

# NodeReal BAS

## Pull Request Audit Report

**Audit Period:** 2023/01/23 - 2023/02/09

**Overall Risk:** **Medium**

Based on this link:

Github	<a href="https://github.com/node-real/semita-netmarble-genesis-config/pull/3">https://github.com/node-real/semita-netmarble-genesis-config/pull/3</a>
--------	---

### Background:

Eric li, [Jan 17, 2023, at 3:58:46 PM]:

We are going to start auditing the new version of BAS

### 1. Requirements

A new economic system

- Block mint tokens are distributed proportionally to (Validator, Operation, ecosystem, and Partner Wallet).
- All transaction fees are sent to a foundation wallet
- At a certain height will not be in mint

Gas price

- Gas price is controlled by chain management

### 2. Implementation

#### 2.1 There will be two hard forks

The first hard fork will increase limitations on the size of the deployment contract from 24K to 48K just like the Shanghai Upgrade. The motivation for this feature is to enable us to add more functionality to the system contracts, like stake contracts.

The second hard fork will allow the new block reward distribution mechanism and the on-chain gas price governance module.

The operator needs to complete after contract48kBlock and before fncv2Block

Upgrade chainconfig and staking contracts

After upgrading the staking and chainconfig contracts, modify the allocation information of block mint, foundation wallet address, and gas price

#### 2.2 Code

##### 2.2.1 A new economic system

Parlia::distributeRewards distributes rewards by calling the contract (staking::distributeRewards) staking::distributeRewards configuration information from the chainconfig contract

### 2.2..2 Gas price

The gas price is controlled by the contract chainconfig, which retrieves the gas price when the transaction pool resets as the gas price of the transaction pool, thus preventing transactions below the gas price from entering the transaction pool

### 2.2.3 Genesis.json Add param example:

"contract48kBlock":10, deployment contract from 24K to 48K

"fncy2Block":3910, a new economic system

parlia."stopMintBlock": 4135, height will not be in mint`

We found out the Audit Scope as

No.	Contract Name	Type	Verdict	Details
1	Staking.sol	Staking	Medium	[M01] [M02] [L01] [L02] [I01] [I02] [I03] [I04] [I05] [I06] [I07]
2	ChainConfig.sol	Config	Informational	[I01] [I02]

# 1. Staking.sol

## Medium Severity

### [Fixed - M01] Use of Transfer in \_distrucuteRewards()

The function \_distributeRewards() uses .transfer() in 3 separate places to send funds to addresses. Due to the possibility of gas costs changing, it could result in transfers failing to smart contracts, potentially causing the function to revert and potentially soft-locking the smart contract, stopping the distribution of rewards.

```
724 + function _distributeRewards(address validatorAddress,uint256 blockRewards, uint256 gasFee) internal {
725 +     require(msg.value == blockRewards + gasFee, "bad rewards");
726 +     address foundationAddress = _CHAIN_CONFIG_CONTRACT.getFoundationAddress();
727 +     if (gasFee > 0) {
728 +         if (foundationAddress == address(0x00)){
729 +             blockRewards = blockRewards + gasFee;
730 +         } else {
731 +             payable(foundationAddress).transfer(gasFee);
732 +             emit ShareRewards(foundationAddress, gasFee);
733 +         }
734 +     }
735 +     if (blockRewards <= 0) {
736 +         return;
737 +     }
738 +     uint16 validatorRewardsShare;
739 +     IChainConfig.DistributeRewardsShare[] memory distributionRewardsShares;
740 +     (validatorRewardsShare, distributionRewardsShares) = _CHAIN_CONFIG_CONTRACT.getDistributeRewardsShares();
741 +     if (distributionRewardsShares.length == 0){
742 +         _depositValue(validatorAddress, blockRewards);
743 +         return;
744 +     }
745 +     uint256 validatorRewards = blockRewards*validatorRewardsShare / 10000;
746 +     uint256 totalPaid = 0;
747 +     for (uint256 i = 0; i < distributionRewardsShares.length; i++) {
748 +         IChainConfig.DistributeRewardsShare memory ds = distributionRewardsShares[i];
749 +         uint256 accountRewards = blockRewards * ds.share / 10000;
750 +         payable(ds.account).transfer(accountRewards);
751 +         emit ShareRewards(ds.account, accountRewards);
752 +         totalPaid += accountRewards;
753 +     }
754 +     uint256 dustRewards = blockRewards - validatorRewards - totalPaid;
755 +     if (dustRewards > 0) {
756 +         if (validatorRewardsShare > 0) {
757 +             validatorRewards = validatorRewards + dustRewards;
758 +         } else {
759 +             payable(distributionRewardsShares[0].account).transfer(dustRewards);
760 +         }
761 +     }
762 +     if (validatorRewards > 0) {
763 +         _depositValue(validatorAddress, validatorRewards);
764 +     }
765 + }
```

**Suggestion:**

Use the already existing `_safeTransferWithGasLimit()` function with a self-defined higher `TRANSFER_GAS_LIMIT` to conduct these value transfers.  
Or use the already existing `_unsafeTransfer()` function without gas limit to conduct these value transfers.

**Alleviation:**  
Fixed.

```
52      uint64 internal constant TRANSFER_GAS_LIMIT = 30_000;
```

```
724      function _distributeRewards(address validatorAddress,uint256 blockRewards, uint256 gasFee) internal {
725          require(msg.value == blockRewards + gasFee, "bad rewards");
726          address foundationAddress = _CHAIN_CONFIG_CONTRACT.getFoundationAddress();
727          if (gasFee > 0) {
728              if (foundationAddress == address(0x00)){
729                  blockRewards = blockRewards + gasFee;
730              } else {
731                  _safeTransferWithGasLimit(payable(foundationAddress), gasFee);
732                  emit ShareRewards(foundationAddress, gasFee);
733              }
734          }
735          if (blockRewards <= 0) {
736              return;
737          }
738          uint16 validatorRewardsShare;
739          IChainConfig.DistributeRewardsShare[] memory distributionRewardsShares;
740          (validatorRewardsShare, distributionRewardsShares) = _CHAIN_CONFIG_CONTRACT.getDistributeRewardsShares();
741          if (distributionRewardsShares.length == 0){
742              _depositValue(validatorAddress, blockRewards);
743              return;
744          }
745          uint256 validatorRewards = blockRewards*validatorRewardsShare / 10000;
746          uint256 totalPaid = 0;
747          for (uint256 i = 0; i < distributionRewardsShares.length; i++) {
748              IChainConfig.DistributeRewardsShare memory ds = distributionRewardsShares[i];
749              uint256 accountRewards = blockRewards * ds.share / 10000;
750              _safeTransferWithGasLimit(payable(ds.account), accountRewards);
751              emit ShareRewards(ds.account, accountRewards);
752              totalPaid += accountRewards;
753          }
754          uint256 dustRewards = blockRewards - validatorRewards - totalPaid;
755          if (dustRewards > 0) {
756              if (validatorRewardsShare > 0) {
757                  validatorRewards = validatorRewards + dustRewards;
758              } else {
759                  _safeTransferWithGasLimit(payable(distributionRewardsShares[0].account), dustRewards);
760              }
761          }
```

### [Fixed - M02] Business Logic - Strict Equality

For the line in the red box, there is a strict equality check where a validator is put in jail if his `slashesCount == felonyThreshold`.

```

function slash(address validatorAddress) external onlyFromSlashingIndicator virtual override {
    _slashValidator(validatorAddress);
}

function _slashValidator(address validatorAddress) internal {
    // make sure validator exists
    Validator memory validator = _validatorsMap[validatorAddress];
    require(validator.status != ValidatorStatus.NotFound, "not found");
    uint64 epoch = currentEpoch();
    // increase slashes for current epoch
    ValidatorSnapshot storage currentSnapshot = _touchValidatorSnapshot(validator, epoch);
    uint32 slashesCount = currentSnapshot.slashesCount + 1;
    currentSnapshot.slashesCount = slashesCount;
    // if validator has a lot of misses then put it in jail for 1 week (if epoch is 1 day)
    if (slashesCount == _CHAIN_CONFIG_CONTRACT.getFelonyThreshold()) {
        validator.jailedBefore = currentEpoch() + _CHAIN_CONFIG_CONTRACT.getValidatorJailEpochLength();
        validator.status = ValidatorStatus.Jail;
        _removeValidatorFromActiveList(validatorAddress);
        _validatorsMap[validatorAddress] = validator;
        emit ValidatorJailed(validatorAddress, epoch);
    } else {
        // validator state might change, lets update it
        _validatorsMap[validatorAddress] = validator;
    }
    // emit event
    emit ValidatorSlashed(validatorAddress, slashesCount, epoch);
}

```

However, consider this situation where the *felonyThreshold* is 2 and this particular validator has 1 *slashesCount*. Now, if the *felonyThreshold* were to be updated to 1 in the ChainConfig.sol contract. If the *\_slashValidator* function is called now, since the *slashesCount* is incremented first, then the check will now be bypassed.

This is because his *slashesCount* is now 2 and the *felonyThreshold* is now 1.

What if the *slashesCount* > *\_CHAIN\_CONFIG\_CONTRACT.getFelonyThreshold()*? What will happen? It seems the *validator.status* will not be changed to the Jail state again. Will the Governance role call the *removeValidator()* method to remove that validator manually?

**Suggestion:** It is recommended to review the business logic to prevent such a situation from happening.

**Alleviation:**

Fixed.

```

865 function _slashValidator(address validatorAddress) internal {
866     // make sure validator exists
867     Validator memory validator = _validatorsMap[validatorAddress];
868     require(validator.status != ValidatorStatus.NotFound, "not found");
869     uint64 epoch = currentEpoch();
870     // increase slashes for current epoch
871     ValidatorSnapshot storage currentSnapshot = _touchValidatorSnapshot(validator, epoch);
872     uint32 slashesCount = currentSnapshot.slashesCount + 1;
873     currentSnapshot.slashesCount = slashesCount;
874     // if validator has a lot of misses then put it in jail for 1 week (if epoch is 1 day)
875     if (slashesCount >= _CHAIN_CONFIG_CONTRACT.getFelonyThreshold() && validator.status != ValidatorStatus.Jail) {
876         validator.jailedBefore = currentEpoch() + _CHAIN_CONFIG_CONTRACT.getValidatorJailEpochLength();
877         validator.status = ValidatorStatus.Jail;
878         _removeValidatorFromActiveList(validatorAddress);
879         _validatorsMap[validatorAddress] = validator;
880         emit ValidatorJailed(validatorAddress, epoch);
881     } else {
882         // validator state might change, lets update it
883         _validatorsMap[validatorAddress] = validator;
884     }
885     // emit event
886     emit ValidatorSlashed(validatorAddress, slashesCount, epoch);
887 }

```

## Low Severity

### [Acknowledged - L01] \_calcDelegatorRewardsAndPendingUndelegates() may exceed the block gas limit

The identified loop logic in `_calcDelegatorRewardsAndPendingUndelegates()` method may exceed the block gas limit when parameter `beforeEpoch` is far bigger than the epoch of last processed delegation.

```

function _calcDelegatorRewardsAndPendingUndelegates(address validator, address delegator, uint64 beforeEpoch, bool withUndelegate) internal view returns (uint256) {
    ValidatorDelegation memory delegation = _validatorDelegations[validator][delegator];
    uint256 availableFunds = 0;
    // process delegate queue to calculate staking rewards
    while (delegation.delegateGap < delegation.delegateQueue.length) {
        DelegationOpDelegate memory delegateOp = delegation.delegateQueue[delegation.delegateGap];
        if (delegateOp.epoch >= beforeEpoch) {
            break;
        }
        uint256 voteChangedAtEpoch = 0;
        if (delegation.delegateGap < delegation.delegateQueue.length - 1) {
            voteChangedAtEpoch = delegation.delegateQueue[delegation.delegateGap + 1].epoch;
        }
        for (; delegateOp.epoch < beforeEpoch && (voteChangedAtEpoch == 0 || delegateOp.epoch < voteChangedAtEpoch); delegateOp.epoch++) {
            ValidatorSnapshot memory validatorSnapshot = _validatorSnapshots[validator][delegateOp.epoch];
            if (validatorSnapshot.totalDelegated == 0) {
                continue;
            }
            (uint256 delegatorFee, /*uint256 ownerFee*/, /*uint256 systemFee*/) = _calcValidatorSnapshotEpochPayout(validatorSnapshot);
            availableFunds += delegatorFee * delegateOp.amount / validatorSnapshot.totalDelegated;
        }
        ++delegation.delegateGap;
    }
    // process all items from undelegate queue
    while (withUndelegate && delegation.undelegateGap < delegation.undelegateQueue.length) {
        DelegationOpUndelegate memory undelegateOp = delegation.undelegateQueue[delegation.undelegateGap];
        if (undelegateOp.epoch > beforeEpoch) {
            break;
        }
        availableFunds += uint256(undelegateOp.amount) * BALANCE_COMPACT_PRECISION;
        ++delegation.undelegateGap;
    }
    // return available for claim funds
    return availableFunds;
}

```

It can happen if functions `redelegateDelegatorFee()` / `claimDelegatorFee()` has not

been called by a delegator for a long time. Since there is no more `claimDelegatorFeeAtEpoch()` method, it cannot process delegation little by little so the possibility increases. It will lead to the denial of service.

### Recommendation

We recommend handling the mentioned situation properly such as adding the same gas check as in the function `_processDelegateQueue()`.

### Alleviation:

Acknowledged. This version will not modify.

## [Deny - L02] Validators with pending validator rewards should not be removable

If a validator with pending validator rewards is removed, the functions `claimValidatorFee()` will malfunction and users' funds can not be withdrawn.

Currently, in the `removeValidator()` method, there is no check if there are pending rewards in the validator to be removed.

```
function removeValidator(address account) external onlyFromGovernance virtual override {
    Validator memory validator = _validatorsMap[account];
    require(validator.status != ValidatorStatus.NotFound, "not found");
    // don't allow to remove validator w/ active delegations
    ValidatorSnapshot memory snapshot = _validatorSnapshots[validator.validatorAddress][validator.changedAt];
    require(snapshot.totalDelegated == 0, "has delegations");
    // remove validator from active list if exists
    _removeValidatorFromActiveList(account);
    // remove from validators map
    delete _validatorOwners[validator.ownerAddress];
    delete _validatorsMap[account];
    // emit event about it
    emit ValidatorRemoved(account);
}
```

### Recommendation:

We recommend not removing a validator when it has pending validator rewards.

Add judgment `require(snapshot.totalRewards == 0, "has pending rewards")`;

### Alleviation:

Removal in the pending state should not be rewarded, avoid frequent obtain rewards.

## Informational Severity

### [Fixed - I01] Duplicate require check

`_depositFee()` checks if `msg.value > 0`, however `_depositValue()` also has this same check integrated, meaning we can save gas by removing the check in the `_depositFee()` method.

```
function _depositFee(address validatorAddress) internal {
    require(msg.value > 0);
    _depositValue(validatorAddress, msg.value);
}
```

```
function _depositValue(address validatorAddress, uint256 value) internal {
    require(value > 0 && value <= msg.value, "bad value");
    // make sure validator is active
    Validator memory validator = _validatorsMap[validatorAddress];
    require(validator.status != ValidatorStatus.NotFound, "not found");
    uint64 epoch = currentEpoch();
}
```

### Suggestion:

Remove `require(msg.value > 0);` from `_depositFee()`

### Alleviation:

Fixed.

```
780     function _depositFee(address validatorAddress) internal {
781         _depositValue(validatorAddress, msg.value);
782     }
```

### [Acknowledged - I02] Missing validator exists check

In `_delegateTo()` we get the validator from the `_validatorsMap[]` and ensure it exists by checking if its status is not equal to `"NotFound"`.

```
function _delegateTo(address fromDelegator, address toValidator, uint256 amount) internal {
    // check is minimum delegate amount
    require(amount >= _CHAIN_CONFIG_CONTRACT.getMinStakingAmount() && amount != 0, "too low");
    require(amount % BALANCE_COMPACT_PRECISION == 0, "no remainder");
    // make sure amount is greater than min staking amount
    // make sure validator exists at least
    Validator memory validator = _validatorsMap[toValidator];
    require(validator.status != ValidatorStatus.NotFound, "not found");
    uint64 atEpoch = nextEpoch();
}
```



In `_undelegateFrom()` we also get the validator from `_validatorsMap[]`, however, the check to ensure that the validator exists is omitted.

```
function _undelegateFrom(address toDelegator, address fromValidator, uint256 amount) internal {
    // check minimum delegate amount
    require(amount >= _CHAIN_CONFIG_CONTRACT.getMinStakingAmount() && amount != 0, "too low");
    require(amount % BALANCE_COMPACT_PRECISION == 0, "no remainder");
    // make sure validator exists at least
    Validator memory validator = _validatorsMap[fromValidator];
    uint64 beforeEpoch = nextEpoch();
    // Lets upgrade next snapshot parameters:
    // + find snapshot for the next epoch after current block
    // + increase total delegated amount in the next epoch for this validator
    // + re-save validator because last affected epoch might change
    ValidatorSnapshot storage validatorSnapshot = _touchValidatorSnapshot(validator, beforeEpoch);
}
```

### Suggestion:

Add

```
require(validator.status != ValidatorStatus.NotFound, "not found");
```

A clear error message for this error might be helpful to users.

### Alleviation:

Acknowledged and consider modifying in the next version.

### [Acknowledged - I03] Missing address 0 checks

When adding a new validator, there is no check to ensure that the `validatorAddress` is not equal to `address(0)`.

```

547 function registerValidator(address validatorAddress, uint16 commissionRate) payable external override {
548     uint256 initialStake = msg.value;
549     // // initial stake amount should be greater than minimum validator staking amount
550     require(initialStake >= _CHAIN_CONFIG_CONTRACT.getMinValidatorStakeAmount(), "too low");
551     require(initialStake % BALANCE_COMPACT_PRECISION == 0, "no remainder");
552     // add new validator as pending
553     _addValidator(validatorAddress, msg.sender, ValidatorStatus.Pending, commissionRate, initialStake, nextEpoch());
554 }
555
556 // // fix(seven): temp hack for code size too large
557 // function addValidator(address account) external onlyFromGovernance virtual override {
558 //     _addValidator(account, account, ValidatorStatus.Active, 0, 0, nextEpoch());
559 // }
560
561 function _addValidator(address validatorAddress, address validatorOwner, ValidatorStatus status, uint16 commissionRate, uint256 initialStake, uint64 sinceEpoch) {
562     // validator commission rate
563     require(commissionRate >= COMMISSION_RATE_MIN_VALUE && commissionRate <= COMMISSION_RATE_MAX_VALUE, "bad commission");
564     // init validator default params
565     Validator memory validator = _validatorsMap[validatorAddress];
566     require(_validatorsMap[validatorAddress].status == ValidatorStatus.NotFound, "already exist");
567     validator.validatorAddress = validatorAddress;
568     validator.ownerAddress = validatorOwner;
569     validator.status = status;
570     validator.changedAt = sinceEpoch;
571     _validatorsMap[validatorAddress] = validator;
572     // save validator owner
573     require(_validatorOwners[validatorOwner] == address(0x00), "owner in use");
574     _validatorOwners[validatorOwner] = validatorAddress;
575     // add new validator to array
576     if (status == ValidatorStatus.Active) {
577         _activeValidatorsList.push(validatorAddress);
578     }
579     // push initial validator snapshot at zero epoch with default params
580     _validatorSnapshots[validatorAddress][sinceEpoch] = ValidatorSnapshot(0, uint112(initialStake / BALANCE_COMPACT_PRECISION), 0, commissionRate);

```

### Suggestion:

It is recommended to include check:

```
require(validatorAddress != address(0), '0 address');
```

### Alleviation:

Acknowledged and consider modifying in the next version.

## [Deny - I04] Unnecessary Condition

The internal function `_redelegateDelegatorRewards()` is only called once in this contract by the `redelegateDelegatorFee()` function. The last 2 parameters being passed through are `true` and `false`.

```

function redelegateDelegatorFee(address validator) external override {
    // claim rewards in the redelegate mode (check function code for more info)
    _redelegateDelegatorRewards(validator, msg.sender, currentEpoch(), true, false;
}


```

The last boolean parameter input is hardcoded as false, therefore the condition for `withUndelegates` will always be false. Hence the operation in the if the scope will never be executed. As such, this condition is redundant and can be removed.

```

function _redelegateDelegatorRewards(address validator, address delegator, uint64 beforeEpochExclude, bool withRewards, bool withUndelegates) internal {
    ValidatorDelegation storage delegation = _validatorDelegations[validator][delegator];
    // claim rewards and undelegates
    uint256 availableFunds = 0;
    if (withRewards) {
        availableFunds += _processDelegateQueue(validator, delegation, beforeEpochExclude);
    }
    if (withUndelegates) {
        availableFunds += _processUndelegateQueue(delegation, beforeEpochExclude);
    }
    (uint256 amountToStake, uint256 rewardsDust) = _calcAvailableForRedelegateAmount(availableFunds);
    // if we have something to re-stake then delegate it to the validator
    if (amountToStake > 0) {
        _delegateTo(delegator, validator, amountToStake);
    }
    // if we have dust from staking then send it to user (we can't keep them in the contract)
    if (rewardsDust > 0) {
        _safeTransferWithGasLimit(payable(delegator), rewardsDust);
    }
    // emit event
    emit Redelegated(validator, delegator, amountToStake, rewardsDust, beforeEpochExclude);
}

```



## Alleviation:

Reserve for future releases.

## [Fixed - I05] Wrong error message

The error message within the red box of the `_undelegateFrom()` internal function is not reasonable because it is not consistent with the require check.

```

function undelegateFrom(address toDelegator, address fromValidator, uint256 amount) internal {
    // check minimum delegate amount
    require(amount >= _CHAIN_CONFIG_CONTRACT.getMinStakingAmount() && amount != 0, "too low");
    require(amount % BALANCE_COMPACT_PRECISION == 0, "no remainder");
    // make sure validator exists at least
    Validator memory validator = _validatorsMap[fromValidator];
    uint64 beforeEpoch = nextEpoch();
    // Lets upgrade next snapshot parameters:
    // + find snapshot for the next epoch after current block
    // + increase total delegated amount in the next epoch for this validator
    // + re-save validator because last affected epoch might change
    ValidatorSnapshot storage validatorSnapshot = _touchValidatorSnapshot(validator, beforeEpoch);
    require(validatorSnapshot.totalDelegated >= uint112(amount / BALANCE_COMPACT_PRECISION), "insufficient balance");
    validatorSnapshot.totalDelegated -= uint112(amount / BALANCE_COMPACT_PRECISION);
    _validatorsMap[fromValidator] = validator;
    // if last pending delegate has the same next epoch then its safe to just increase total
    // staked amount because it can't affect current validator set, but otherwise we must create
    // new record in delegation queue with the last epoch (delegations are ordered by epoch)
    ValidatorDelegation storage delegation = _validatorDelegations[fromValidator][toDelegator];
    require(delegation.delegateQueue.length > 0, "insufficient balance");
    DelegationOpDelegate storage recentDelegateOp = delegation.delegateQueue[delegation.delegateQueue.length - 1];
    require(recentDelegateOp.amount >= uint64(amount / BALANCE_COMPACT_PRECISION), "insufficient balance");
    uint112 nextDelegatedAmount = recentDelegateOp.amount - uint112(amount / BALANCE_COMPACT_PRECISION);
    if (recentDelegateOp.epoch >= beforeEpoch) {
        // decrease total delegated amount for the next epoch
        recentDelegateOp.amount = nextDelegatedAmount;
    } else {

```

## Suggestion:

Use "no items in the delegateQueue" as the error message.

### Alleviation:

Fixed.

```
346 function UndelegateFrom(address toDelegator, address fromValidator, uint256 amount) internal {
347     // check minimum delegate amount
348     require(amount >= _CHAIN_CONFIG_CONTRACT.getMinStakingAmount() && amount != 0, "too low");
349     require(amount % BALANCE_COMPACT_PRECISION == 0, "no remainder");
350     // make sure validator exists at least
351     Validator memory validator = _validatorsMap[fromValidator];
352     uint64 beforeEpoch = nextEpoch();
353     // Lets upgrade next snapshot parameters:
354     // + find snapshot for the next epoch after current block
355     // + increase total delegated amount in the next epoch for this validator
356     // + re-save validator because last affected epoch might change
357     ValidatorSnapshot storage validatorSnapshot = _touchValidatorSnapshot(validator, beforeEpoch);
358     require(validatorSnapshot.totalDelegated >= uint112(amount / BALANCE_COMPACT_PRECISION), "insufficient balance");
359     validatorSnapshot.totalDelegated -= uint112(amount / BALANCE_COMPACT_PRECISION);
360     _validatorsMap[fromValidator] = validator;
361     // if last pending delegate has the same next epoch then its safe to just increase total
362     // staked amount because it can't affect current validator set, but otherwise we must create
363     // new record in delegation queue with the last epoch (delegations are ordered by epoch)
364     ValidatorDelegation storage delegation = _validatorDelegations[fromValidator][toDelegator];
365     require(delegation.delegateQueue.length > 0, "no items in the delegateQueue");
```

### [Acknowledged - I06] Ambiguous function name

For the *getValidators()* function, it will only return validators in the Active status. The validators in Pending or Jail status will not be included.

```
function getValidators() public view override returns (address[] memory) {
    uint256 n = _activeValidatorsList.length;
    address[] memory orderedValidators = new address[](n);
    for (uint256 i = 0; i < n; i++) {
        orderedValidators[i] = _activeValidatorsList[i];
    }
    // we need to select k top validators out of n
    uint256 k = _CHAIN_CONFIG_CONTRACT.getActiveValidatorsLength();
    if (k > n) {
        k = n;
    }
    for (uint256 i = 0; i < k; i++) {
        uint256 nextValidator = i;
        Validator memory currentMax = _validatorsMap[orderedValidators[nextValidator]];
        ValidatorSnapshot memory maxSnapshot = _validatorSnapshots[currentMax.validatorAddress][currentMax.changedAt];
        for (uint256 j = i + 1; j < n; j++) {
            Validator memory current = _validatorsMap[orderedValidators[j]];
            ValidatorSnapshot memory currentSnapshot = _validatorSnapshots[current.validatorAddress][current.changedAt];
            if (maxSnapshot.totalDelegated < currentSnapshot.totalDelegated) {
                nextValidator = j;
                currentMax = current;
                maxSnapshot = currentSnapshot;
            }
        }
    }
}
```

**Suggestion:**

Use `getActiveValidators()` to avoid ambiguity.

**Alleviation:**

This function is called in the Golang code. If the modification cost is too large, consider modifying it in a future version.

**[Acknowledged - I07] Misleading comments**

```
function depositValue(address validatorAddress, uint256 value) internal {
    require(value > 0 && value <= msg.value, "bad value");
    // make sure validator is active
    Validator memory validator = _validatorsMap[validatorAddress];
    require(validator.status != ValidatorStatus.NotFound, "not found");
    uint64 epoch = currentEpoch();
    // increase total pending rewards for validator for current epoch
    ValidatorSnapshot storage currentSnapshot = _touchValidatorSnapshot(validator, epoch);
    currentSnapshot.totalRewards += uint96(msg.value);
    currentSnapshot.totalRewards += uint96(value);
    // emit event
    emit ValidatorDeposited(validatorAddress, msg.value, epoch);
    emit ValidatorDeposited(validatorAddress, value, epoch);
}
```

**Suggestion:**

The comments should be changed to `"make sure validator exists at least"`.

**Alleviation:**

The validator here must be active, and this comment is considered to be optimized in future versions.

## 2. ChainConfig.sol

### Informational Severity

**[Acknowledged - I01] Missing same value check**

The `setEnabledDelegate()` function is missing a require statement checking that `newValue` is not equal to the previous value. All other functions that change the config values have this check, therefore it would be advisable to add this check to maintain consistency and save gas costs from conducting useless variable updates.

```
function setEnableDelegate(bool newValue) external override onlyFromGovernance {
    bool prevValue = enableDelegate;
    enableDelegate = newValue;
    emit EnableDelegateChanged(prevValue, newValue);
}
```

#### Suggestion:

Add

```
require(prevValue != newValue, "same value");
```

#### Alleviation:

Acknowledged and consider modifying in the next version.

### [Acknowledged - I02] Centralization issues

In the contract ChainConfig the privileged role freeGasAddressAdmin has authority over the addFreeGasAddress() and the removeFreeGasAddress() functions. Any compromise to the freeGasAddressAdmin account may allow the hacker to take advantage of this authority and add/remove gas-free addresses at will.

```
function addFreeGasAddress(address freeGasAddress) external onlyFromFreeGasAddressAdmin virtual override {
    _addFreeGasAddress(freeGasAddress);
}
```

```
function removeFreeGasAddress(address freeGasAddress) external onlyFromFreeGasAddressAdmin virtual override {
    _removeFreeGasAddress(freeGasAddress);
}
```

Also, these gas-free addresses can send transactions without any cost. So if they become malicious or their private keys are leaked, the hackers can shut down the blockchain network by flooding the network with numerous gas-free transactions.

#### Recommendation

In general, we strongly recommend these centralized roles be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multi-signature wallets or timelock contracts.

#### Alleviation:

Acknowledged by the team.