MixBytes()

PoA Consensus Audit

[CRITICAL]

Not found

[SERIOUS]

1. ValidatorMetadata.sol#L131

The obsolete mechanism of voting should be removed. Its disadvantages include, firstly, the inability to vote against and thus, in some way, cancel the proposed proxy address, secondly, a low and fixed in the form of a constant (`3`) decision threshold.

Fixed at PR 150.

2. VotingToChangeMinThreshold.sol#L98

If the vote threshold is increased for changing the keys or voting key (KeysManager.sol#L454) is deleted, it is not verified that the total number of voting keys remaining in the system is still greater than the current threshold and voting can in principle be finished (see. VotingToChange.sol#L229). As a result, a denial of service may occur.

Client: Yes, we have an issue created on this topic — it is assumed that if the threshold is higher than the number of validators, then you can vote for changing the implementation of `BallotsStorage`, which stores this threshold value because the voting for the change of implementation in such cases is not blocked. Please comment on it if you still see the problem given the above. I am thinking of solving this issue after the hard fork in order to save time now. Also at the stage of creating a vote for the change of threshold, we have such a test for the number of validators: VotingToChangeMinThreshold.sol#L21 — although this is a test for the number of

mining keys and not voting keys, we do not have validators with an empty voting key (theoretically and technically they might exist, but this is unlikely).

Yes, that's it. Plus, the number of voters, equal to the aforementioned `getProxyThreshold`, is also necessary to successfully complete the voting to change the implementation of `BallotsStorage`. There are no critical issues, the rest are available for you to see and evaluate their probabilities and make decisions.

Client: about `getProxyThreshold` — if there is a shortage of keys to successfully finalise the vote for changing the implementation of `BallotsStorage`, then the validators will be able to create a new vote, just the same, in which the actual value of proxyThreshold will be recorded, which will be enough to finalise the new vote.

[WARNINGS]

1. BallotsStorage.sol#L101

If there are > 200 validators, `getBallotLimitPerValidator` will return zero. This function is used in the function `withinLimit`, which means that no one will be able to create new ballots any more — all changes will be blocked.

This problem is not currently relevant due to an error in the implementation of the `withinLimit` modifier where instead of `<=` there should be `<`, otherwise it is allowed to exceed the limit by 1 for calls to `_createBallot`. If this error is corrected, there will be a blocking of the possibility of voting for changes when the number of validators reaches 200.

Fixed at PR 146.

2. VotingToChange.sol#L94

Here, the number of open ballots is imported only for the current validators. If the validator has opened the ballot, then ceased to be an effective validator (`poa.currentValidators`), and then a `migrateBasicAll` call was made, then the ballot of this validator cannot be closed because it will fail on `assert` when calling `_decreaseValidatorLimit` during `_finalizeBallot`.

Fixed at PR 145.

3. VotingToChangeKeys.sol#L329

It is possible to create two or more ballots for deleting the same mining key while this mining key is still active. However, only the first of such ballots will be successfully completed, while the rest will remain active.

Fixed at PR 145.

4. KeysManager.sol#L77

`require(poa.getCurrentValidatorsLength() <= maxLimitValidators());` — here it is necessary to change `<=` to `<`; otherwise it turns out that when the limit is reached, one more addition is allowed.

Fixed at PR 145.

5. ValidatorMetadata.sol#L260

The public modifier allows the voting key owner to use any mining key, including creating a request for changing other people's data or editing someone else's request before it is accepted.

Client: the `changeRequestForValidator` function has been removed with its code having been transferred to the changeRequest function at PR 146.

6. ValidatorMetadata.sol#L331

`uint256 public constant MAX_PENDING_CHANGE_CONFIRMATIONS = 50` — such a definition of a constant will block the ability to make changes if the threshold (returned from `ballotsStorage.getBallotThreshold (metadataChangeThresholdType))`through voting is set above 50.

Fixed at PR 146.

Client: here https://github.com/poanetwork/poa-network-consensus-contracts/pull/164/files#diff-2c00ed73fa555c3fa081a96e5b5893dc it was decided to return the value of 50 as a constraint to limit the iteration of the cycles here https://github.com/varasev/poa-network-consensus-contracts/blob/aa6eab9a28fbc710858b0e020e7f5f69dda8ba17/contracts/ValidatorMetadata.sol#L447 and here https://github.com/varasev/poa-network-consensus-contracts/blob/aa6eab9a28fbc710858b0e020e7f5f69dda8ba17/contracts/ValidatorMetadata.sol#L638. The added code takes into account the minimum threshold, so if the threshold is higher than 50, the lock will not occur during finalisation: https://github.com/varasev/poa-network-consensus-contracts/blob/aa6eab9a28fbc710858b0e020e7f5f69dda8ba17/contracts/ValidatorMetadata.sol#L275-L280

But in fact, the restriction that was supposed to be brought back can be bypassed https://github.com/varasev/poa-network-consensus-contracts/blob/aa6eab9a28fbc710858b0e020e7f5f69dda8ba17/contracts/ValidatorMetadata.sol#L277-L278

Client: yes, it can be bypassed if `minThreshold` is greater than `50`. But it is unlikely that `minThreshold` will be set to such a large value when initialising the `BallotsStorage` contract. I will additionally do a check inside the function `BallotsStorage.init` so that this value cannot be exceeded. Improvements according to this warning are introduced in https://github.com/poanetwork/poa-network-consensus-contracts/pull/166

7. KeysManager.sol#L323

Is it possible to add a validator that is greater by 1 than `maxLimitValidators`?

Client: yes, replaced with a strict inequality: PR 145.

By the way, this is another place where you can create an unclosable ballot.

Client: yes, right. It will be necessary to add a check for the number of validators when creating a ballot. It is unlikely that the system will reach 2,000 validators, but nonetheless.

No, in a simple form, a check during creation won't suffice; I wrote about this in the last comment. If it is done, then you have to keep a counter of the validators expected to be added and add them to the current number when checking the condition.

Client: We decided not to add such a check because the situation with a number of validators equal to 2000 is unlikely. In order not to complicate the code for processing such an unlikely event. The solution for the issue with unfinalisable ballots is described in remark No 9.

8. KeysManager.sol#L263

Exceeding `maxLimitValidators` is possible, if some of the initial keys are used after reaching `maxLimitValidators` via `addMiningKey`.

Fixed at PR 153.

9. VotingTo.sol#L191

In some parts of the code that read the mining keys logs, there are limits to reading the logs (e.g. also here KeysManager.sol#L181), which are different. However, there is no limitation for mining keys logging. This means that when the set limits are reached, reading the log will stop returning the correct data. The reaching of the limits can occur as a result of attacker actions, and also, with some extremely low

probability, during a regular operation. We recommend setting identical limits everywhere for reading and writing.

Client: if we just put a limit on the mining key exchange recording, the function `VotingToChangeKeys.checkIfMiningExisted`will behave the same as now — return false for the key that was created more than 25 key exchanges ago. Do I understand correctly that you propose to prohibit swapping mining key if the limit is reached? As for the `KeysManager.migrateMiningKey`function — it will now take the value of the same limit from the public-getter maxOldMiningKeysDeepCheck(): PR 156.

Yes, it turns out that the limit is 25 key changes. You would know better how this will interfere. There are alternatives (make a tree structure, Bloom filter, ...), but they are relatively cumbersome.

Client: I think we will stick with the current implementation and current limits.

10. PoaNetworkConsensus.sol#L78

We recommend adding a check whether the `currentValidators[i]` validator has not been added before. Otherwise, some cells in `currentValidators` and `pendingList` will be perpetually occupied, and `getCurrentValidatorsLength` and `getCurrentValidatorsLengthWithoutMoC` will return inflated numbers.

Fixed at PR 156.

11. Global limits for ballots

In the documentation global limits for ballots are mentioned (e.g. “Validator Management Ballot: 9 active ballots at one time”). In the audited version of the code they are not present, but if they are implemented, they will not only be useless against spam, but in a compartment with the problem of unclosable ballots they can lead to a global denial of service.

Client: Yes, this information in the documentation is outdated (we will correct it). Instead of global limits, we use limits for each validator, calculated in the `BallotsStorage.getBallotLimitPerValidator` function: BallotsStorage.sol#L135-L145. The solution for unfinalisable ballots is described in remark No 9. Please comment: Do you see any problems with spam protection in the current implementation, which relies on `BallotsStorage.getBallotLimitPerValidator`?

We see no problems here.

12. Documentation

In the documentation on the diagram there is a comment that the contract method looks for votes ready to be finalised and finalises them, but it is not present in the code. The next diagram contains the same comment https://github.com/poanetwork/wiki/wiki/POA-Network-Whitepaper#voting-on-a-ballot

Client: we will remove the superfluous note from the diagrams. The corresponding issue: https://github.com/poanetwork/wiki/issues/67.

13. BlockReward.sol#L71

The `isMiningActive` check is sufficient only if the miner has remained active for the entire period. In a situation where there have been no rewards for a long time, it may turn out that the miner is no longer active, but has not yet received a reward for previous periods.

Client: here it is assumed that the reward function will be consistently called every 5 seconds. At each call, the validator (miner) will be different (at the moment, there are 19 validators on the network) — this function is called by the Parity engine in a loop for all validators from the list. When the last validator is processed, the call starts from the beginning of the list. Therefore, for example, in case of 20 validators, the wait time for any validator is 100 seconds. If the validator is no longer present, but Parity for some reason called the reward function for this validator, then a revert will occur. Theoretically, such situations should not occur; Parity should operate with a list of validators that is always up-to-date. But require has been inserted here, just in case.

14. BlockReward.sol#L115

In the `rewardHBBFT` function, in the loop, there is a distribution of rewards between the miners, with the list of miners being rebuilt on the go. There may be a situation in which the rewards distribution begins with an outdated list, but continues with the updated one, and it is unclear how the updated list and the list of miners who really should receive a reward for their work should correspond (as at the moment of the function call, it is guaranteed that the list contains precisely the miners who completed the work within a certain period in the past).

One of the problems that arise: if the miner has lost control of the key and changed it through a ballot, it is possible that the reward will go to the old key because the list of `hbbftPayoutKeys` had not been updated for a long time.

Client: the work here is built on the following principle. It is assumed that all validators should receive compensation in turn. At the very beginning, the list of validators is taken. During the cycle, the reward is charged to each validator in the order of its number in the array. If the loop goes beyond the current size of the array

of validators, then after processing the last validator in the array, the list is updated because during the processing of the array from the zero element to the last, the lineup of validators could change. Thus, the processing begins with the first validator of the already updated list. I.e. it takes a snapshot of the validators list, and it does not change until this list is processed to the end. Once it is processed to the end, the snapshot is taken again. Etc. This contract has not been fully worked out yet — it is possible that the algorithm will change, but we still need to test the current version of its code. Let me know if the calculation algorithm given above is not completely clear to you. We have a small description of the task by which the function was implemented: https://github.com/poanetwork/RFC/issues/16

[REMARKS]

1. KeysManager.sol#L238-L265

Several `initialKey`s can use the same mining key. In the current implementation of the `PoaNetworkConsensus.sol`contract, the addValidator function will throw an exception in this situation, but it's better to add an explicit check via `successfulValidatorClone`, as this will reduce the chance of errors occurring when making changes. In particular, you will not have to think about what will happen if you have already created and initialised `KeysMaster`, but `ProxyStorage` is not initialised yet, and `poaNetworkConsensus()` returns `address(0)`. Or, what happens if the ballot decided to change `ProxyStorage`, and with it, the address `EthernalStorageProxy` of the contract for `PoaNetworkConsensus` changes, but the migration of data from `PoaNetworkConsensus` is a separate transaction, and the call `createKeys` has wedged between these events.

Any `initialKey` can use someone else's payment key and/or voting key.

Fixed at PR 135.

2. KeysManager.sol#L346-L368

In the `swapMiningKey` function, `miningKey` can be overwritten along with all other keys, create an infinite loop in the `miningKeyHistory` chain (actually kill the connection of the current key with all previous ones) if the attacker knows only the mining key — it is not checked if the new and old keys are different. In the current contract system, the possibility of such a call is eliminated, but it's better to insert an explicit check and not to rely on the fact that future changes in `VotingToChangeKeys` will not permit exploiting the vulnerability.

Fixed at PR 148.

3. ValidatorMetadata.sol#L317

Linear complexity in verifying the fact of voting. However, the length of the array is never greater than three (written in ValidatorMetadata.sol#L139), so there isn't a problem.

Client: functions related to changing the `ProxyStorage` address are removed from `ValidatorMetadata` in PR 150 — these functions were migrated from the old version of the contract. Now they are not needed because we have upgradable `ProxyStorage`.

4. ValidatorMetadata.sol#L367

Directly specifying an integer value (enum) will potentially cause problems when making changes.

Fixed at PR 149.

5. VotingToChangeKeys.sol#L93

Nowhere after the call to `areBallotParamsValid` it is checked that `mining key != address(0)`. Therefore, one can, for example, create a ballot to add a voting key for the zero address. If, for some reason, such a ballot ends with an accept, at the finalisation stage, a revert will occur when trying to add voting keys to `KeysManager`, and this member will have an unremovable Ballot freeze.

Fixed at PR 147.

6. EternalStorageProxy.sol#L59

The value of the free memory pointer is not updated. Before using a fragment of memory, it is recommended to allocate it by adding `mstore(0x40, a new free memory pointer)`.

Client: I think we can do without a free memory pointer, as described in https://github.com/zeppelinos/zos-lib/issues/70 — I have made a corresponding PR to delete it: https://github.com/poanetwork/poa-network-consensus-contracts/pull/151.

Commented on the issue. Long story short — it is a dangerous practice. However, if you do not use the "allocator" at all, the question is removed.

Client: Does it mean that in our case, such a solution may be used? EternalStorageProxy.sol#L58-L75.

Yes.

7. VotingToChangeKeys.sol#L219

During ballot migration, the following parameters are not migrated: `NEW_PAYOUT_KEY`, `NEW_VOTING_KEY`.

Client: yes, that's right, since the migration is assumed from the old version of the contract, in which there are no such parameters. For the future, I added the migration of these parameters in a commented form. If in the future a migration from a new version of the contract to a new one is needed, these lines will be uncommented: PR 152.

8. KeysManager.sol#L409

The attacker has the opportunity to force specification of another, already registered voting key on their mining key. Further serious development of this vector of attack is not visible; however, we recommend to preventing this situation by a pre-checking the corresponding `miningKeyByVoting` for 0.

Client: the check is added simultaneously to KeysManager and VotingToChangeKeys: PR 152.

But, it seems, there is again the risk of an unclosable ballot.

Client: yes, but we decided that unclosable ballots in such cases are normal. If, for example, we cannot add a voting key to a nonexistent mining key, then such a ballot should not be finalised and should have the sign "not finalised" to show that the proposed changes have not been applied. The very cases of such voting are theoretically possible by code, but are unlikely (now including due to additional checks even during the creation of voting). However, if you think that unfinalised ballots may pose a threat, please provide your comments.

Although if the ballot is not finalised, the function _decreaseValidatorLimit will not be called, which would entail a decrease in the ballot creation limit of a particular validator. I'll think of a better solution.

No global consequences from the unclosable ballots (e.g. DoS) are seen.

Client: the solution for this issue is complete in remark No 9.

9. VotingToChangeKeys.sol#L116

Such checks, referring to the global state, are valid only at the time of creating the ballot. When the voting effects are applied, the global state may change and the test

conditions will not be satisfied, but this will go unnoticed. One of the consequences is mentioned above.

Client: Yes, similar checks are added to the KeysManager (at the finalisation stage): PR 152.

Yes, now there are checks, which is good, but now a lot of potential places where you can create an unclosable ballot have popped up. Perhaps, instead of dealing with this set, it would be better to apply ballot effects by using sub-call and processing the Boolean completion code (e.g. in the case of `false`, just put the corresponding information flag in the vote)?

Client: so, you suggest to get rid of reverts in KeysManager, and to set the 'if'-conditions. If the condition is not met, then the ballot effects are not applied, but the ballot is still marked as finalised. Am I right?

Not completely. I suggest to keep the reverts, but here VotingToChangeKeys.sol#L297 and in all similar places, replace the external call with a code like `keysManager.addMiningKey.value (0) (affectedKey);`, i.e. ignore pop-up exceptions.

Now that I have outlined such a rebellious though, I started doubting myself. Maybe, as an alternative — we could get rid of the reverts in the code that applies the changes and ignore the changes, if they cannot be applied, i.e. just `return;`.

Client: if we consider the negative consequences of unfinalised ballots, then I see only one instance with the function `VotingToChange._decreaseValidatorLimit` (described above).

We can change the code for all the `KeysManager` functions that are called from `VotingToChangeKeys`, so that they do not initiate revert, but rather return false if it fails. Then the finalisation function will not do a revert (as done with `VotingToManageEmissionFunds`) and we can do without `prefinalise`.

It was decided to replace the reverts with returns, as you suggested above. Now when finalising the ballot, if the conditions at the time of finalisation are not met, the transaction will still pass, but the changes will not be applied. This has been fixed at PR 155. To fix the problem with the `_decreaseValidatorLimit` function in the `VotingToChange._finalizeBallot` function, the code is changed as follows: diff — i.e. the ballot limit change happens once (during the first finalisation transaction). If the finalisation attempts are repeated, the counter value does not change, so that it cannot be increased by cheaters.

10. ValidatorMetadata.sol#L279

One validator can nullify the ballot for changing the metadata of another validator.

Client: the `changeRequestForValidator` function has been removed with the transfer of its code to the `changeRequest`function in PR 146.

11. ValidatorMetadata.sol#L334

Since the voting key is taken into account during the ballot, rather than the mining key, and those in turn can change for the same validator, it is possible for one validator to vote several times.

Fixed at PR 157.

12. ValidatorMetadata.sol

Is the expected behaviour that when changing the mining key of the validator, it will lose its metainformation? Is the expected behaviour that when returning the existing mining key to the active validators set, the old metainformation will automatically be associated with this mining key? We recommend processing the deletion and change of the validator key in `ValidatorMetadata`.

Fixed at PR 158.

13. VotingToManageEmissionFunds.sol#L48

Since there can only be one ballot at a time about the emission, one validator can with some limited success organise a DoS attack on emission funds payout, constantly creating malicious ballots (e.g. on payout of emissions funds to itself).

Client: if the validator does so, other validators will be able to vote against and exclude it from the list of validators (in a separate ballot for the keys). For the validators to have more difficulty in colluding, there should be 50% + 1 validator voted, as done in PR 160. If the validator does this inadvertently (e.g. created a ballot with an incorrect payout address), it will have 15 minutes to cancel the erroneous ballot. The abolition of erroneous ballots is implemented in PR 168.

14. `abi.encodePacked`

Generating keys for contract storages through `abi.encodePacked` seems a dangerous practice because of the probability of collision: e.g. if keys for two different records are formed from such sets of fragments [‘foo‘, ‘bar‘], [‘foob‘, ‘ar‘], a collision will occur. It is recommended to use `abi.encode`.

Fixed at PR 159.

15. Enums

In contracts, the comparison of the elements of `enum` to more/less is used, and also the assumption that the first `enum`element declared in the source code will be converted to `0` and back. Unfortunately, the solidity documentation gives very little guarantee about the aspects of `enum`, and does not guarantee the functioning of the techniques described above. We recommend avoiding them.

Client: in practice for Solidity it is checked that the first element in enum is always 0, and each next element is 1 more than the previous one. This is accepted in most programming languages (except for languages that allow explicit assignment of values to enum elements), so we believe that this basic principle will not change in Solidity in the future. At least the current version 0.4.24 is all right in this aspect. If this plan somehow changes in the future, then our unit tests will show errors - we will see it and make corrections, if necessary.

16. EmissionFunds.sol#L49

Theoretically, an attacker can fill up the EVM stack in such a way that the call to the legitimate recipient will not work and will simply return false (for more details, see https://solidity.readthedocs.io/en/latest/security-considerations.html#callstack-depth), i.e. the attacker can disrupt the payment (at the same time, the funds remain safe). However, for this purpose, the attacker must be a validator, and his voting key must be the address of a contract.

Client: not completely clear about the voting key. Do I correctly understand that the attacker should call the function `VotingToManageEmissionFunds.finalize` with an overflowed stack? The `EmissionFunds.sendFundsTo` function can only be called from there.

Yes, you understand correctly, to call with an almost overflowed stack, so that the finalise call is processed, and the subsequent ones are not.

Client: maybe we need to add `require (msg.sender == tx.origin);` to the `VotingToManagerEmissionFunds._finalize`function?

I think it will be a little more reliable to require that `msg.sender` not be a contract.

Client: as I understand, for this purpose it is possible to use asm-function `extcodesize(msg.sender)`, although its call will cost 700 Gas. Do you happen to know a cheaper solution to determine if the address belongs to an arbitrary contract?

We do not know. 700 Gas is 30 times less than the fee for the Ether transfer. I think that it is acceptable.

Fixed at PR 165.

17. BlockReward.sol#L122

It is possible to achieve a block Gas limit due to writing to the storage during `_hbbftRefreshPayoutKeys()`. Directly in the code, strict restrictions on the growth of `keysNumberToReward` are not visible.

Client: here it is assumed that the reward function will be called by the Parity engine (or the engine based on it). The Parity documentation says that such functions are called on behalf of the special address `0xfffffffffffffffffffffffffffffffffffffffe` and the transactions are "system" ones: https://wiki.parity.io/Block-Reward-Contract.html#limitations

I ran the experiment using https://github.com/varasev/test-block-reward and looked at the value of `block.gaslimit` in the contract at the time the reward function is called. It equals `0xFFFF...FFFF` (256 bit). I.e. the biggest number that fits into `uint256` (infinity, essentially).

Therefore, we can assume that the Gas is unlimited for such "system" calls because it is not really spent by anyone.

Also, it is not decided yet where exactly we will use the `RewardByTime` contract (it is made for the future) — whether it will be called in a similar way, or some other way. In any case, I have added a TODO comment to the contract code so that in the future I do not forget about the potential Gas restriction in the block: PR 168.

18. BlockReward.sol#L151

Is it correct that the payout to the validator always occurs, regardless of the Boolean flag `keysManager.isPayoutActive(miningKey)`?

Client: Yes, if the validator does not have a payout key, then the payment should occur to its mining key (for example, such a case is with the Master of Ceremony).

19. BallotsStorage.sol#L20

The threshold type `MetadataChange` is not actually used during the consensus work (voting for changes in metadata are constructed differently).

Client: this type of threshold is used in the function `ValidatorMetadata.getMinThreshold`: ValidatorMetadata.sol#L366 — the threshold value used in the function `ValidatorMetadata.finalize`:ValidatorMetadata.sol#L342.

Yes, the question is removed.

[PROPOSALS]

1. KeysManager.sol#L109 and KeysManager.sol#L318

The function `initialKeys` and `getInitialKey` have the same signature and return the same result. In this case, the result of the function is not the key, as one might think from the name, but the status of the key. The setter for this value is called `_setInitialKeyStatus`. It is suggested to make one function with the name `getInitialKeyStatus` instead of two functions.

Fixed at PR 148.

2. VotingTo.sol#L186

The function `hasAlreadyVoted` deducts the mining key by the voting key via `KeysManager` and calls `hasMiningKeyAlreadyVoted`. A line earlier, we already deducted the mining key, and this call can be replaced with the `hasMiningKeyAlreadyVoted` call.

Fixed at PR 149.

3. VotingToChange.sol#L124-L126

It is better to be extra safe and use `SafeMath` for the increment/decrement of the voice counter.

Client: I think it's better to leave it as it is, because increment/decrement occurs over the signed `int256`. Progress in this case can have a negative value — this is normal.

4. BlockReward.sol#L88

The function `rewardHBBFT` should probably be described in the IBlockReward interface.

Client: The BlockReward smart contract in one of the PRs was divided into two parts: RewardByBlock u RewardByTime (each with its own interface): PR 140.

5. PoaNetworkConsensus.sol#L168

`delete pendingList [lastIndex]` for the `pendingList` address array assigns the value of `address(0)` to the last element of the array, and the next step will be to cut this element off, which means resetting it does not make sense.

Fixed at PR 149.

6. PoaNetworkConsensus.sol#L169

The `if (pendingList.length > 0)` check in the code of the `removeValidator` function looks meaningless because if `length == 0`, `revert` happens earlier in the code.

Fixed at PR 149.

7. VotingToChangeKeys.sol#L230

In cases other than `_affectedKeyType == uint256(KeyTypes.MiningKey)`, it is recommended to add a check `require(keysManager.isMiningActive(_miningKey));`, similar to the functions `_areKeySwapBallotParamsValid` and `_areKeyRemovalBallotParamsValid`.

Fixed at PR 152.

8. VotingToChange.sol#L73

It is proposed to distinguish between migration and the regular mode of work, so that they cannot be executed in parallel. In particular, we propose requiring the migration to be completed before the execution of any regular ballot code.

Fixed at PR 167.