, in addition to the comments made in section III.:

- Several key names are repeated in multiple contracts, for instance `proxyStorage` and `owner`. These and others could be declared in a parent contract and inherited by the child contracts that need it.
- A variant of this design, the Inherited Storage pattern[11], could maintain type-safety while keeping the same structure and upgreadability. In particular, this would allow the following:
  - To remove the generic storage fields from `EternalStorage`.
  - For each POA contract (i.e. `KeysManager`), to create a storage contract that declares the storage fields (i.e. `KeysManagerData`).
  - To make implementations inherit from a storage contract (i.e. `KeysManager` is `KeysManagerData`).
- The storage layout might change in future versions of Solidity, either intentionally or by mistake.
- In future upgrades, there is a risk that a user or attacker uses the new implementation before it is initialized. Multiple approaches are available, such as using only constants, supporting zero-values, storing a new `isInitialized` flag, or always calling an `init` function in `upgradeTo`.

Miscellaneous

☑ Several contracts make use of smaller `uint8` data types. This should only be done if necessary by the contract logic and not to save gas, since the Solidity compiler does not optimize memory layout as expected. If the smaller data type is not explicitly tightly packed (i.e. in a `struct`), smaller data types are more costly because the EVM operates on 256 bit words and higher order bits need to be masked out with an additional operation.

☐ Following best practices, upper case notation should be used for constants, which is not the case in the `RewardByBlock` and `RewardByTime` contracts.

☐ We recommend using inclusive bounds in the `VotingToChange` contract. Concretely, the two checks

- `require(diffTime > minBallotDuration());`
- `require(diffTime <= maxBallotDuration());`

should both test for equality, which would make the code clearer and easier to use.

☑ In the `KeysManager` contract the statement `if (oldMiningKey == 0)` compares an `address` field to an integer. While this works, for consistency reasons the comparison should be made with `address(0)`.

☑ ChainSecurity noticed several comments in the code base mentioning that values must be changed or hardcoded before deployment in the `PoaNetworkConsensus`, `RewardByTime` and `RewardByBlock` contracts. We recommend explicitly tracking these fields and additionally reviewing them to avoid potentially costly mistakes upon deployment.

☐ The following fields are public even though getters are implemented:

```
address[] public currentValidators;
address[] public pendingList;
```

PoaNetworkConsensus.sol

They should be made private or internal instead, to ensure that there is only one way to read their values.

☑ There are several unused imports in different contracts which can be removed:

[11] More examples are given here: `https://blog.gnosis.pm/solidity-delegateproxy-contracts-e09957d0f201`.

- BallotsStorage: IKeysManager
- RewardByBlock: IPoaNetworkConsensus
- VotingToChangeKeys: IKeysManager
- VotingToChangeProxyAddress: IProxyStorage, IBallotsStorage
- VotingToChange: IBallotsStorage

☑ The string variable `fullAddress` is unbounded. It would be better to use some comfortable upper limit, to prevent unbounded gas usages.

☐ The array variable `voters` in `VotingToChange` and `VotingToManageEmissionFunds` is unbounded, which could provoke an out-of-gas exception in some cases. Some bound preventing this should be enforced.

☐ The first `require` is useless given the second one in this snippet:

```
require(_id < nextBallotId());
require(_id == nextBallotId().sub(1));
```

VotingToManageEmissionFunds.sol

☑ In `EternalStorageProxy.upgradeTo`, the name `implementation` is inconsistent and shadows a getter function. It could be renamed to `_newImplementation` for consistency.

☐ The same key `proposedValue` is used for different purposes (an address and a number) and should be split in two for clarity.

☑ It seems that the `this` keyword is used only because of name conflicts in:

- KeysManager
- VotingToChangeMinThreshold
- VotingToChangeProxyAddress
- VotingToManageEmissionFunds

However, it is not the same as a simple call, and it causes a round trip through the proxy as well as `delegatecall` again. We recommend renaming the variables to be different from the function names, and not to use `this`.

☑ The Checks-Effects-Interactions pattern[12] is not properly implemented in the following snippet:

```
uint256 ballotId = nextBallotId();
// ...
_setMinThresholdOfVoters(ballotId, getGlobalMinThresholdOfVoters());
// ...
_setNextBallotId(ballotId.add(1));
```

VotingTo.sol

Here, an effect happens after an interaction (through the `getGlobalMinThresholdOfVoters` function). Given that this is an interaction with a trusted contract, it is not a security risk, but it is good practice to implement this pattern nonetheless, as it makes reentrancy auditing easier, and prevents the introduction of bugs if the interaction is modified to become one with a non-trusted contract. Calling `_setNextBallotId` earlier would suffice in this case.

☑ CHAINSECURITY has the following suggestions regarding the `VotingToManageEmissionFunds` contract:

- The check `require(_startTime > releaseTime);` could be removed, since it is also checked that `currentTime > releaseTime` and `_startTime > currentTime`.
- In the same function, `currentTime > releaseTime` might fail if `releaseTime` was just updated. This can be avoided by allowing for equality.
- The variable `previousBallotFinalized` could be renamed to `noActiveBallotExists` for clarity.
- The visibility of `refreshEmissionReleaseTime` can be restricted from `public`.

[12] https://solidity.readthedocs.io/en/v0.4.24/security-considerations.html

☑ Although the short address attack seems to be a concern for POA NETWORK, the number of functions having address arguments was too high to make fine-grained recommendations in that regard. To limit the risk, address arguments could be put at the end of the argument list of functions, so that (at least) other arguments are not changed through padding. If POA NETWORK wants to solve this problem on-chain more systematically, checks should exist in every function using address inputs (as is done in `sendFundsTo`). On the other hand, it is arguably a problem which should be solved at other layers; on top of this, the next major version of Solidity is meant to solve this problem at the compilation phase.

Post-audit comment: POA NETWORK has fixed some of the issues above and is aware of all the implications of those points which were not addressed. Given this awareness, POA NETWORK has to perform no more code changes with regards to these recommendations.

Disclaimer