BEOSIN
Blockchain Security

Ronin Network

Public Block Chain Security Audit

V1.5

No. 202303031441

Mar 3rd, 2023

Contents

Overview

Project Overview

Project Name	Ronin Network
Audit Scope	https://github.com/axieinfinity/ronin (Ronin Node) https://github.com/axieinfinity/ronin-dpos-contracts (Ronin DPoS Contracts)
Ronin Node Commit Hash	66f20552589269278c415ab59fe18a7b874012e6 (Initial) 57a52b0010f86be1b4bd341fbfafd02be3023a96 0c98d1fe8f4f8ffb877bc70df100ce63b4504c46 de588caf79b03c68f062fe1014702f5b6688c076 a285a1a887307d6962b400c1069b50931ccd90c2 8e1799ea45be72be55486d41a3ab1179ebc06c8b 9cba5d7cdc60a6950311abf2791607cbff761e52 9bf4895fbd51a9964a9b09c03b38abf0ace53008 (Final)
Ronin DPoS Contracts Commit Hash	1d3f5e3c1de471edd6e8b4ea15167130f40e3d90 (Initial) 66903dbcdfb64964abe16994b4b2e7d5d9057ded (Final)

Audit Overview

Audit work duration: February 1, 2023 – March 3, 2023

Update Details: Project commit hash and findings status updated on March 13, 2023

Update Details: Project commit hash and **Historical Vulnerability Detection** chapter updated on March 30, 2023

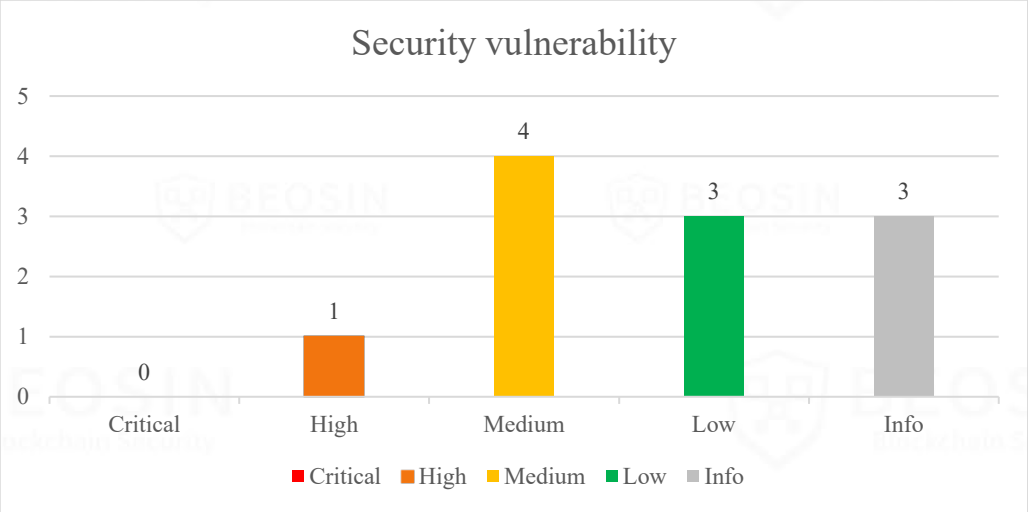
Update Details: **Historical Vulnerability Detection** and **Summary of audit results** chapters updated on May 10, 2023

Audit methods: Formal Verification, Static Analysis, Typical Case Testing and Manual Review.

Audit team: Beosin Security Team

Summary of audit results

After auditing, 1 High risk, 4 Medium-risk, 3 Low-risk and 3 Info items were identified in the Ronin Network project. Specific audit details will be presented in the **Findings for Blockchain and Findings for Consensus Contracts** sections. Users should pay attention to the following aspects when interacting with this project:



Notes:

- **Risk Description:**

1. Ronin is a fork of Go-Ethereum. Compared with Go-Ethereum, the project party has fixed multiple vulnerabilities such as CVE-2022-29177 in Ronin Node Ver.2.5.2. And in May 2023, all Ronin validators have successfully run Ver.2.5.2 Ronin Node in DPoS mainnet. It is recommended that users use Ronin node Ver.2.5.2 or later.

- **Notes for Project Party:**

1. Ronin is a fork of Go-Ethereum. Through this audit, it was found that there are multiple vulnerabilities that are the same as those in the historical version of Go-Ethereum. Therefore, it is recommended that the project party should pay attention to the following two security management aspects of the Ronin Node project:

   1) Pay attention to whether the vulnerabilities that have been exposed by Ethereum exist in the Ronin network, so as to prevent these vulnerabilities from being exploited in the Ronin network.

   2) Establish a periodic update mechanism to upgrade the third-party components/libraries that the Ronin project depends on in a timely, so as to avoid using risky third-party dependent components/libraries that endanger the safe operation of the Ronin node.

Findings for Blockchain

Index	Risk description	Severity level	Status
Ronin Node-1	consortium V2 checks the snapshot improperly	Medium	Fixed
Ronin Node-2	The *consortiumVerifyHeaders.Run* method lacks data association validation	Medium	Partially fixed.
Ronin Node-3	Changing the coinbase by the validator node will result in incorrect block generation	Info	Acknowledged

Status Notes:

- Ronin Node-2 is partially fixed by Ronin project party. The *Run* method has fixed the issue that the two-header data are not completely consistent, but it cannot be determined whether the two blocks exist on the chain. This interface will affect the *slashDoubleSign* function of the consensus contract, so the project party will add admin permission to *slashDoubleSign* function to prevent malicious data input (See **[DPoS Smart Contracts-6]** in **Findings For Consensus Contracts** chapter for permission addition details). In addition, in order to prevent double-signature attacks, the project party added the judgment of double-signature blocks to the *resultLoop* method of miner to prevent double-signature attacks.
- Ronin Node-3 is acknowledged by Ronin project party. During the operation of the consortium V1 consensus, it is recommended that the verification nodes do not use the *miner.setEtherbase* API to change the coinbase, otherwise they will face the risk of not being able to generate blocks, but **this risk does not exist when using consortium V2**.

Finding Details:

[Ronin Node-1] consortium V2 checks the snapshot improperly

Severity Level	Medium
Lines	consensus/consortium/v2/consortium.go #L310-382
Description	When the block reaches the height of the consortium V2, due to the error handling and verification of the snapshot, if the node restarts, it will not be able to continue to produce blocks.

```go
func (c *Consortium) snapshot(chain consensus.ChainHeaderReader, number uint64, hash common.Hash, parents []*types.Header) (*Snaps
    if err := c.initContract(); err != nil {
        return nil, err
    }
    // Search for a snapshot in memory or on disk for checkpoints
    var (
        headers    []*types.Header
        snap       *Snapshot
        cpyParents = make([]*types.Header, len(parents))
    )
    // NOTE(linh): We must copy parents before going to the loop because parents are modified.
    //  If not, the FindAncientHeader function can not find its block ancestor
    copy(cpyParents, parents)

    for snap == nil {
        // If an in-memory snapshot was found, use that
        if s, ok := c.recents.Get(hash); ok {
            snap = s.(*Snapshot)
            break
        }

        // init snapshot if it is at forkedBlock
        if number == c.forkedBlock-1 {
            var (
                err        error
                validators []common.Address
            )
            // get validators set from number
            validators, err = c.contract.GetValidators(big.NewInt(0).SetUint64(number))
            if err != nil {
                log.Error("Load validators at the beginning failed", "err", err)
                return nil, err
            }
            snap = newSnapshot(c.chainConfig, c.config, c.signatures, number, hash, validators, c.ethAPI)

            // load v1 recent list to prevent recent producing-block-validators produce block again
            snapV1 := c.v1.GetSnapshot(chain, number, parents)
```

Figure 1 Source code of *snapshot* method (Unfixed)

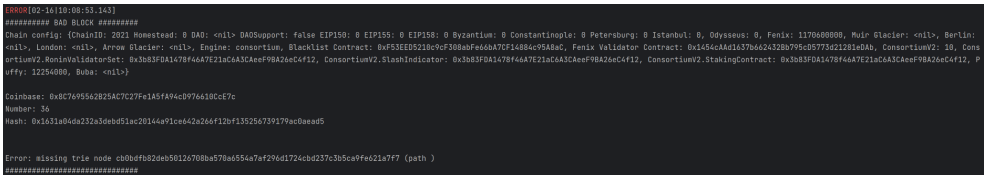


Figure 2 bad block

Recommendations	It is recommended to modify the snapshot verification logic in consortium v2.
Status	Fixed. See the PR for the fix details: https://github.com/axieinfinity/ronin/pull/212

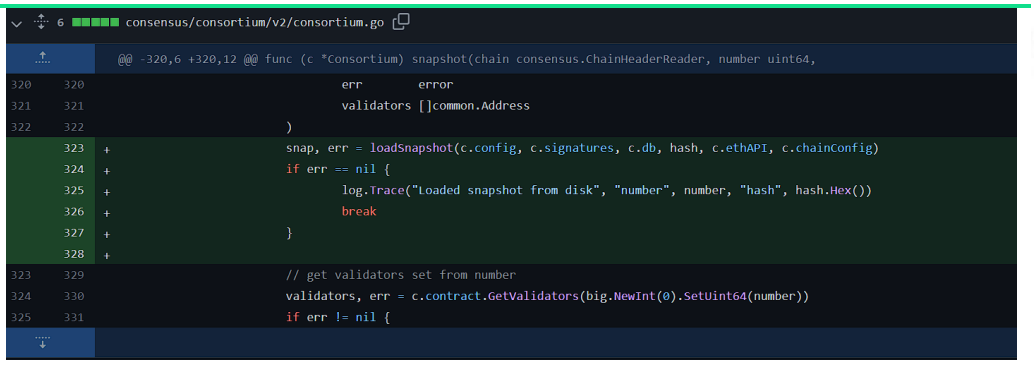


Figure 3 Source code of *snapshot* method (Fixed)

[Ronin Node-2] The *consortiumVerifyHeaders.Run* method lacks data association validation

Severity Level	Medium
Lines	core/vm/consortium_precompiled_contracts.go #L421-449 core/vm/consortium_precompiled_contracts.go #L478-493
Description	The *Run* method function does not check whether the data of a block passed in already exists in the chain database, nor does it check whether the data of the two blocks passed in are the same. Therefore, malicious data can be constructed to bypass *Run* method check.

```go
func (c *consortiumVerifyHeaders) Run(input []byte) ([]byte, error) {
    if c.evm.ChainConfig().ConsortiumV2Contracts == nil {
        return nil, errors.New("cannot find consortium v2 contracts")
    }
    if !c.evm.ChainConfig().ConsortiumV2Contracts.IsSystemContract(c.caller.Address()) {
        return nil, errors.New("unauthorized sender")
    }
    // get method, args from abi
    smcAbi, method, args, err := loadMethodAndArgs(consortiumVerifyHeadersAbi, input)
    if err != nil {
        return nil, err
    }
    if method.Name != verifyHeaders {
        return nil, errors.New("invalid method")
    }
    if len(args) != 2 {
        return nil, errors.New(fmt.Sprintf("invalid arguments, expected 2 got %d", len(args)))
    }
    // decode header1, header2
    var blockHeader1, blockHeader2 BlockHeader
    if err := c.unpack(smcAbi, &blockHeader1, args[0].([]byte)); err != nil {
        return nil, err
    }
    if err := c.unpack(smcAbi, &blockHeader2, args[1].([]byte)); err != nil {
        return nil, err
    }
    output := c.verify(blockHeader1, blockHeader2)
    return smcAbi.Methods[verifyHeaders].Outputs.Pack(output)
}
```

Figure 4 Source code of *Run* method

```go
func (c *consortiumVerifyHeaders) verify(header1, header2 BlockHeader) bool {
    if header1.toHeader().ParentHash.Hex() != header2.toHeader().ParentHash.Hex() {
        return false
    }
    signer1, err := c.getSigner(header1)
    if err != nil {
        log.Trace("[consortiumVerifyHeaders][verify] error while getting signer from header1", "err", err)
        return false
    }
    signer2, err := c.getSigner(header2)
    if err != nil {
        log.Trace("[consortiumVerifyHeaders][verify] error while getting signer from header2", "err", err)
        return false
    }
    return signer1.Hex() == signer2.Hex() && signer2.Hex() == header2.Benificiary.Hex()
}
```

Figure 5 Source code of *verify* method (Unfixed)

Recommendations	It is recommended to check whether the blocks data are the same and modify the logic of whether the block exists on the chain.

Status

Partially fixed. The Run method has fixed the issue that the two-header data are not completely consistent, but it cannot be determined whether the two blocks exist on the chain. This interface will affect the *slashDoubleSign* function of the consensus contract, so the project party will add the Admin permission to *slashDoubleSign* function to prevent malicious data input (See **[DPoS Smart Contracts-6] in *Findings For Consensus Contracts* chapter** for permission addition details). In addition, in order to prevent double-signature attacks, the project party added the judgment of double-signature blocks to the *resultLoop* method of miner to prevent double-signature attacks.

See the PRs for the fix details:

- https://github.com/axieinfinity/ronin/pull/220
- https://github.com/axieinfinity/ronin/pull/206

```
+               return false
+       }
        if header1.ToHeader().ParentHash.Hex() != header2.ToHeader().ParentHash.Hex() {
                return false
        }
        if len(header1.ExtraData) < crypto.SignatureLength || len(header2.ExtraData) < crypto.SignatureLength {
                return false
        }
        if bytes.Equal(SealHash(header1.ToHeader(), header1.ChainId).Bytes(), SealHash(header2.ToHeader(), header2.ChainId).Bytes()) {
                return false
        }
        signer1, err := c.getSigner(header1)
        if err != nil {
                log.Trace("[consortiumVerifyHeaders][verify] error while getting signer from header1", "err", err)
                return false
        }
        signer2, err := c.getSigner(header2)
        if err != nil {
                log.Trace("[consortiumVerifyHeaders][verify] error while getting signer from header2", "err", err)
                return false
        }
-       return signer1.Hex() == signer2.Hex() && signer2.Hex() == header2.Benificiary.Hex()
+       methodAbi, _ := abi.JSON(strings.NewReader(getDoubleSignSlashingConfigsAbi))
+
+       if c.test {
+               maxOffset = big.NewInt(doubleSigningOffsetTest)
+       } else {
+               if c.evm.chainConfig.ConsortiumV2Contracts == nil {
+                       return false
+               } else {
+                       rawMaxOffset, err := staticCall(
+                               c.evm,
+                               methodAbi,
+                               getDoubleSignSlashingConfigs,
+                               c.evm.chainConfig.ConsortiumV2Contracts.SlashIndicator,
+                               common.Address{},
+                       )
+                       if err != nil {
+                               log.Error("Failed to get double sign config", "err", err)
+                               return false
+                       }
+                       if len(rawMaxOffset) < 3 {
+                               log.Error("Invalid output length when getting double sign config", "length", len(rawMaxOffset))
+                               return false
+                       }
+                       maxOffset = rawMaxOffset[2].(*big.Int)
+               }
+       }
+
+       currentBlock := c.evm.Context.BlockNumber
+       // What if current block < header1.Number?
+       if currentBlock.Cmp(header1.Number) > 0 && new(big.Int).Sub(currentBlock, header1.Number).Cmp(maxOffset) > 0 {
+               return false
+       }
+
+       return signer1.Hex() == signer2.Hex() &&
+               signer2.Hex() == header2.Benificiary.Hex() &&
+               bytes.Equal(consensusAddr.Bytes(), signer1.Bytes())
}
```

Figure 6 Source code of *verify* method (Partially fixed)

```
676 +
677 +                       if w.chainConfig.IsConsortiumV2(block.Number()) {
678 +                               if parents, ok := w.recentMinedBlocks.Get(block.NumberU64()); ok {
679 +                                       isDoubleSign := false
680 +                                       parentsHash := parents.([]common.Hash)
681 +                                       for _, parent := range parentsHash {
682 +                                               if block.ParentHash() == parent {
683 +                                                       isDoubleSign = true
684 +                                                       log.Info("Possibly double sign, drop block", "block", block.Number(), "hash", block.Hash(),
    "parent", block.ParentHash())
685 +                                               }
686 +                                       }
687 +                                       // Don't broadcast and write this block to chain,
688 +                                       // continue receiving next sealed block from result channel
689 +                                       if isDoubleSign {
690 +                                               continue
691 +                                       }
692 +                                       parentsHash = append(parentsHash, block.ParentHash())
693 +                                       w.recentMinedBlocks.Add(block.NumberU64(), parentsHash)
694 +                               } else {
695 +                                       w.recentMinedBlocks.Add(block.NumberU64(), []common.Hash{block.ParentHash()})
696 +                               }
697 +                       }
698 +
```

Figure 7 Source code of *worker.resultLoop* method

[Ronin Node-3] Changing the coinbase by the validator node will result in incorrect block generation

Severity Level	Info
Lines	core/block_validator.go #L99
Description	When the validator node calls miner.setEtherbase API to change the coinbase, it will likely cause the Merkle Patricia Trie to be abnormal, thus packing the wrong block in consortium V1.

```
func (v *BlockValidator) ValidateState(block *types.Block, statedb *state.StateDB, receipts types.Receipts, usedGas uint64) error {
    header := block.Header()
    if block.GasUsed() != usedGas {
        return fmt.Errorf("invalid gas used (remote: %d local: %d)", block.GasUsed(), usedGas)
    }
    // Validate the received block's bloom with the one derived from the generated receipts.
    // For valid blocks this should always validate to true.
    rbloom := types.CreateBloom(receipts)
    if rbloom != header.Bloom {
        return fmt.Errorf("invalid bloom (remote: %x  local: %x)", header.Bloom, rbloom)
    }
    // Tre receipt Trie's root (R = (Tr [[H1, R1], ... [Hn, Rn]]))
    receiptSha := types.DeriveSha(receipts, trie.NewStackTrie(nil))
    if receiptSha != header.ReceiptHash {
        return fmt.Errorf("invalid receipt root hash (remote: %x local: %x)", header.ReceiptHash, receiptSha)
    }
    // Validate the state root against the received state root and throw
    // an error if they don't match.
    if root := statedb.IntermediateRoot(v.config.IsEIP158(header.Number)); header.Root != root {
        return fmt.Errorf("invalid merkle root (remote: %x local: %x)", header.Root, root)
    }
    return nil
}
```

Figure 8 Source code of *validateState* method

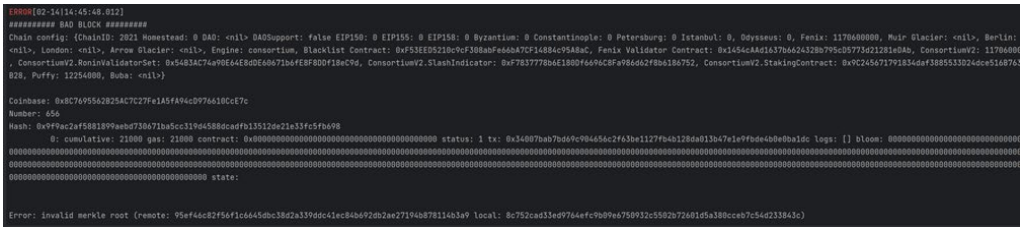


Figure 9 bad block

Recommendations	It is recommended to disable miner.setEtherbase API in source code.
Status	Acknowledged. According to the description of the project party, the namespace "miner" is used for administration configuration, so the project party don't enable methods in that namespace by default. Moreover, on validator nodes, http RPC is bound to 127.0.0.1 only. And the vulnerability does not exist when using consortium V2.

Findings for Consensus Contracts

Index	Risk description	Severity level	Status
DPoS Smart Contracts-1	Wrong use of _candidates as current validator	High	Fixed
DPoS Smart Contracts-2	Credit score update exception	Medium	Fixed
DPoS Smart Contracts-3	Will cause the validator to exit unexpectedly	Medium	Fixed
DPoS Smart Contracts-4	The proposal vote will result in defeat	Low	Fixed
DPoS Smart Contracts-5	Parameter assignment error	Low	Fixed
DPoS Smart Contracts-6	*SlashDoubleSign* function call is not implemented	Low	Fixed
DPoS Smart Contracts-7	Redundant codes	Info	Fixed
DPoS Smart Contracts-8	The *_updateTrustedOrganization* function is not designed properly	Info	Fixed

Finding Details:

[DPoS Smart Contracts-1] Wrong use of _candidates as current validator

Severity Level	High
Type	Business Security
Lines	ronin/validator/CoinbaseExecution.sol #L445
Description	The incorrect use of _candidates in the *_revampRoles* function of the CoinbaseExecution contract to get the value of *_maintainedList* will result in a possible exception in data processing.

```
439   */
440   function _revampRoles(
441     uint256 _newPeriod,
442     uint256 _nextEpoch,
443     address[] memory _currentValidators
444   ) private {
445     bool[] memory _maintainedList = _maintenanceContract.checkManyMaintained(_candidates, block.number + 1);
446
447     for (uint _i = 0; _i < _currentValidators.length; _i++) {
448       address _validator = _currentValidators[_i];
449       bool _emergencyExitRequested = block.timestamp <= _emergencyExitJailedTimestamp[_validator];
450       bool _isProducerBefore = isBlockProducer(_validator);
451       bool _isProducerAfter = !(_jailed(_validator) || _maintainedList[_i] || _emergencyExitRequested);
452
453       if (!_isProducerBefore && _isProducerAfter) {
454         _validatorMap[_validator] = _validatorMap[_validator].addFlag(EnumFlags.ValidatorFlag.BlockProducer);
455       } else if (_isProducerBefore && !_isProducerAfter) {
456         _validatorMap[_validator] = _validatorMap[_validator].removeFlag(EnumFlags.ValidatorFlag.BlockProducer);
457       }
458
459       bool _isBridgeOperatorBefore = isOperatingBridge(_validator);
460       bool _isBridgeOperatorAfter = !_emergencyExitRequested;
461       if (!_isBridgeOperatorBefore && _isBridgeOperatorAfter) {
462         _validatorMap[_validator] = _validatorMap[_validator].addFlag(EnumFlags.ValidatorFlag.BridgeOperator);
463       } else if (_isBridgeOperatorBefore && !_isBridgeOperatorAfter) {
464         _validatorMap[_validator] = _validatorMap[_validator].removeFlag(EnumFlags.ValidatorFlag.BridgeOperator);
465       }
466     }
467
468     emit BlockProducerSetUpdated(_newPeriod, _nextEpoch, getBlockProducers());
469     emit BridgeOperatorSetUpdated(_newPeriod, _nextEpoch, getBridgeOperators());
470   }
```

Figure 10 Source code of *_revampRoles* function (Unfixed)

Recommendations	It is recommended to change _candidates to _currentValidators.
Status	Fixed. See the PR for the fix details: https://github.com/axieinfinity/ronin-dpos-contracts/pull/171

```
439   */
440   function _revampRoles(
441     uint256 _newPeriod,
442     uint256 _nextEpoch,
443     address[] memory _currentValidators
444   ) private {
445     bool[] memory _maintainedList = _maintenanceContract.checkManyMaintained(_currentValidators, block.number + 1);
446
447     for (uint _i = 0; _i < _currentValidators.length; _i++) {
448       address _validator = _currentValidators[_i];
449       bool _emergencyExitRequested = block.timestamp <= _emergencyExitJailedTimestamp[_validator];
450       bool _isProducerBefore = isBlockProducer(_validator);
451       bool _isProducerAfter = !(_jailed(_validator) || _maintainedList[_i] || _emergencyExitRequested);
452
453       if (!_isProducerBefore && _isProducerAfter) {
454         _validatorMap[_validator] = _validatorMap[_validator].addFlag(EnumFlags.ValidatorFlag.BlockProducer);
455       } else if (_isProducerBefore && !_isProducerAfter) {
456         _validatorMap[_validator] = _validatorMap[_validator].removeFlag(EnumFlags.ValidatorFlag.BlockProducer);
457       }
458
459       bool _isBridgeOperatorBefore = isOperatingBridge(_validator);
460       bool _isBridgeOperatorAfter = !_emergencyExitRequested;
461       if (!_isBridgeOperatorBefore && _isBridgeOperatorAfter) {
462         _validatorMap[_validator] = _validatorMap[_validator].addFlag(EnumFlags.ValidatorFlag.BridgeOperator);
463       } else if (_isBridgeOperatorBefore && !_isBridgeOperatorAfter) {
464         _validatorMap[_validator] = _validatorMap[_validator].removeFlag(EnumFlags.ValidatorFlag.BridgeOperator);
465       }
466     }
467
468     emit BlockProducerSetUpdated(_newPeriod, _nextEpoch, getBlockProducers());
469     emit BridgeOperatorSetUpdated(_newPeriod, _nextEpoch, getBridgeOperators());
470   }
471
```

Figure 11 Source code of *revampRoles* function (Fixed)

[DPoS Smart Contracts-2] Credit score update exception

Severity Level	Medium
Type	Business Security
Lines	ronin/validator/CoinbaseExecution.sol #L108
Description	When the *wrapUpEpoch* function is triggered at the end of the period, it will first update *_currentPeriodStartAtBlock* to *block.number+1*, which will affect the *updateCreditScores* function from getting *_periodStartAtBlock* to get currentPeriodStartAtBlock is *block.number+1*, which causes the _maintaineds status of all *_validators* in *checkManyMaintainedInBlockRange* to return a value of false, and the *_maintaineds* status of all *_validators* in *checkManyMaintainedInBlockRange* to return a value of false when performing the judgment (*_ isJailedInPeriod* || *_isMaintainingInPeriod*) will increase the value of *_actualGain* even if the *_validators* are in the maintained state.

```
 */
function wrapUpEpoch() external payable virtual override onlyCoinbase whenEpochEnding oncePerEpoch {
  uint256 _newPeriod = _computePeriod(block.timestamp);
  bool _periodEnding = _isPeriodEnding(_newPeriod);

  address[] memory _currentValidators = getValidators();
  address[] memory _revokedCandidates;
  uint256 _epoch = epochOf(block.number);
  uint256 _nextEpoch = _epoch + 1;
  uint256 _lastPeriod = currentPeriod();

  if (_periodEnding) {
    _currentPeriodStartAtBlock = block.number + 1;
    _syncBridgeOperatingReward(_lastPeriod, _currentValidators);
    (
      uint256 _totalDelegatingReward,
      uint256[] memory _delegatingRewards
    ) = _distributeRewardToTreasuriesAndCalculateTotalDelegatingReward(_lastPeriod, _currentValidators);
    _settleAndTransferDelegatingRewards(_lastPeriod, _currentValidators, _totalDelegatingReward, _delegatingRewards);
    _tryRecycleLockedFundsFromEmergencyExits();
    _recycleDeprecatedRewards();
    _slashIndicatorContract.updateCreditScores(_currentValidators, _lastPeriod);
    (_currentValidators, _revokedCandidates) = _syncValidatorSet(_newPeriod);
    if (_revokedCandidates.length > 0) {
      _slashIndicatorContract.execResetCreditScores(_revokedCandidates);
    }
  }
  _revampRoles(_newPeriod, _nextEpoch, _currentValidators);
  emit WrappedUpEpoch(_lastPeriod, _epoch, _periodEnding);
  _periodOf[_nextEpoch] = _newPeriod;
  _lastUpdatedPeriod = _newPeriod;
}
```

Figure 12 Source code of *wrapUpEpoch* function (Unfixed)

Recommendations	It is recommended to put the _currentPeriodStartAtBlock update after updateCreditScores.
Status	Fixed. See the PR for the fix details: https://github.com/axieinfinity/ronin-dpos-contracts/pull/172

```
 */
function wrapUpEpoch() external payable virtual override onlyCoinbase whenEpochEnding oncePerEpoch {
  uint256 _newPeriod = _computePeriod(block.timestamp);
  bool _periodEnding = _isPeriodEnding(_newPeriod);

  address[] memory _currentValidators = getValidators();
  address[] memory _revokedCandidates;
  uint256 _epoch = epochOf(block.number);
  uint256 _nextEpoch = _epoch + 1;
  uint256 _lastPeriod = currentPeriod();

  if (_periodEnding) {
    _syncBridgeOperatingReward(_lastPeriod, _currentValidators);
    (
      uint256 _totalDelegatingReward,
      uint256[] memory _delegatingRewards
    ) = _distributeRewardToTreasuriesAndCalculateTotalDelegatingReward(_lastPeriod, _currentValidators);
    _settleAndTransferDelegatingRewards(_lastPeriod, _currentValidators, _totalDelegatingReward, _delegatingRewards);
    _tryRecycleLockedFundsFromEmergencyExits();
    _recycleDeprecatedRewards();
    _slashIndicatorContract.updateCreditScores(_currentValidators, _lastPeriod);
    (_currentValidators, _revokedCandidates) = _syncValidatorSet(_newPeriod);
    if (_revokedCandidates.length > 0) {
      _slashIndicatorContract.execResetCreditScores(_revokedCandidates);
    }
    _currentPeriodStartAtBlock = block.number + 1;
  }
  _revampRoles(_newPeriod, _nextEpoch, _currentValidators);
  emit WrappedUpEpoch(_lastPeriod, _epoch, _periodEnding);
  _periodOf[_nextEpoch] = _newPeriod;
  _lastUpdatedPeriod = _newPeriod;
}
```

Figure 13 Source code of *wrapUpEpoch* function (Fixed)

[DPoS Smart Contracts-3] Will cause the validator to exit unexpectedly

Severity Level	Medium
Type	Business Security
Lines	ronin/validator/CoinbaseExecution.sol #L403-428
Description	The *_setNewValidatorSet* function of the CoinbaseExecution contract is not designed properly and will result in the validator's EnumFlags.ValidatorFlag. As an example, when the validator of the previous round is [A,B,C,D] and the new round is [A,C,B,D], when setting validator B, it will clear the _validatorMap of validator C, which will cause validator C to exit.

```
403  function _setNewValidatorSet(
404    address[] memory _newValidators,
405    uint256 _newValidatorCount,
406    uint256 _newPeriod
407  ) private {
408    for (uint256 _i = _newValidatorCount; _i < validatorCount; _i++) {
409      delete _validatorMap[_validators[_i]];
410      delete _validators[_i];
411    }
412
413    uint256 _count;
414    for (uint256 _i = 0; _i < _newValidatorCount; _i++) {
415      address _newValidator = _newValidators[_i];
416      if (_newValidator == _validators[_count]) {
417        _count++;
418        continue;
419      }
420
421      delete _validatorMap[_validators[_count]];
422      _validatorMap[_newValidator] = EnumFlags.ValidatorFlag.Both;
423      _validators[_count] = _newValidator;
424      _count++;
425    }
426
427    validatorCount = _count;
428    emit ValidatorSetUpdated(_newPeriod, _newValidators);
429  }
```

Figure 14 Source code of *_setNewValidatorSet* function (Unfixed)

Recommendations	It is recommended to remove the validator information of the previous round and then update the validator information of this round.
Status	Fixed. See the PR for the fix details: https://github.com/axieinfinity/ronin-dpos-contracts/pull/182

```
402   function _setNewValidatorSet(
403     address[] memory _newValidators,
404     uint256 _newValidatorCount,
405     uint256 _newPeriod
406   ) private {
407     // Remove exceeding validators in the current set
408     for (uint256 _i = _newValidatorCount; _i < validatorCount; _i++) {
409       delete _validatorMap[_validators[_i]];
410       delete _validators[_i];
411     }
412
413     // Remove flag for all validator in the current set
414     for (uint _i = 0; _i < _newValidatorCount; _i++) {
415       delete _validatorMap[_validators[_i]];
416     }
417
418     // Update new validator set and set flag correspondingly.
419     for (uint256 _i = 0; _i < _newValidatorCount; _i++) {
420       address _newValidator = _newValidators[_i];
421       _validatorMap[_newValidator] = EnumFlags.ValidatorFlag.Both;
422       _validators[_i] = _newValidator;
423     }
424
425     validatorCount = _newValidatorCount;
426     emit ValidatorSetUpdated(_newPeriod, _newValidators);
427   }
```

Figure 15 Source code of *_setNewValidatorSet* function (Fixed)

[DPoS Smart Contracts-4] The proposal vote will result in defeat

Severity Level	**Low**
Type	Business Security
Lines	ronin/extensions/sequential-governance/CoreGovernance.sol #L118
Description	In the *proposal* function of the RoninGovernanceAdmin contract, when the Governor submits a proposal, the nonce is added to 1. But in the *_createVotingRound* function it can lead to a situation where _round is not equal to nonce. When the return value of _latestProposalVote is true (the case of an expired proposal), this time the proposal will overwrite the previous one. Since round is not equal to nonce, it can lead to a situation where the user fails to vote on the proposal, and since the proposal is serial (this proposal ends before the proposal can be initiated), it will result in the inability to perform another proposal vote.

```
106  function _proposeProposal(
107    uint256 _chainId,
108    uint256 _expiryTimestamp,
109    address[] memory _targets,
110    uint256[] memory _values,
111    bytes[] memory _calldatas,
112    uint256[] memory _gasAmounts,
113    address _creator
114  ) internal virtual returns (Proposal.ProposalDetail memory _proposal) {
115    require(_chainId != 0, "CoreGovernance: invalid chain id");
116
117    _proposal = Proposal.ProposalDetail(
118      round[_chainId] + 1,
119      _chainId,
120      _expiryTimestamp,
121      _targets,
122      _values,
123      _calldatas,
124      _gasAmounts
125    );
126    _proposal.validate(_proposalExpiryDuration);
127
128    bytes32 _proposalHash = _proposal.hash();
129    uint256 _round = _createVotingRound(_chainId, _proposalHash, _expiryTimestamp);
130    emit ProposalCreated(_chainId, _round, _proposalHash, _proposal, _creator);
131  }
```

Figure 16 Source code of *_proposeProposal* function (Unfixed)

Recommendations	It is recommended that nonce and _round be consistent.
Status	Fixed. See the PR for the fix details: https://github.com/axieinfinity/ronin-dpos-contracts/pull/167.

```
106  function _proposeProposal(
107    uint256 _chainId,
108    uint256 _expiryTimestamp,
109    address[] memory _targets,
110    uint256[] memory _values,
111    bytes[] memory _calldatas,
112    uint256[] memory _gasAmounts,
113    address _creator
114  ) internal virtual returns (Proposal.ProposalDetail memory _proposal) {
115    require(_chainId != 0, "CoreGovernance: invalid chain id");
116    uint256 _round = _calculateVotingRoundAndDeleteExpiry(_chainId);
117
118    _proposal = Proposal.ProposalDetail(_round, _chainId, _expiryTimestamp, _targets, _values, _calldatas, _gasAmounts);
119    _proposal.validate(_proposalExpiryDuration);
120
121    bytes32 _proposalHash = _proposal.hash();
122    _createVotingRound(_chainId, _round, _proposalHash, _expiryTimestamp);
123    emit ProposalCreated(_chainId, _round, _proposalHash, _proposal, _creator);
124  }
125
```

Figure 17 Source code of *_proposeProposal* function (Fixed)

[DPoS Smart Contracts-5] Parameter assignment error

Severity Level	**Low**
Type	Business Security
Lines	ronin/slash-indicator/SlashBridgeVoting.sol #131-132
Description	In the *_setBridgeVotingSlashingConfigs* function in the SlashBridgeVoting contract, the *_slashAmount* parameter and the *_bridgeVotingSlashAmount* parameter are in the wrong place, which will cause the *_bridgeVotingSlashAmount* cannot be assigned a value.

```
62  function _setBridgeVotingSlashingConfigs(uint256 _threshold, uint256 _slashAmount) internal {
63    _bridgeVotingThreshold = _threshold;
64    _slashAmount = _bridgeVotingSlashAmount;
65    emit BridgeVotingSlashingConfigsUpdated(_threshold, _slashAmount);
66  }
67 }
68
```

Figure 18 Source code of *_setBridgeVotingSlashingConfigs*function (Unfixed)

Recommendations	It is recommended to replace the parameter positions in the function.
Status	Fixed. See the PR for the fix details: https://github.com/axieinfinity/ronin-dpos-contracts/pull/179

```
66      */
67     function _setBridgeVotingSlashingConfigs(uint256 _threshold, uint256 _slashAmount) internal {
68       _bridgeVotingThreshold = _threshold;
69  +    _bridgeVotingSlashAmount = _slashAmount;
70       emit BridgeVotingSlashingConfigsUpdated(_threshold, _slashAmount);
71     }
72   }
```

Figure 19 Source code of *setBridgeVotingSlashingConfigs* function (Fixed)

[DPoS Smart Contracts-6] *SlashDoubleSign* function call is not implemented

Severity Level	**Low**
Type	Business Security
Lines	ronin/slash-indicator/SlashDoubleSign.sol #29
Description	The *SlashDoubleSign* function is used to punish Double-sign in the contract, but this function is not called in consortiumV2. According to the project party: The double sign function is intended to be called manually by anyone who has the proof of double sign attack, but as the source code shows, it can only be called through the coinbase account, which is not the intended design.

```
22    * @inheritdoc ISlashDoubleSign
23    */
24   function slashDoubleSign(
25     address _consensusAddr,
26     bytes calldata _header1,
27     bytes calldata _header2
28   ) external override {
29     require(msg.sender == block.coinbase, "SlashIndicator: method caller must be coinbase");
30     if (!_shouldSlash(_consensusAddr)) {
31       return;
32     }
33
34     if (_pcValidateEvidence(_header1, _header2)) {
35       uint256 _period = _validatorContract.currentPeriod();
36       emit Slashed(_consensusAddr, SlashType.DOUBLE_SIGNING, _period);
37       _validatorContract.execSlash(_consensusAddr, _doubleSigningJailUntilBlock, _slashDoubleSignAmount, false);
38     }
39   }
40
```

Figure 20 Source code of *SlashDoubleSign* function (Unfixed)

Recommendations	It is recommended to remove redundant code for related functions.
Status	Fixed. The *SlashDoubleSign* function has added admin permission and added penalty status records. See the PR for the fix details: https://github.com/axieinfinity/ronin-dpos-contracts/pull/180

```
24  30    function slashDoubleSign(
25        -   address _consensusAddr,
    31    +   address _consensusAddr,
26  32        bytes calldata _header1,
27  33        bytes calldata _header2
28        -   ) external override {
29        -   require(msg.sender == block.coinbase, "SlashIndicator: method caller must be coinbase");
30        -   if (!_shouldSlash(_consensusAddr)) {
31        -     return;
32        -   }
    34    +   ) external override onlyAdmin {
    35    +   bytes32 _header1Checksum = keccak256(_header1);
    36    +   bytes32 _header2Checksum = keccak256(_header2);
    37    +
    38    +   require(
    39    +     !_submittedEvidence[_header1Checksum] && !_submittedEvidence[_header2Checksum],
    40    +     "SlashDoubleSign: evidence already submitted"
    41    +   );
33  42
34        -   if (_pcValidateEvidence(_header1, _header2)) {
    43    +   if (_pcValidateEvidence(_consensusAddr, _header1, _header2)) {
35  44        uint256 _period = _validatorContract.currentPeriod();
36        -     emit Slashed(_consensusAddr, SlashType.DOUBLE_SIGNING, _period);
37        -     _validatorContract.execSlash(_consensusAddr, _doubleSigningJailUntilBlock, _slashDoubleSignAmount, false);
    45    +     _submittedEvidence[_header1Checksum] = true;
    46    +     _submittedEvidence[_header2Checksum] = true;
    47    +     emit Slashed(_consensusAddr, SlashType.DOUBLE_SIGNING, _period);
    48    +     _validatorContract.execSlash(_consensusAddr, _doubleSigningJailUntilBlock, _slashDoubleSignAmount, true);
38  49      }
39  50    }
```

Figure 21 Source code of *SlashDoubleSign* function (fixed)

[DPoS Smart Contracts-7] Redundant codes

Severity Level	Info
Type	Coding Conventions
Lines	ronin/BridgeTracking.sol #L205-208
Description	The *_trySyncPeriodStats* function in the BridgeTracking.sol contract has redundant code. In the function of the if block *_validatorContract. TryGetPeriodOfEpoch(_temporaryStats. LastEpoch + 1)* return _filled variable value is always true. Since *_temporaryStats.lastEpoch < _currentEpoch* to enter the if block, when executing *tryGetPeriodOfEpoch(_temporaryStats.lastEpoch+1)* in the if block, Since *_epoch <= epochOf(block.number)* is always true, the value returned is always true.

```
202  function _trySyncPeriodStats() internal {
203    uint256 _currentEpoch = _validatorContract.epochOf(block.number);
204    if (_temporaryStats.lastEpoch < _currentEpoch) {
205      (bool _filled, uint256 _period) = _validatorContract.tryGetPeriodOfEpoch(_temporaryStats.lastEpoch + 1);
206      if (!_filled) {
207        return;
208      }
209
210      VoteStats storage _stats = _periodStats[_period];
211      _stats.totalVotes += _temporaryStats.info.totalVotes;
212      _stats.totalBallots += _temporaryStats.info.totalBallots;
213
214      address _voter;
215      for (uint _i = 0; _i < _temporaryStats.info.voters.length; _i++) {
216        _voter = _temporaryStats.info.voters[_i];
217        _stats.totalBallotsOf[_voter] += _temporaryStats.info.totalBallotsOf[_voter];
218        delete _temporaryStats.info.totalBallotsOf[_voter];
219      }
220      delete _temporaryStats.info;
221      _temporaryStats.lastEpoch = _currentEpoch;
222    }
223  }
```

Figure 22 Source code of *_trySyncPeriodStats* function (Unfixed)

Recommendations	It is recommended to remove redundant code for related functions.
Status	Fixed. See the PR for the fix details: https://github.com/axieinfinity/ronin-dpos-contracts/pull/170

```
44     * @inheritdoc ITimingInfo
45     */
46    function tryGetPeriodOfEpoch(uint256 _epoch) external view returns (bool _filled, uint256 _periodNumber) {
47      return (_periodOf[_epoch] > 0, _periodOf[_epoch]);
48    }
49
```

Figure 23 Source code of *tryGetPeriodOfEpoch* function (Fixed)

[DPoS Smart Contracts-8] The *_updateTrustedOrganization* function is not designed properly

Severity Level	Info
Type	Business Security
Lines	src\registries\NodeOperatorRegistry.sol #L172-178
Description	In the *_addTrustedOrganization* function of the RoninTrustedOrganization contract, when setting the address, it is required that the consensusAddr, governor and bridgeVoter addresses cannot be the same, but when calling the *_updateTrustedOrganization* function to update, it does not check that the consensusAddr, governor and bridgeVoter addresses cannot be the same. In the *updateTrustedOrganization* function, there is no check that the consensusAddr, governor and bridgeVoter addresses cannot be the same, so you can set all three addresses to be the same by using the *_updateTrustedOrganization* function.

```
332  function _updateTrustedOrganization(TrustedOrganization memory _v) internal virtual {
333    require(_v.weight > 0, "RoninTrustedOrganization: invalid weight");
334
335    uint256 _weight = _consensusWeight[_v.consensusAddr];
336    if (_weight == 0) {
337      revert(
338        string(
339          abi.encodePacked(
340            "RoninTrustedOrganization: consensus address ",
341            Strings.toHexString(uint160(_v.consensusAddr), 20),
342            " is not added"
343          )
344        )
345      );
346    }
347
348    uint256 _count = _consensusList.length;
349    for (uint256 _i = 0; _i < _count; _i++) {
350      if (_consensusList[_i] == _v.consensusAddr) {
351        _totalWeight -= _weight;
352        _totalWeight += _v.weight;
353
354        if (_governorList[_i] != _v.governor) {
355          require(_governorWeight[_v.governor] == 0, "RoninTrustedOrganization: query for duplicated governor");
356          delete _governorWeight[_governorList[_i]];
357          _governorList[_i] = _v.governor;
358        }
359
360        if (_bridgeVoterList[_i] != _v.bridgeVoter) {
361          require(
362            _bridgeVoterWeight[_v.bridgeVoter] == 0,
363            "RoninTrustedOrganization: query for duplicated bridge voter"
364          );
365          delete _bridgeVoterWeight[_bridgeVoterList[_i]];
366          _bridgeVoterList[_i] = _v.bridgeVoter;
367        }
368
369        _consensusWeight[_v.consensusAddr] = _v.weight;
370        _governorWeight[_v.governor] = _v.weight;
371        _bridgeVoterWeight[_v.bridgeVoter] = _v.weight;
372        return;
373      }
374    }
375  }
```

Figure 24 Source code of *_updateTrustedOrganization* function (Unfixed)

Recommendations	It is recommended to add the same address check.
Status	Fixed. See the PR for the fix details: https://github.com/axieinfinity/ronin-dpos-contracts/pull/174

```
325     */
326   function _updateTrustedOrganization(TrustedOrganization memory _v) internal virtual {
327     _sanityCheckTODData(_v);
328
329     uint256 _weight = _consensusWeight[_v.consensusAddr];
330     if (_weight == 0) {
331       revert(
332         string(
333           abi.encodePacked(
334             "RoninTrustedOrganization: consensus address ",
335             Strings.toHexString(uint160(_v.consensusAddr), 20),
336             " is not added"
337           )
338         )
339       );
340     }
341
342     uint256 _count = _consensusList.length;
343     for (uint256 _i = 0; _i < _count; _i++) {
344       if (_consensusList[_i] == _v.consensusAddr) {
345         _totalWeight -= _weight;
346         _totalWeight += _v.weight;
347
348         if (_governorList[_i] != _v.governor) {
349           require(_governorWeight[_v.governor] == 0, "RoninTrustedOrganization: query for duplicated governor");
350           delete _governorWeight[_governorList[_i]];
351           _governorList[_i] = _v.governor;
352         }
353
354         if (_bridgeVoterList[_i] != _v.bridgeVoter) {
355           require(
356             _bridgeVoterWeight[_v.bridgeVoter] == 0,
357             "RoninTrustedOrganization: query for duplicated bridge voter"
358           );
359           delete _bridgeVoterWeight[_bridgeVoterList[_i]];
360           _bridgeVoterList[_i] = _v.bridgeVoter;
361         }
362
363         _consensusWeight[_v.consensusAddr] = _v.weight;
364         _governorWeight[_v.governor] = _v.weight;
365         _bridgeVoterWeight[_v.bridgeVoter] = _v.weight;
366         return;
367       }
368     }
369   }
```

Figure 25 Source code of *_updateTrustedOrganization* function (Fixed)

Blockchain Audit Contents

The Ronin is a modular and extensible framework for building Ethereum-compatible blockchain networks and general scaling solutions.

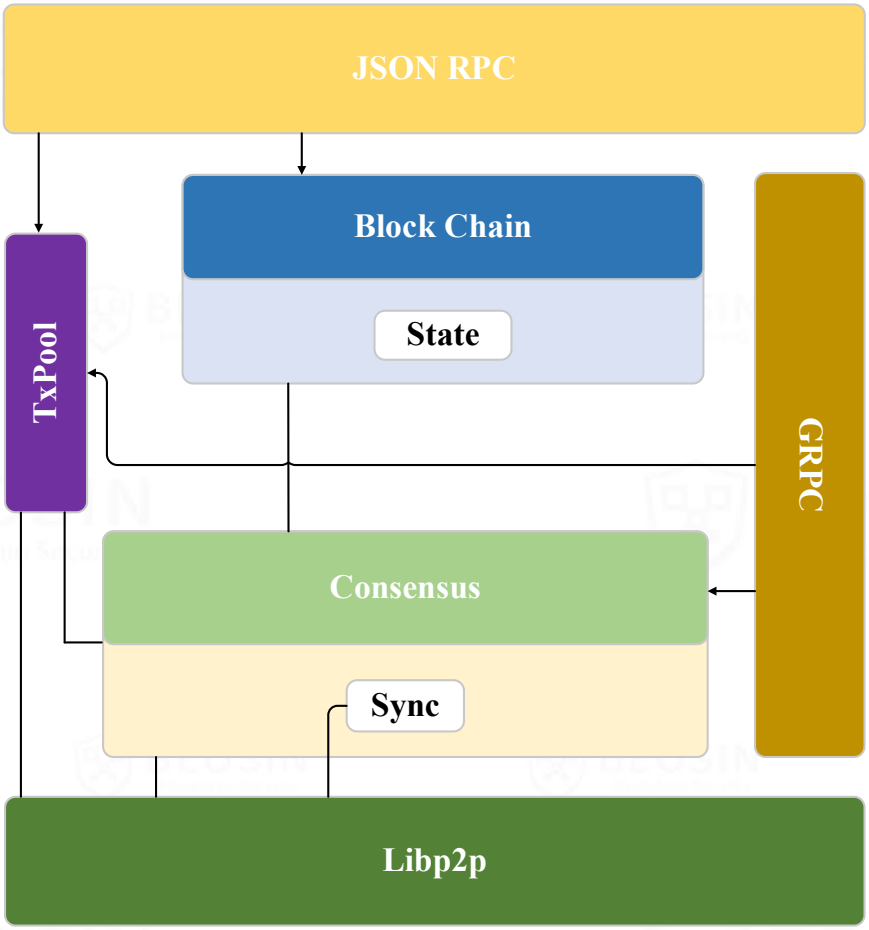


Figure 26 The Ronin overall framework

1 JSON RPC Security Audit

1.1 JSON RPC Introduction

RPC provides a way to access the exported objects through the grid or other I/O connections. RPC serializes the methods and parameters to be called and transmits them to the other side through TCP/UDP protocol, and the other side deserializes them to find the methods and parameters to be called, and then serializes the return values to return. After creating the RPC service, the object can be registered to the service to make it accessible to the outside. The export method according to this specific way is called remote call, which also supports the publish/subscribe model.

When the Ronin node starts, it will pull up the JSON RPC service, register the corresponding RPC module, and use RPC or HTTP to call the corresponding interface. The JSON RPC implemented by Ronin node is basically the same as the RPC module commonly used in Ethereum, except that a consensus-related consortium module is added, which includes three methods: GetSigners, GetDBValue and GetAncientValue.

```go
// GetSigners retrieves the list of authorized signers at the specified block.
func (api *API) GetSigners(number *rpc.BlockNumber) ([]common.Address, error) {
    // Retrieve the requested block number (or current if none requested)
    var header *types.Header
    if number == nil || *number == rpc.LatestBlockNumber {
        header = api.chain.CurrentHeader()
    } else {
        header = api.chain.GetHeaderByNumber(uint64(number.Int64()))
    }
    // Ensure we have an actually valid block and return the signers from its snapshot
    if header == nil {
        return nil, consortiumCommon.ErrUnknownBlock
    }

    validators, err := api.consortium.getValidatorsFromLastCheckpoint(api.chain, header.Number.Uint64(), nil)
    if err != nil {
        return nil, err
    }
    return validators, nil
}

func (api *API) GetDBValue(key string) (string, error) {
    value, err := api.chain.DB().Get(common.Hex2Bytes(key))
    if err != nil {
        log.Debug("Get value in DB failed, try to get it from trie node", "key", key)
        if value, err = api.chain.StateCache().TrieDB().Node(common.HexToHash(key)); err != nil {
            return common.Bytes2Hex([]byte{}), err
        }
    }
    return common.Bytes2Hex(value), nil
}

func (api *API) GetAncientValue(kind string, number rpc.BlockNumber) (string, error) {
    value, err := api.chain.DB().Ancient(kind, uint64(number.Int64()))
    if err != nil {
        return common.Bytes2Hex([]byte{}), err
    }
    return common.Bytes2Hex(value), nil
}
```

Figure 27 JSON RPC consortium module

The consensus mechanism used in this audit contains three interfaces: GetSigners, GetDBValue and GetAncientValue. GetSigners is used to view the validator node address set of a block. GetDBValue is used to look up an encoded cache trie node in the database or in memory. GetAncientValue is used to retrieve an ancient binary blob from the append-only immutable files.

1.2 JSON RPC Processing Logic

In Ronin node, after receiving an RPC request, the node will call the *serveSingleRequest* function to check the request function type, to parse the corresponding RPC request and locate the corresponding module and function to call.

```go
// serveSingleRequest reads and processes a single RPC request from the given codec. This
// is used to serve HTTP connections. Subscriptions and reverse calls are not allowed in
// this mode.
func (s *Server) serveSingleRequest(ctx context.Context, codec ServerCodec) {
    // Don't serve if server is stopped.
    if atomic.LoadInt32(&s.run) == 0 {
        return
    }

    h := newHandler(ctx, codec, s.idgen, &s.services)
    h.allowSubscribe = false
    defer h.close(io.EOF, nil)

    reqs, batch, err := codec.readBatch()
    if err != nil {
        if err != io.EOF {
            codec.writeJSON(ctx, errorMessage(&invalidMessageError{"parse error"}))
        }
        return
    }
    if batch {
        h.handleBatch(reqs)
    } else {
        h.handleMsg(reqs[0])
    }
}
```

Figure 28 Source code of *serveSingleRequest* function

1.3 RPC Sensitive Interface Permission

The vast majority of the RPC interfaces currently provided by Ronin node are for querying data, and there are no high authority interfaces.

1.4 CLI Commands Security Audit

The Ronin's CLI is mainly for nodes to perform related configurations, and its corresponding parameters are as follows:

```
COMMANDS:
   account                  Manage accounts
   attach                   Start an interactive JavaScript environment (connect to node)
   console                  Start an interactive JavaScript environment
   db                       Low level database operations
   dump                     Dump a specific block from storage
   dumpconfig               Show configuration values
   dumpgenesis              Dumps genesis block JSON configuration to stdout
   export                   Export blockchain into file
   export-preimages         Export the preimage database into an RLP stream
   import                   Import a blockchain file
   import-preimages         Import the preimage database from an RLP stream
   init                     Bootstrap and initialize a new genesis block
   js                       Execute the specified JavaScript files
   license                  Display license information
   makecache                Generate ethash verification cache (for testing)
   makedag                  Generate ethash mining DAG (for testing)
   removedb                 Remove blockchain and state databases
   show-deprecated-flags    Show flags that have been deprecated
   snapshot                 A set of commands based on the snapshot
   version                  Print version numbers
   version-check            Checks (online) whether the current version suffers from any known security vulnerabilities
   wallet                   Manage Ethereum presale wallets
   help, h                  Shows a list of commands or help for one command
```

Figure 29 The Ronin node command line parameters

After testing, the relevant commands in the CLI meet the audit requirements and there is no security risk.

1.5 Node Account Unlocking Security Audit

Ronin uses the DPoS consensus. The DPoS consensus requires that the coinbase account of the node must be unlocked after it is activated, otherwise it will not be able to generate blocks. There is a potential risk here, that is, if the node opens the RPC interface to the public network, any user connected to the node can operate the node's coinbase, which may cause the node to crash when running the consortium V1 consensus. But according to the project party, this interface is only open to 127.0.0.1.

In theory, unlocked accounts have the above-mentioned risks on open JSON-RPC nodes (users cannot confirm whether their API is used by hackers to sign raw data, unless the log is turned on). Therefore, it is recommended to close the sensitive RPC or only open it to the whitelist addresses when the node is online.

2 Node Security

2.1 Number of Node Connections

The Ronin nodes can use the networking module to set parameters at startup, where --maxpeers limit the number of links to the node, and if the number of links exceeds the specified value, no links will be made.

```
NETWORKING OPTIONS:
  --bootnodes value                  Comma separated enode URLs for P2P discovery bootstrap
  --discovery.dns value              Sets DNS discovery entry points (use "" to disable DNS)
  --port value                       Network listening port (default: 30303)
  --maxpeers value                   Maximum number of network peers (network disabled if set to 0) (default: 50)
  --maxpendpeers value               Maximum number of pending connection attempts (defaults used if set to 0) (default: 0)
  --nat value                        NAT port mapping mechanism (any|none|upnp|pmp|extip:<IP>) (default: "any")
  --nodiscover                       Disables the peer discovery mechanism (manual peer addition)
  --v5disc                           Enables the experimental RLPx V5 (Topic Discovery) mechanism
  --netrestrict value                Restricts network communication to the given IP networks (CIDR masks)
  --nodekey value                    P2P node key file
  --nodekeyhex value                 P2P node key as hex (for testing)
```

Figure 30 Node start-up parameters

After testing, the Ronin nodes can limit the number of connections to the current node using the server parameter (which defaults even if not specified), which can prevent the node from having too many connections.

```
TRACE[02-24|16:24:15.042] Found seed node in database          id=4ae70c9793b68453 addr=127.0.0.1:30314 age=1m36.0426578s
TRACE[02-24|16:24:15.042] Found seed node in database          id=68934d9ff5e18b54 addr=127.0.0.1:30315 age=1m52.0426578s
TRACE[02-24|16:24:15.042] Found seed node in database          id=822150deaaa449a1 addr=127.0.0.1:30310 age=5m38.0426578s
TRACE[02-24|16:24:15.042] Failed p2p handshake                 id=4ae70c9793b68453 addr=127.0.0.1:30314 conn=dyndial err="too many peers"
TRACE[02-24|16:24:15.042] Found seed node in database          id=822150deaaa449a1 addr=127.0.0.1:30310 age=5m38.0426578s
```

Figure 31 Node connection information

2.2 Packet Size Limit

As shown in the figure below, the size of the data is checked when a transaction is received, which can avoid DoS attacks on nodes.

```go
func (pool *TxPool) validateTx(tx *types.Transaction, local bool) error {
    // Accept only legacy transactions until EIP-2718/2930 activates.
    if !pool.eip2718 && tx.Type() != types.LegacyTxType {
        return ErrTxTypeNotSupported
    }
    // Reject dynamic fee transactions until EIP-1559 activates.
    if !pool.eip1559 && tx.Type() == types.DynamicFeeTxType {
        return ErrTxTypeNotSupported
    }
    // Reject transactions over defined size to prevent DOS attacks
    if uint64(tx.Size()) > txMaxSize {
        return ErrOversizedData
    }
    // Transactions can't be negative. This may never happen using RLP decoded
    // transactions but may occur if you create a transaction using the RPC.
    if tx.Value().Sign() < 0 {
        return ErrNegativeValue
    }
    // Ensure the transaction doesn't exceed the current block limit gas.
    if pool.currentMaxGas < tx.Gas() {
        return ErrGasLimit
    }
    // Sanity check for extremely large numbers
    if tx.GasFeeCap().BitLen() > 256 {
        return ErrFeeCapVeryHigh
    }
    if tx.GasTipCap().BitLen() > 256 {
        return ErrTipVeryHigh
    }
```

Figure 32 Source code for the *validateTX* function

2.3 Node Network Access Restrictions

Network parameters can be set when the node starts the chain, which can protect the node from attacks to some extent.

```
NETWORKING OPTIONS:
  --bootnodes value                Comma separated enode URLs for P2P discovery bootstrap
  --discovery.dns value            Sets DNS discovery entry points (use "" to disable DNS)
  --port value                     Network listening port (default: 30303)
  --maxpeers value                 Maximum number of network peers (network disabled if set to 0) (default: 50)
  --maxpendpeers value             Maximum number of pending connection attempts (defaults used if set to 0) (default: 0)
  --nat value                      NAT port mapping mechanism (any|none|upnp|pmp|extip:<IP>) (default: "any")
  --nodiscover                     Disables the peer discovery mechanism (manual peer addition)
  --v5disc                         Enables the experimental RLPx V5 (Topic Discovery) mechanism
  --netrestrict value              Restricts network communication to the given IP networks (CIDR masks)
  --nodekey value                  P2P node key file
  --nodekeyhex value               P2P node key as hex (for testing)
```

Figure 33 The networking module parameters

3 Account &Asset Security

3.1 Account Model

The Ronin node uses a same account model with Ethereum, with the following basic data structure:

```
// Account is a modified version of a state.Account, where the root is replaced
// with a byte slice. This format can be used to represent full-consensus format
// or slim-snapshot format which replaces the empty root and code hash as nil
// byte slice.
type Account struct {
    Nonce    uint64
    Balance  *big.Int
    Root     []byte
    CodeHash []byte
}
```

Figure 34 Source code of the data structure of Account

- Nonce: The serial number of the transaction sent by the account;
- Balance: The balance of the account's platform coins;
- Root: The root hash []byte of the storage state tree;
- CodeHash: The EVM code bound to the account;

3.2 Account Generation

(1) externally owned account

The Ronin use Ethereum way to create external accounts

Through the RPC module personal to create an account interface: NewAccount

```
// NewAccount generates a new key and stores it into the key directory,
// encrypting it with the passphrase.
func (ks *KeyStore) NewAccount(passphrase string) (accounts.Account, error) {
    _, account, err := storeNewKey(ks.storage, crand.Reader, passphrase)
    if err != nil {
        return accounts.Account{}, err
    }
    // Add the account to the cache immediately rather
    // than waiting for file system notifications to pick it up.
    ks.cache.add(account)
    ks.refreshWallets()
    return account, nil
}
```

Figure 35 Source code of the *NewAccount* function

This interface will call the NewAccount function in accounts/keystore/keystore.go to generate an account and return the account address.

```
// NewAccount generates a new key and stores it into the key directory,
// encrypting it with the passphrase.
func (ks *KeyStore) NewAccount(passphrase string) (accounts.Account, error) {
    _, account, err := storeNewKey(ks.storage, crand.Reader, passphrase)
    if err != nil {
        return accounts.Account{}, err
    }
    // Add the account to the cache immediately rather
    // than waiting for file system notifications to pick it up.
    ks.cache.add(account)
    ks.refreshWallets()
    return account, nil
}
```

Figure 36 Source code of the *NewAccount* function

The NewAccount function in keystore.go will actually call the storeNewKey function in the accounts/keystore/key.go file to create an account, generate a public and private key, and store the created account in the cache.

```
func newKey(rand io.Reader) (*Key, error) {
    privateKeyECDSA, err := ecdsa.GenerateKey(crypto.S256(), rand)
    if err != nil {
        return nil, err
    }
    return newKeyFromECDSA(privateKeyECDSA), nil
}

func storeNewKey(ks keyStore, rand io.Reader, auth string) (*Key, accounts.Account, error) {
    key, err := newKey(rand)
    if err != nil {
        return nil, accounts.Account{}, err
    }
    a := accounts.Account{
        Address: key.Address,
        URL:     accounts.URL{Scheme: KeyStoreScheme, Path: ks.JoinPath(keyFileName(key.Addres
    }
    if err := ks.StoreKey(a.URL.Path, key, auth); err != nil {
        zeroKey(key.PrivateKey)
        return nil, a, err
    }
    return key, a, err
}
```

Figure 37 Source code of the *storeNewKey* function and *newKey* function

The storeNewKey function calls the newKey function to generate a complete key instance (including the account public and private key and account address), and calls the StoreKey function to write the generated account file to the local for persistent storage.

```go
// Generate an elliptic curve public / private keypair. If params is nil,
// the recommended default parameters for the key will be chosen.
func GenerateKey(rand io.Reader, curve elliptic.Curve, params *ECIESParams) (prv *PrivateKey, err error) {
    pb, x, y, err := elliptic.GenerateKey(curve, rand)
    if err != nil {
        return
    }
    prv = new(PrivateKey)
    prv.PublicKey.X = x
    prv.PublicKey.Y = y
    prv.PublicKey.Curve = curve
    prv.D = new(big.Int).SetBytes(pb)
    if params == nil {
        params = ParamsFromCurve(curve)
    }
    prv.PublicKey.Params = params
    return
}
```

Figure 38 Source code of the *GenerateKey* function

The *newKey* function will call *GenerateKey* function to generate the public and private keys.

```go
func newKeyFromECDSA(privateKeyECDSA *ecdsa.PrivateKey) *Key {
    id, err := uuid.NewRandom()
    if err != nil {
        panic(fmt.Sprintf("Could not create random uuid: %v", err))
    }
    key := &Key{
        Id:         id,
        Address:    crypto.PubkeyToAddress(privateKeyECDSA.PublicKey),
        PrivateKey: privateKeyECDSA,
    }
    return key
}
```

Figure 39 Source code of the *GnewkeyFromECDSA* function

Then pass the private key into the *newKeyFromECDSA* function, and then call the *PubkeyToAddress* function in the crypto/crypto.go file to calculate the account address according to the public key, thus generating a complete key instance.

```go
func PubkeyToAddress(p ecdsa.PublicKey) common.Address {
    pubBytes := FromECDSAPub(&p)
    return common.BytesToAddress(Keccak256(pubBytes[1:])[12:])
}
```

Figure 40 Source code of the *PubkeyToAddress* function

(2) contract account

Ronin node supports Ethereum Create and Create2 instructions to create contract accounts.

```
// Create creates a new contract using code as deployment code.
func (evm *EVM) Create(caller ContractRef, code []byte, gas uint64, value *big.Int) (ret []byte, contractAddr common.Address, leftOverGas uint64, err error) {
    contractAddr = crypto.CreateAddress(caller.Address(), evm.StateDB.GetNonce(caller.Address()))
    return evm.create(caller, &codeAndHash{code: code}, gas, value, contractAddr, CREATE)
}

// Create2 creates a new contract using code as deployment code.
//
// The different between Create2 with Create is Create2 uses sha3(0xff ++ msg.sender ++ salt ++ sha3(init_code))[12:]
// instead of the usual sender-and-nonce-hash as the address where the contract is initialized at.
func (evm *EVM) Create2(caller ContractRef, code []byte, gas uint64, endowment *big.Int, salt *uint256.Int) (ret []byte, contractAddr common.Address, leftOverGas uint64, err error) {
    codeAndHash := &codeAndHash{code: code}
    contractAddr = crypto.CreateAddress2(caller.Address(), salt.Bytes32(), codeAndHash.Hash().Bytes())
    return evm.create(caller, codeAndHash, gas, endowment, contractAddr, CREATE2)
}
```

Figure 41 Source code of the *Create* function and *Create2* function

3.3 Transaction Signature

First call the *SignTx* function in the accounts/keystore/keystore/wallet.go file to check whether the account is unlocked.

```
// SignTx implements accounts.Wallet, attempting to sign the given transaction
// with the given account. If the wallet does not wrap this particular account,
// an error is returned to avoid account leakage (even though in theory we may
// be able to sign via our shared keystore backend).
func (w *keystoreWallet) SignTx(account accounts.Account, tx *types.Transaction, chainID *big.Int) (*types.Transaction, error) {
    // Make sure the requested account is contained within
    if !w.Contains(account) {
        return nil, accounts.ErrUnknownAccount
    }
    // Account seems valid, request the keystore to sign
    return w.keystore.SignTx(account, tx, chainID)
}
```

Figure 42 Source code of the *SignTx* function

If it is confirmed that the account is valid, call the *SignTx* function in the accounts/keystore/keystore.go file to sign.

```
// SignTx signs the given transaction with the requested account.
func (ks *KeyStore) SignTx(a accounts.Account, tx *types.Transaction, chainID *big.Int) (*types.Transaction, error) {
    // Look up the key to sign with and abort if it cannot be found
    ks.mu.RLock()
    defer ks.mu.RUnlock()

    unlockedKey, found := ks.unlocked[a.Address]
    if !found {
        return nil, ErrLocked
    }
    // Depending on the presence of the chain ID, sign with 2718 or homestead
    signer := types.LatestSignerForChainID(chainID)
    return types.SignTx(tx, signer, unlockedKey.PrivateKey)
}
```

Figure 43 Source code of the *SignTx* function

Because the chainID is not empty, in order to prevent repeated attacks across chains, EIP155Signer is selected as the signer and the *SignTx* function in core/types/transaction_signing.go is called to complete the transaction signature.

```go
// SignTx signs the transaction using the given signer and private key.
func SignTx(tx *Transaction, s Signer, prv *ecdsa.PrivateKey) (*Transaction, error) {
    h := s.Hash(tx)
    sig, err := crypto.Sign(h[:], prv)
    if err != nil {
        return nil, err
    }
    return tx.WithSignature(s, sig)
}
```

Figure 44 Source code of the *SignTx* function

```go
func NewEIP155Signer(chainId *big.Int) EIP155Signer {
    if chainId == nil {
        chainId = new(big.Int)
    }
    return EIP155Signer{
        chainId:    chainId,
        chainIdMul: new(big.Int).Mul(chainId, big.NewInt(2)),
    }
}
```

Figure 45 Source code of the *NewEIP155Signer* function

3.4 Asset Security

(1) Platform Asset Security

After a comprehensive audit, it can be confirmed that when running the DPoS (consortium V2) consensus, no new Ronin native tokens will be generated when validator nodes package blocks (call *FinalizeAndAssemble* function). This means that all native tokens have been fully pre-mined, and no new native tokens will be generated afterwards.

```go
func (c *Consortium) FinalizeAndAssemble (chain consensus.ChainHeaderReader, header *types.Header, state *state.StateDB,
    txs []*types.Transaction, uncles []*types.Header, receipts []*types.Receipt) (*types.Block, []*types.Receipt, error) {
    if err := c.initContract(); err != nil {
        return nil, nil, err
    }

    // No block rewards in PoA, so the state remains as is and uncles are dropped
    if txs == nil {
        txs = make([]*types.Transaction, 0)
    }
    if receipts == nil {
        receipts = make([]*types.Receipt, 0)
    }
    evmContext := core.NewEVMBlockContext(header, consortiumCommon.ChainContext{Chain: chain, Consortium: c}, &header.Coinbase, chain.OpEvents()...)
    transactOpts := &consortiumCommon.ApplyTransactOpts{
        ApplyMessageOpts: &consortiumCommon.ApplyMessageOpts{
            State:       state,
            Header:      header,
            ChainConfig: c.chainConfig,
            EVMContext:  &evmContext,
        },
        Txs:      &txs,
        Receipts: &receipts,
        // a.k.a. System Txs
        // It always equals nil since FinalizeAndAssemble doesn't receive any transactions
        ReceivedTxs: nil,
        UsedGas:     &header.GasUsed,
        Mining:      true,
        Signer:      c.signer,
        SignTxFn:    c.signTxFn,
    }

    if err := c.processSystemTransactions(chain, header, transactOpts, true); err != nil {
        return nil, nil, err
    }

    // should not happen. Once happen, stop the node is better than broadcast the block
    if header.GasLimit < header.GasUsed {
        return nil, nil, errors.New("gas consumption of system txs exceed the gas limit")
    }
    header.UncleHash = types.CalcUncleHash(nil)
    var blk *types.Block
    var rootHash common.Hash
    wg := sync.WaitGroup{}
    wg.Add(2)
    go func() {
        rootHash = state.IntermediateRoot(chain.Config().IsEIP158(header.Number))
        wg.Done()
    }()
    go func() {
        blk = types.NewBlock(header, *transactOpts.Txs, nil, *transactOpts.Receipts, trie.NewStackTrie(nil))
        wg.Done()
    }()
    wg.Wait()
    blk.SetRoot(rootHash)
    // Assemble and return the final block for sealing
    return blk, *transactOpts.Receipts, nil
}
```

Figure 46 Source code of the *FinalizeAndAssemble* function

(2) Contract Asset Security

The Ronin supports EVM in order to be compatible with Ether, and the contracts on it are not upgradable or modifiable.

4 Consensus Security

To increase the degree of decentralization of Ronin, Ronin uses Proof of Stake Authority (PoSA) instead of PoA. Proof of Stake Authority is a combination of delegated Proof of Stake (DPoS) and PoA, specifically divided into two parts:

1. Token holders use their shares to vote and elect a set of validators (DPoS main logic).
2. Validators take turns to produce blocks in a PoA manner, similar to Ethereum's Clique consensus design.

In DoPS, all token holders can register as validator candidates. They can also act as delegators by staking tokens to validator candidates. Validator and delegator stakes are updated at the start of each day. Afterwards, the Ronin network will select a group of 21 validators (including 11 trusted organizations and 10 most staked token holders). However, during the day, some validators may be (temporarily) removed from the validator set (for punishment, the part is maintained by the Slash contract, they may be in jail or in maintenance mode). These changes will be updated every epoch (consisting of 200 blocks ~ 10 minutes).

Furthermore, while increasing the decentralization of the system, the validator selection process through staking introduces new attack vectors. An attacker who controls more than 51% of the coins can take over the blockchain.

To prevent such attacks, the Ronin Network relies on a group of 11 trusted organizations selected by the community and Sky Mavis. Since trusted organizations occupy 11/21 of the validator set, an attacker cannot control the majority of validators and take over the blockchain.

In PoA, one of the sorted validator sets (updated once per epoch) will be responsible for packaging a certain block, and the block will not include block rewards (the rewards are allocated by the consensus contract).

In summary, the logical framework of Ronin consensus processing can refer to the following figure:

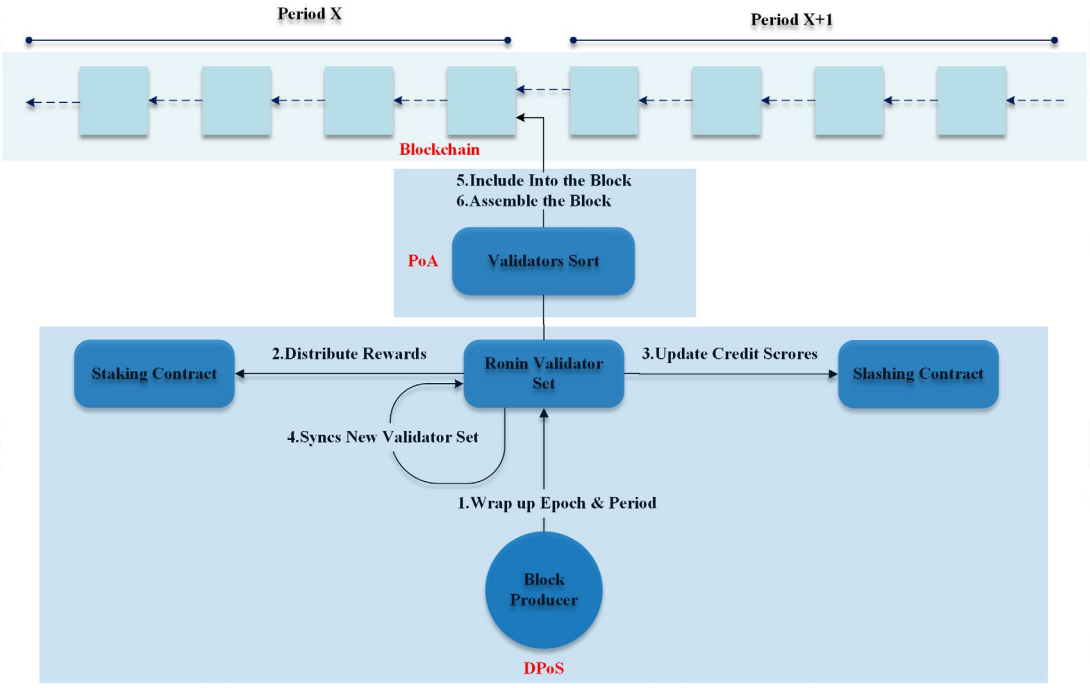


Figure 47 Consensus Framework

4.1 DPoS Consensus Process Analysis

The DPoS algorithm is divided into two parts: electing a group of block producers and scheduling production. The election process makes sure that stakeholders are ultimately in control because stakeholders lose the most when the network does not operate smoothly. How people are elected has little impact on how consensus is achieved on a minute by minute basis.

To help explain this algorithm, assume 3 block producers, A, B, and C. Because consensus requires 2/3 + 1 to resolve all cases, this simplified model will assume that producer C is deemed the tie breaker. In the real world there would be 21 or more block producers. Like proof of work, the general rule is that longest chain wins. Any time an honest peer sees a valid strictly longer chain it will switch from its current fork to the longer one.

- **Normal Operation**

Under normal operation block producers take turns producing a block every 3 seconds. Assuming no one misses their turn then this will produce the longest possible chain. It is invalid for a block producer to produce a block at any other time slot than the one they are scheduled for.

4.2 PoA Consensus Process Analysis

In the PoA consensus preparation phase, the Ronin Consortium V2 consensus engine will obtain the set of block validators of the current epoch from the consensus contract (DPoS consensus phase) at the beginning of each epoch, and sort them according to the size of the set addresses. Then convert the sorted validator set into bytecode and fill it into the extra field of the block header, so that all Ronin nodes can read the validator set and package the verification block sequence. In each epoch, there are multiple periods of packaging blocks, and each cycle is trained by a set of validators to produce blocks. When there is a block timeout, repeating the current block generation and nodes submitting malicious system transactions, the validators will be punished (see the Slashing Logic Implementation chapter for details).

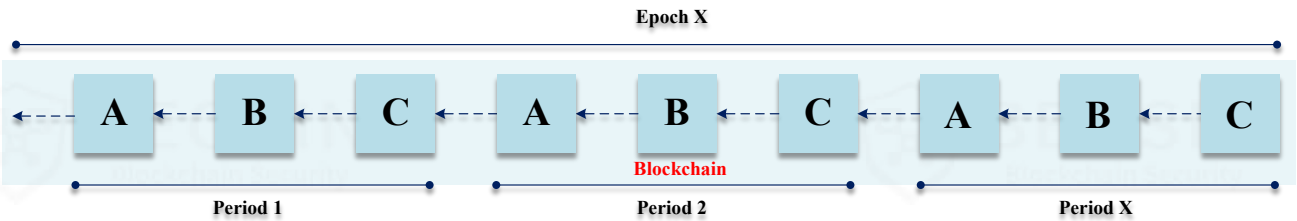


Figure 48 Pack blocks in an epoch

4.3 Logic Implementation of DPoS

The DPoS consensus contract engine in Ronin is realized through the system contract and the precompiled contract. The system contract is used to manage validators and candidates; the precompiled contract is used to sort the candidates of the system contract and select effective validators.

After Ronin runs, if the current block is the last block before the end of this epoch, its block builder will call the wrapUpEpoch function of the system contract validatorSet to wrap the relevant data of the epoch. This operation will check the status of all validators in this period, and update the valid validators for the next epoch (by means of permission replacement). If the current block is still the last block before the end of this period, the wrap content also includes:

1. Settle the bridge operating reward of each validator.
2. Submit bridge operating reward and mining reward.
3. Update the delegating rewards and send the rewards to the system contract staking.
4. recycle the locked funds from emergency exit requests.
5. Disposal of discarded rewards.
6. Update the credit scores of each validator.
7. Update the validators set.

Among them, when updating the validator list, the system contract validatorSet will send the current candidate list, its corresponding pledge amount and trusted weight as parameters to the precompile contract Precompile Pick Validator Set for sorting; in the precompile contract Precompile Pick Validator Set , will first sort these candidates according to their pledge amount, and then directly add the candidates whose trusted weight is greater than 0 to the front of the sorted candidate list to determine the valid validator for this period; finally, return the sorted validators to the system contract The wrapUpEpoch function updates records.

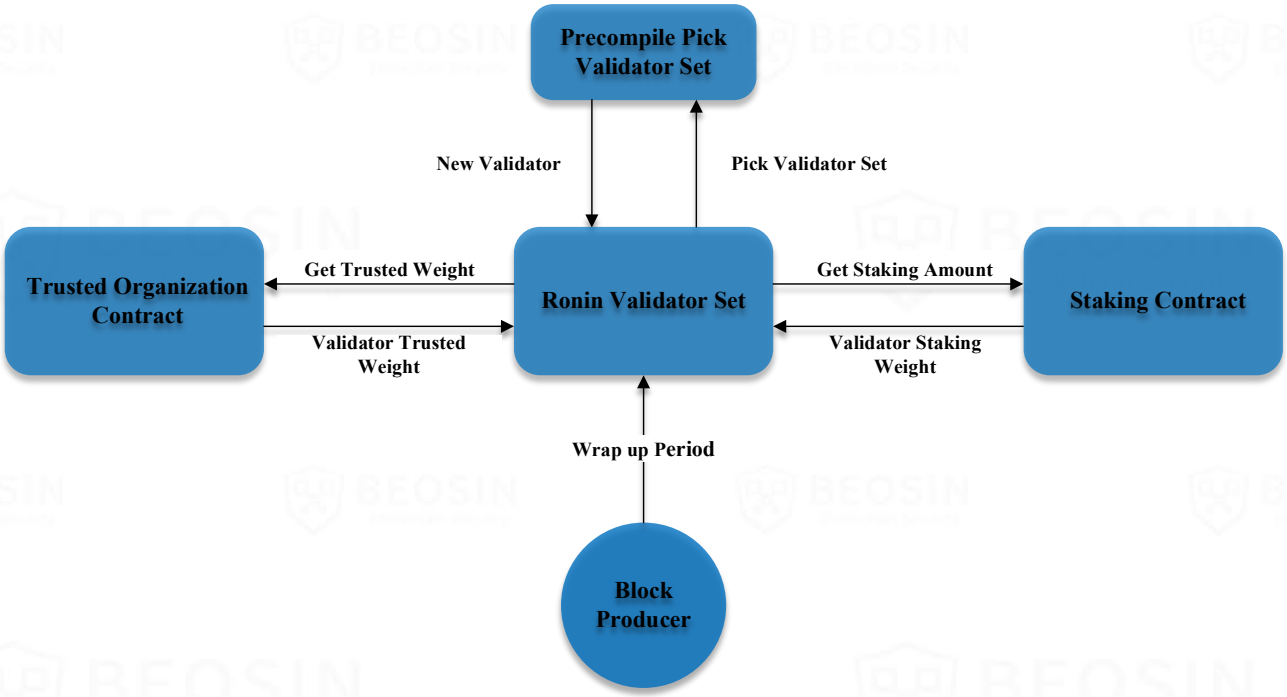


Figure 49 DPoS

4.4 Logic Implementation of PoA

The Ronin PoA consensus logic is mainly divided into three parts:

1. Initialize the consensus engine
2. Complete system transactions and assemble blocks.
3. Verify that other nodes produce blocks legally.

During the start-up phase of the Consortium V2 consensus engine, the Ronin node will call the *Prepare* method to prepare all consensus fields in the block header so that transactions can be run on top. The *Prepare* method first initializes the consensus contract, and then queries the validator nodes and fills its order into the block header when it is judged that each Epoch starts.

```go
func (c *Consortium) Prepare(chain consensus.ChainHeaderReader, header *types.Header) error {
    if err := c.initContract(); err != nil {
        return err
    }

    header.Coinbase = c.val
    header.Nonce = types.BlockNonce{}

    number := header.Number.Uint64()
    snap, err := c.snapshot(chain, number-1, header.ParentHash, nil)
    if err != nil {
        return err
    }

    // Set the correct difficulty
    header.Difficulty = CalcDifficulty(snap, c.val)

    // Ensure the extra data has all it's components
    if len(header.Extra) < extraVanity {
        header.Extra = append(header.Extra, bytes.Repeat([]byte{0x00}, extraVanity-len(header.Extra))...)
    }
    header.Extra = header.Extra[:extraVanity]

    if number%c.config.EpochV2 == 0 || c.chainConfig.IsOnConsortiumV2(big.NewInt(int64(number))) {
        // This block is not inserted, the transactions in this block are not applied, so we need
        // the call GetValidators at the context of previous block
        newValidators, err := c.contract.GetValidators(new(big.Int).Sub(header.Number, common.Big1))
        if err != nil {
            return err
        }
        // Sort validators by address
        sort.Sort(validatorsAscending(newValidators))
        for _, validator := range newValidators {
            header.Extra = append(header.Extra, validator.Bytes()...)
        }
    }

    // Add extra seal space
    header.Extra = append(header.Extra, make([]byte, extraSeal)...)

    // Mix digest is reserved for now, set to empty
    header.MixDigest = common.Hash{}

    // Ensure the timestamp has the correct delay
    parent := chain.GetHeader(header.ParentHash, number-1)
    if parent == nil {
        return consensus.ErrUnknownAncestor
    }

    header.Time = c.computeHeaderTime(header, parent, snap)
    return nil
}
```

Figure 50 *Prepare* method

The Ronin node will start the miner network and call the FinalizeAndAssemble method to execute the system transaction, then determine whether the uncle block exists when the consensus is running to generate blocks. The processSystemTransactions method is called to process the system transaction, it will check whether the previous verifier has completed the operation of packing the block. If not, the Slash consensus contract will be called to punish, and then the SubmitBlockReward function of the consensus contract will be called to submit the transaction fee as a reward. Finally, at the end of each Epoch, the WrapUpEpoch method of the consensus contract is called to update the validator set.

```go
func (c *Consortium) FinalizeAndAssemble(chain consensus.ChainHeaderReader, header *types.Header, state *state.StateDB,
    txs []*types.Transaction, uncles []*types.Header, receipts []*types.Receipt) (*types.Block, []*types.Receipt, error) {
    if err := c.initContract(); err != nil {
        return nil, nil, err
    }

    // No block rewards in PoA, so the state remains as is and uncles are dropped
    if txs == nil {
        txs = make([]*types.Transaction, 0)
    }
    if receipts == nil {
        receipts = make([]*types.Receipt, 0)
    }
    evmContext := core.NewEVMBlockContext(header, consortiumCommon.ChainContext{Chain: chain, Consortium: c}, &header.Coinbase, chain.OpEvents()...)
    transactOpts := &consortiumCommon.ApplyTransactOpts{
        ApplyMessageOpts: &consortiumCommon.ApplyMessageOpts{
            State:       state,
            Header:      header,
            ChainConfig: c.chainConfig,
            EVMContext:  &evmContext,
        },
        Txs:      &txs,
        Receipts: &receipts,
        // a.k.a. System Txs
        // It always equals nil since FinalizeAndAssemble doesn't receive any transactions
        ReceivedTxs: nil,
        UsedGas:     &header.GasUsed,
        Mining:      true,
        Signer:      c.signer,
        SignTxFn:    c.signTxFn,
    }

    if err := c.processSystemTransactions(chain, header, transactOpts, true); err != nil {
        return nil, nil, err
    }

    // should not happen. Once happen, stop the node is better than broadcast the block
    if header.GasLimit < header.GasUsed {
        return nil, nil, errors.New("gas consumption of system txs exceed the gas limit")
    }
    header.UncleHash = types.CalcUncleHash(nil)
    var blk *types.Block
    var rootHash common.Hash
    wg := sync.WaitGroup{}
    wg.Add(2)
    go func() {
        rootHash = state.IntermediateRoot(chain.Config().IsEIP158(header.Number))
        wg.Done()
    }()
    go func() {
        blk = types.NewBlock(header, *transactOpts.Txs, nil, *transactOpts.Receipts, trie.NewStackTrie(nil))
        wg.Done()
    }()
    wg.Wait()
    blk.SetRoot(rootHash)
    // Assemble and return the final block for sealing
    return blk, *transactOpts.Receipts, nil
}

// Authorize injects a private key into the consensus engine to mint new blocks with
```

Figure 51 FinalizeAndAssemble method

```go
func (c *Consortium) processSystemTransactions(chain consensus.ChainHeaderReader, header *types.Header,
    transactOpts *consortiumCommon.ApplyTransactOpts, isFinalizeAndAssemble bool) error {

    if header.Difficulty.Cmp(diffInTurn) != 0 {
        number := header.Number.Uint64()
        snap, err := c.snapshot(chain, number-1, header.ParentHash, nil)
        if err != nil {
            return err
        }
        spoiledVal := snap.supposeValidator()
        signedRecently := false
        for _, recent := range snap.Recents {
            if recent == spoiledVal {
                signedRecently = true
                break
            }
        }
        if !signedRecently {
            if !isFinalizeAndAssemble {
                log.Info("Slash validator", "number", header.Number, "spoiled", spoiledVal)
            }
            if err := c.contract.Slash(transactOpts, spoiledVal); err != nil {
                // it is possible that slash validator failed because of the slash channel is disabled.
                log.Error("Failed to slash validator", "block hash", header.Hash(), "address", spoiledVal)
                return err
            }
        }
    }

    // Previously, we call WrapUpEpoch before SubmitBlockReward which is the wrong order.
    // We create a hardfork here to fix the contract call order.
    if c.chainConfig.IsPuffy(header.Number) {
        if err := c.submitBlockReward(transactOpts); err != nil {
            return err
        }
    }

    if header.Number.Uint64()%c.config.EpochV2 == c.config.EpochV2-1 {
        if err := c.contract.WrapUpEpoch(transactOpts); err != nil {
            log.Error("Failed to wrap up epoch", "err", err)
            return err
        }
    }

    if !c.chainConfig.IsPuffy(header.Number) {
        return c.submitBlockReward(transactOpts)
    }

    return nil
}
```

Figure 52 processSystemTransactions method

In the verification phase, other nodes will call the Finalize method to repeatedly verify whether the system transaction is legal, and finally insert the block into the database.

```go
func (c *Consortium) Finalize(chain consensus.ChainHeaderReader, header *types.Header, state *state.StateDB, txs *[]*types.Transaction,
    uncles []*types.Header, receipts *[]*types.Receipt, systemTxs *[]*types.Transaction, internalTxs *[]*types.InternalTransaction, usedGas *uint64) error {
    if err := c.initContract(); err != nil {
        return err
    }
    evmContext := core.NewEVMBlockContext(header, consortiumCommon.ChainContext{Chain: chain, Consortium: c}, &header.Coinbase, chain.OpEvents()...)
    transactOpts := &consortiumCommon.ApplyTransactOpts{
        ApplyMessageOpts: &consortiumCommon.ApplyMessageOpts{
            State:       state,
            Header:      header,
            ChainConfig: c.chainConfig,
            EVMContext:  &evmContext,
        },
        Txs:      txs,
        Receipts: receipts,
        // a.k.a. System Txs
        // systemTxs is received from other nodes
        ReceivedTxs: systemTxs,
        UsedGas:     usedGas,
        Mining:      false,
        Signer:      c.signer,
        SignTxFn:    c.signTxFn,
        EthAPI:      c.ethAPI,
    }

    // If the block is an epoch end block, verify the validator list
    // The verification can only be done when the state is ready, it can't be done in VerifyHeader.
    if header.Number.Uint64()%c.config.EpochV2 == 0 {
        // The GetValidators in Prepare is called on the context of previous block so here it must
        // be called on context of previous block too
        newValidators, err := c.contract.GetValidators(new(big.Int).Sub(header.Number, common.Big1))
        if err != nil {
            return err
        }
        // sort validator by address
        sort.Sort(validatorsAscending(newValidators))
        validatorsBytes := make([]byte, len(newValidators)*validatorBytesLength)
        for i, validator := range newValidators {
            copy(validatorsBytes[i*validatorBytesLength:], validator.Bytes())
        }

        extraSuffix := len(header.Extra) - extraSeal
        if !bytes.Equal(header.Extra[extraVanity:extraSuffix], validatorsBytes) {
            return errMismatchingEpochValidators
        }
    }

    if err := c.processSystemTransactions(chain, header, transactOpts, false); err != nil {
        return err
    }
    if len(*transactOpts.EVMContext.InternalTransactions) > 0 {
        *internalTxs = append(*internalTxs, *transactOpts.EVMContext.InternalTransactions...)
    }
    if len(*systemTxs) > 0 {
        return errors.New("the length of systemTxs do not match")
    }
    return nil
}
```

Figure 53 *Finalize* method

4.4 Rewards for Building Blocks

In Ronin, miner reward is divided into main chain miner reward and side chain miner reward. First, the algorithm calculates the minerReward of each block through the Period and initSignerBlockReward set in the config, then calculates the miner reward of the side chain and uses the minerReward to issue it. The remaining minerReward will then pay the Gas-related Refund. Finally, all the remaining rewards are distributed to the miners corresponding to the blocks.

```
func accumulateRewards(config *params.ChainConfig, state *state.StateDB, header *types.Header, snap *Snapshot, refundGas RefundGas) error {
    // Calculate the block reword by year
    blockNumPerYear := secondsPerYear / config.Alien.Period
    initSignerBlockReward := new(big.Int).Div(totalBlockReward, big.NewInt(int64(2*blockNumPerYear)))
    yearCount := header.Number.Uint64() / blockNumPerYear
    blockReward := new(big.Int).Rsh(initSignerBlockReward, uint(yearCount))

    minerReward := new(big.Int).Set(blockReward)
    minerReward.Mul(minerReward, new(big.Int).SetUint64(snap.MinerReward))
    minerReward.Div(minerReward, big.NewInt(1000)) // cause the reward is calculate by cnt per thousand

    votersReward := blockReward.Sub(blockReward, minerReward)
```

Figure 54 Calculate miner rewards

```
scReward, minerLeft := snap.calculateSCReward(minerReward)
minerReward.Set(minerLeft)
// rewards for the side chain coinbase
for scCoinbase, reward := range scReward {
    state.AddBalance(scCoinbase, reward)
}
// refund gas for custom txs
for sender, gas := range refundGas {
    state.AddBalance(sender, gas)
    minerReward.Sub(minerReward, gas)
}

// rewards for the miner, check minerReward value for refund gas
if minerReward.Cmp(big.NewInt(0)) > 0 {
    state.AddBalance(header.Coinbase, minerReward)
}
```

Figure 55 Calculate left miner rewards

4.4 Slashing Logic Implementation

In the DPoS system, in order to make validators more actively participate in the system and decentralize to a certain extent, when validators do not provide good services for the Ronin network or have malicious and negative behaviors, they will be cut. And in the slash contract, a reduction method for validators and bridge operators is designed, mainly for punishment in terms of rewards and jail time. The following figure shows the slash interaction process:

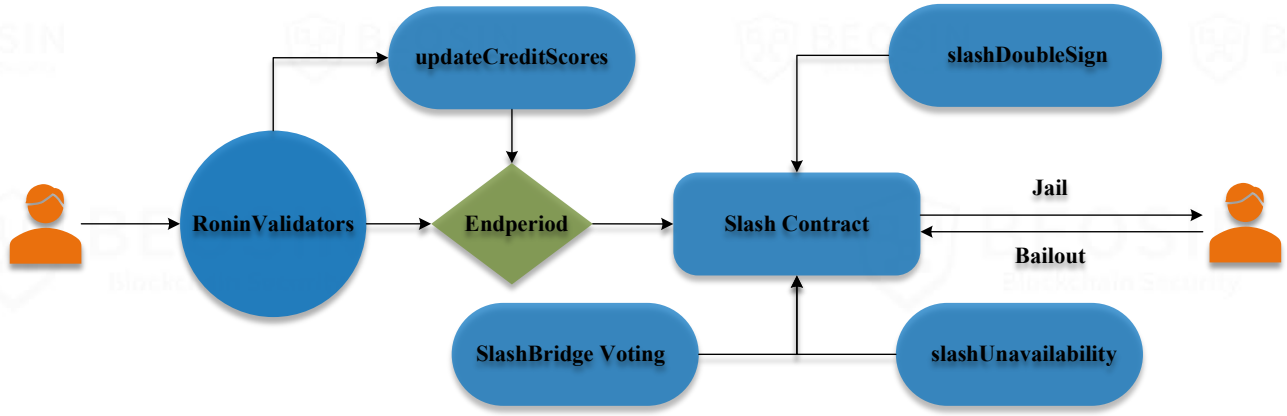


Figure 56 Slashing Logic

As shown in the figure above, at the end of the period, the RoninValidatorSet.wrapUpEpoch() function will update the credit score and judge the maintenance status, and also interact with the slash contract. If it is in jail, it will not be able to become a validator, which will affect the new The setting of the Validator set and the distribution of rewards.

In terms of bridge operators, when the period ends, the *_updateValidatorRewardBaseOnBridgeOperatingPerformance* function will update the proportion of votes. If the bridge operator does not provide enough signatures, then two penalties (slash) will be imposed according to the number of votes at fault of the bridge operator:

1. If the number of votes obtained by bridge operators is less than the _ratioTier1 (10% on the white paper) threshold, the bridge rewards for the current period of the operators will be cut.

2. If the number of votes obtained by bridge operators is less than the threshold of _ratioTier2 (20% on the white paper), then (s)he's bridge rewards and mining rewards will be cut and the jail time will be increased (2 days on the white paper), while increasing the same amount of time so that the verifier cannot bail himself out by deducting the credit score through the BailOut function, so as to encourage the verifier to get enough votes and actively participate.

```
191     uint256 _votedRatio = (_validatorBallots * _MAX_PERCENTAGE) / _totalVotes;
192     uint256 _missedRatio = _MAX_PERCENTAGE - _votedRatio;
193     if (_missedRatio > _ratioTier2) {
194       _bridgeRewardDeprecatedAtPeriod[_validator][_period] = true;
195       _miningRewardDeprecatedAtPeriod[_validator][_period] = true;
196
197       // Cannot saving gas by temp variable here due to too deep stack.
198       _blockProducerJailedBlock[_validator] = Math.max(
199         block.number + _jailDurationTier2,
200         _blockProducerJailedBlock[_validator]
201       );
202       _cannotBailoutUntilBlock[_validator] = Math.max(
203         block.number + _jailDurationTier2,
204         _cannotBailoutUntilBlock[_validator]
205       );
206
207       emit ValidatorPunished(_validator, _period, _blockProducerJailedBlock[_validator], 0, true, true);
208     } else if (_missedRatio > _ratioTier1) {
209       _bridgeRewardDeprecatedAtPeriod[_validator][_period] = true;
210       emit ValidatorPunished(_validator, _period, _blockProducerJailedBlock[_validator], 0, false, true);
211     } else if (_totalBallots > 0) {
212       _bridgeOperatingReward[_validator] = (_totalBridgeReward * _validatorBallots) / _totalBallots;
213     }
```

Figure 57 *_updateValidatorRewardBaseOnBridgeOperatingPerformance* function

Since the PoA consensus is used for block generation, it is quite a serious error when a validator signs more than one block with the same height. After verifying the double-signed malicious behavior, in the slashDoubleSign function, the malicious violating verifier will be directly punished with an amount of _slashDoubleSignAmount of funds, and the jail time of _doubleSigningJailUntilBlock will be increased. The current punishment is considered to be released on bail of.

```
24  function slashDoubleSign(
25    address _consensuAddr,
26    bytes calldata _header1,
27    bytes calldata _header2
28  ) external override {
29    require(msg.sender == block.coinbase, "SlashIndicator: method caller must be coinbase");
30    if (!_shouldSlash(_consensuAddr)) {
31      return;
32    }
33
34    if (_pcValidateEvidence(_header1, _header2)) {
35      uint256 _period = _validatorContract.currentPeriod();
36      emit Slashed(_consensuAddr, SlashType.DOUBLE_SIGNING, _period);
37      _validatorContract.execSlash(_consensuAddr, _doubleSigningJailUntilBlock, _slashDoubleSignAmount, false);
38    }
39  }
```

Figure 58 slashDoubleSign function (unfixed)

In terms of block generation, the performance of Ronin nodes depends on everyone in the validator set producing blocks in time when it is their turn. If validators miss the opportunity to create blocks, it will damage the performance of the system. Therefore, Ronin introduces an unavailable slashing logic to penalize validators who miss too many blocks.

In the function slashUnavailability, the number of blocks _count missed by the verifier will be recorded. When _count reaches _unavailabilityTier1Threshol(50 on the white paper) and triggers the threshold, the rewards of the current period will be penalized. If it reaches _count_unavailabilityTier2Threshold(150 on the white paper paper) and the threshold is triggered, the rewards of the current cycle will be confiscated, the amount of _slashAmountForUnavailabilityTier2Threshold (10,000 RON on the white paper) will be deducted, and the jail time of _jailDurationForUnavailabilityTier2Threshold (2 days on the white paper) will be increased. If a verifier triggers Tier1 in the current cycle and uses credit score to bail himself out, triggers Tier1 again, then the verifier will be penalized at Tier2 level and cannot use credit score to bail.

```
function slashUnavailability(address _validatorAddr) external override oncePerBlock {
  require(msg.sender == block.coinbase, "SlashUnavailability: method caller must be coinbase");
  if (!_shouldSlash(_validatorAddr)) {
    return;
  }

  uint256 _period = _validatorContract.currentPeriod();
  uint256 _count = ++_unavailabilityIndicator[_validatorAddr][_period];

  if (_count == _unavailabilityTier2Threshold) {
    emit Slashed(_validatorAddr, SlashType.UNAVAILABILITY_TIER_2, _period);
    _validatorContract.execSlash(
      _validatorAddr,
      block.number + _jailDurationForUnavailabilityTier2Threshold,
      _slashAmountForUnavailabilityTier2Threshold,
      false
    );
  } else if (_count == _unavailabilityTier1Threshold) {
    bool _tier1SecondTime = checkBailedOutAtPeriod(_validatorAddr, _period);
    if (!_tier1SecondTime) {
      emit Slashed(_validatorAddr, SlashType.UNAVAILABILITY_TIER_1, _period);
      _validatorContract.execSlash(_validatorAddr, 0, 0, false);
    } else {
      /// Handles tier-3
      emit Slashed(_validatorAddr, SlashType.UNAVAILABILITY_TIER_3, _period);
      uint256 _jailedUntilBlock = block.number + _jailDurationForUnavailabilityTier2Threshold;
      _validatorContract.execSlash(
        _validatorAddr,
        _jailedUntilBlock,
        _slashAmountForUnavailabilityTier2Threshold,
        true
      );
    }
  }
}
```

Then the trusted organization has not voted for the bridge operator for more than _bridgeVotingThreshold(3 days on the white paper) time threshold, then the slashBridgeVoting function will be called to confiscate the corresponding node number is _bridgeVotingSlashAmount(10,000 RON on the white paper) funds.

```
function slashBridgeVoting(address _consensusAddr) external {
  IRoninTrustedOrganization.TrustedOrganization memory _org = _roninTrustedOrganizationContract
    .getTrustedOrganization(_consensusAddr);
  uint256 _lastVotedBlock = Math.max(_roninGovernanceAdminContract.lastVotedBlock(_org.bridgeVoter), _org.addedBlock);
  uint256 _period = _validatorContract.currentPeriod();
  if (block.number - _lastVotedBlock > _bridgeVotingThreshold && !_bridgeVotingSlashed[_consensusAddr][_period]) {
    _bridgeVotingSlashed[_consensusAddr][_period] = true;
    emit Slashed(_consensusAddr, SlashType.BRIDGE_VOTING, _period);
    _validatorContract.execSlash(_consensusAddr, 0, _bridgeVotingSlashAmount, false);
  }
}
```

When updating a candidate node, if the candidate has not become a validator node, then he will reset the credit score in the execResetCreditScores function after the update.

```
      _slashIndicatorContract.updateCreditScores(_currentValidators, _lastPeriod);
      (_currentValidators, _revokedCandidates) = _syncValidatorSet(_newPeriod);
      if (_revokedCandidates.length > 0) {
        _slashIndicatorContract.execResetCreditScores(_revokedCandidates);
      }
    }
```

In the bailOut function, each validator can hold a maximum credit score of maxCreditScore. The credit score spent on bail is multiplied by the remaining jailed time with bailOutCostMultiplier as the basic unit. The less the remaining jailed time, the less credits will be spent.

```
77  function bailOut(address _consensusAddr) external override {
78    require(
79      _validatorContract.isValidatorCandidate(_consensusAddr),
80      "SlashIndicator: consensus address must be a validator candidate"
81    );
82    require(
83      _validatorContract.isCandidateAdmin(_consensusAddr, msg.sender),
84      "SlashIndicator: method caller must be a candidate admin"
85    );
86
87    (bool _isJailed, , uint256 _jailedEpochLeft) = _validatorContract.getJailedTimeLeft(_consensusAddr);
88    require(_isJailed, "SlashIndicator: caller must be jailed in the current period");
89
90    uint256 _period = _validatorContract.currentPeriod();
91    require(!_checkBailedOutAtPeriod[_consensusAddr][_period], "SlashIndicator: validator has bailed out previously");
92
93    uint256 _score = _creditScore[_consensusAddr];
94    uint256 _cost = _jailedEpochLeft * _bailOutCostMultiplier;
95    require(_score >= _cost, "SlashIndicator: insufficient credit score to bail out");
96
97    _validatorContract.execBailOut(_consensusAddr, _period);
98
99    _creditScore[_consensusAddr] -= _cost;
100   _setUnavailabilityIndicator(_consensusAddr, _period, 0);
101   _checkBailedOutAtPeriod[_consensusAddr][_period] = true;
102 }
```

Figure 59 bailOut function

The gainCreditScores function is used to update the score, when the validator is in jail or is executing the maintenance plan, the score will not be awarded.

5 Transaction Model Security

5.1 Transaction Processing Flow

Ronin's transaction processing is divided into two main parts: adding the transaction to the pool and executing the transaction. Adding a transaction to the pool means that each node adds the submitted transaction to the pool. As shown in the figure below, Ronin will perform the following checks on the transaction.

- Data size must be < 128KB
- Transaction amount must be non-negative (>=0)
- Ensure the transaction doesn't exceed the current block limit gas
- The signature data must be valid and the sender address can be resolved
- Gas price required to be transaction is not greater than the Gas price of the pool
- The nonce value of the transaction must be higher than the nonce value of the account on the current chain (lower than that means the transaction has been packaged)
- Current account balance must be greater than "Transaction Amount"
- Trading GAS requirements cannot be less than the specified quantity

```go
// validateTx checks whether a transaction is valid according to the consensus
// rules and adheres to some heuristic limits of the local node (price and size).
func (pool *TxPool) validateTx(tx *types.Transaction, local bool) error {
    // Accept only legacy transactions until EIP-2718/2930 activates.
    if !pool.eip2718 && tx.Type() != types.LegacyTxType {
        return ErrTxTypeNotSupported
    }
    // Reject dynamic fee transactions until EIP-1559 activates.
    if !pool.eip1559 && tx.Type() == types.DynamicFeeTxType {
        return ErrTxTypeNotSupported
    }
    // Reject transactions over defined size to prevent DOS attacks
    if uint64(tx.Size()) > txMaxSize {
        return ErrOversizedData
    }
    // Transactions can't be negative. This may never happen using RLP decoded
    // transactions but may occur if you create a transaction using the RPC.
    if tx.Value().Sign() < 0 {
        return ErrNegativeValue
    }
    // Ensure the transaction doesn't exceed the current block limit gas.
    if pool.currentMaxGas < tx.Gas() {
        return ErrGasLimit
    }
    // Sanity check for extremely large numbers
    if tx.GasFeeCap().BitLen() > 256 {
        return ErrFeeCapVeryHigh
    }
    if tx.GasTipCap().BitLen() > 256 {
        return ErrTipVeryHigh
    }
    // Ensure gasFeeCap is greater than or equal to gasTipCap.
    if tx.GasFeeCapIntCmp(tx.GasTipCap()) < 0 {
        return ErrTipAboveFeeCap
    }
    // Make sure the transaction is signed properly.
    from, err := types.Sender(pool.signer, tx)
    if err != nil {
        return ErrInvalidSender
    }
    // Drop non-local transactions under our own minimal accepted gas price or tip.
    pendingBaseFee := pool.priced.urgent.baseFee
    if !local && tx.EffectiveGasTipIntCmp(pool.gasPrice, pendingBaseFee) < 0 {
        return ErrUnderpriced
    }
    // Ensure the transaction adheres to nonce ordering
    if pool.currentState.GetNonce(from) > tx.Nonce() {
        return ErrNonceTooLow
    }
    // Transactor should have enough funds to cover the costs
    // cost == V + GP * GL
    if pool.currentState.GetBalance(from).Cmp(tx.Cost()) < 0 {
        return ErrInsufficientFunds
    }
    // Ensure the transaction has more gas than the basic tx fee.
    intrGas, err := IntrinsicGas(tx.Data(), tx.AccessList(), tx.To() == nil, true, pool.istanbul)
    if err != nil {
        return err
    }
    if tx.Gas() < intrGas {
        return ErrIntrinsicGas
    }

    // Contract creation transaction
    if tx.To() == nil && pool.chainconfig.Consortium != nil {
        whitelisted := state.IsWhitelistedDeployer(pool.currentState, from)
        if !whitelisted {
            return ErrUnauthorizedDeployer
        }
    }

    // Check if sender and recipient are blacklisted
    if pool.chainconfig.Consortium != nil && pool.odysseus {
        contractAddr := pool.chainconfig.BlacklistContractAddress
        if state.IsAddressBlacklisted(pool.currentState, contractAddr, &from) || state.IsAddressBlacklisted(pool.currentState, contractAddr, tx.To()) {
            return ErrAddressBlacklisted
        }
    }

    return nil
}
```

Figure 60 Partial source code of transaction check

5.2 Transaction replay audit

Replay attack refers to when the blockchain is hard-forked, the blockchain has a permanent difference and produces two historical transactions, on the exact corresponding chains, with differences such as address, private key, and balance, in which the same transaction can take effect at the same time on the two chains.

After testing, the trades on the Ronin chain (including EVM transactions) cannot be replayed, and during the transaction when adding to transaction pool, validity is verified, while illegal transactions are rejected because the transaction data format does not match; Chains with substrate frameworks can also fail transactions because of differences in chain ids. Therefore, it is recommended that the project developer pay close attention to chain id to avoid duplicate ids for the same type of chain.

5.3 Dusting Attack

The dusting attack is when an attacker sends a very small amount of tokens, called "dust", to a user's wallet. By tracking the dusted wallet funds and all transactions, the attacker can then connect to these addresses and determine the company or person to whom these wallet addresses belong, destroying the anonymity of the block chain; or misuse the block chain resources, causing the block chain memory pool strain. If the dust money is not moved, the attacker cannot establish a connection to it and cannot complete the de-anonymization of the wallet or address owners.

The Ronin chain is based on the account model, so dust attacks are not considered.

5.4 Trading Flooding Attacks

In Ronin, transactions are subject to a fee. The base fee for creating a contract is 53,000, and the base fee for a normal transaction is 21,000.

```
const (
    GasLimitBoundDivisor          uint64 = 1024 // The bound divisor of the gas limit, used in update calculations.
    ConsortiumGasLimitBoundDivisor uint64 = 256
    MinGasLimit                   uint64 = 5000    // Minimum the gas limit may ever be.
    GenesisGasLimit               uint64 = 4712388 // Gas limit of the Genesis block.

    MaximumExtraDataSize  uint64 = 32    // Maximum size extra data may be after Genesis.
    ExpByteGas            uint64 = 10    // Times ceil(log256(exponent)) for the EXP instruction.
    SloadGas              uint64 = 50    // Multiplied by the number of 32-byte words that are copied (round up) for any *COPY operation and added.
    CallValueTransferGas  uint64 = 9000  // Paid for CALL when the value transfer is non-zero.
    CallNewAccountGas     uint64 = 25000 // Paid for CALL when the destination address didn't exist prior.
    TxGas                 uint64 = 21000 // Per transaction not creating a contract. NOTE: Not payable on data of calls between transactions.
    TxGasContractCreation uint64 = 53000 // Per transaction that creates a contract. NOTE: Not payable on data of calls between transactions.
    TxDataZeroGas         uint64 = 4     // Per byte of data attached to a transaction that equals zero. NOTE: Not payable on data of calls between
    QuadCoeffDiv          uint64 = 512   // Divisor for the quadratic particle of the memory cost equation.
    LogDataGas            uint64 = 8     // Per byte in a LOG* operation's data.
    CallStipend           uint64 = 2300  // Free gas given at beginning of call.
```

Figure 61 Transactions minimum gas consumption

However, it is worth mentioning that gasprice is set by the super node and if it is 0, it will lead to trading flooding attacks.

```
PS D:\work\Project\ronin\gonglian\ronin-master\ronin-master\cmd\ronin\ronin_windows> ./roninWindows.exe --http.api eth,net,web3,consortium --networkid 2022 --bootnodes enode://cddba93323aa548ff232ac14aea5104ecbf7544d43ecb9b1e76284985afface13cbd8cf71e7de7f6d72c635803e063f8af955e22ae0abd575c2b1b87f117194d@127.0.0.1:30310 --datadir ./node1 --port 30316 --http --http.corsdomain '*' --http.addr 0.0.0.0 --http.port 8555 --http.vhosts '*' --ws --ws.addr 0.0.0.0 --ws.port 8556 --ws.origins '*'  --unlock 35668CFeE2D8ba9456af175fe30acb39143e384A --password pwd.txt --mine --allow-insecure-unlock --ipcdisable --miner.gasprice 1
```

Figure 62 Screenshot of the node start command

5.5 Double-spending Attack

For Ronin's transactions, each transaction of the account contains a unique and mintable nonce, and Ronin uses the consensus model of PoSA, it is also difficult to complete a double-spending through a 51% attack.

5.6 Illegal Transactions

The user will sign the entire transaction data when initiating a transaction, and any changes to the data in the transaction will cause the Ronin node to fail the signature check.

```go
func (c *Consortium) verifySeal(chain consensus.ChainHeaderReader, header *types.Header, parents []*types.Header, snap *Snapshot) error {
    // Verifying the genesis block is not supported
    number := header.Number.Uint64()
    if number == 0 {
        return consortiumCommon.ErrUnknownBlock
    }

    // Resolve the authorization key and check against validators
    signer, err := ecrecover(header, c.signatures, c.chainConfig.ChainID)
    if err != nil {
        return err
    }

    if signer != header.Coinbase {
        return errCoinBaseMisMatch
    }

    if _, ok := snap.Validators[signer]; !ok {
        return errUnauthorizedValidator
    }

    for seen, recent := range snap.Recents {
        if recent == signer {
            // Signer is among recents, only fail if the current block doesn't shift it out
            if limit := uint64(len(snap.Validators)/2 + 1); seen > number-limit {
                return consortiumCommon.ErrRecentlySigned
            }
        }
    }

    // Ensure that the difficulty corresponds to the turn-ness of the signer
    if !c.fakeDiff {
        inturn := snap.inturn(signer)
        if inturn && header.Difficulty.Cmp(diffInTurn) != 0 {
            return consortiumCommon.ErrWrongDifficulty
        }
        if !inturn && header.Difficulty.Cmp(diffNoTurn) != 0 {
            return consortiumCommon.ErrWrongDifficulty
        }
    }

    return nil
}
```

Figure 63 Transaction signature check(1/2)

```go
// ecrecover extracts the Ronin account address from a signed header.
func ecrecover(header *types.Header, sigcache *lru.ARCCache, chainId *big.Int) (common.Address, error) {
    // If the signature's already cached, return that
    hash := header.Hash()
    if address, known := sigcache.Get(hash); known {
        return address.(common.Address), nil
    }
    // Retrieve the signature from the header extra-data
    if len(header.Extra) < consortiumCommon.ExtraSeal {
        return common.Address{}, consortiumCommon.ErrMissingSignature
    }
    signature := header.Extra[len(header.Extra)-consortiumCommon.ExtraSeal:]

    // Recover the public key and the Ethereum address
    pubkey, err := crypto.Ecrecover(SealHash(header, chainId).Bytes(), signature)
    if err != nil {
        return common.Address{}, err
    }
    var signer common.Address
    copy(signer[:], crypto.Keccak256(pubkey[1:])[12:])

    sigcache.Add(hash, signer)
    return signer, nil
}
```

Figure 64 Transaction signature check(2/2)

If the Ronin node is malicious, the signature checks fails here, but the verification node also checks the transaction signature again, and the forged transaction will fail to be verified.

5.7 Fake Deposit Attack

Fake recharge attack is the transaction execution failure, but the corresponding receipt status shows success. In Ronin, there are only two types of transaction status, Failed(0x0) and Success(0x1). For transactions that

pass the check, both gas exceeds the block limit and transaction execution fails will return failed, and there is no chain-level fake deposit attack.

Figure 65 Transaction receipt information(1/2)

Figure 66 Transaction receipt information(2/2)

5.8 Contract trading security

Ronin is compatible with EVM transactions. Our team tested various types of EVM contract transactions on the Ronin chain and found that Ethereum contract transactions have no security issues and can run on the Ronin chain.

6 Cross-chain Bridge Security

The Ronin cross-chain bridge contract allows users to transfer assets between Ronin Network and other EVM blockchains. When a deposit event occurs on the main chain, the Bridge component in each validator node will receive it and forward it to Ronin by sending the corresponding transaction. For withdrawals and governance events, it will start on Ronin and then relay on other chains. After auditing, no security issues were found in the cross-chain bridge contract.

6.1 Deposit Feature of Cross-Chain Bridge

Users can top up ETH, ERC20 and ERC721 (NFT) by sending transactions to MainchainGatewayV2 on the main chain and waiting for Ronin to verify the deposit. The verifier will listen to the DepositRequested event on the main chain, and then vote and recharge on Ronin. Before the deposit happens, the gateway should establish a mapping between Ethereum and the token contract on Ronin. For deposits, there is no limit to the amount deposited.

The specific process of deposit is as follows:

(1) The user deposits funds to MainchainGatewayV2 on Mainchain and triggers the DepositRequested event.

(2) The bridge relay listens to the DepositRequested event and forwards the deposit request in the DepositRequested event to the RoninGatewayV2 contract on the Ronin chain.

(3) Bridge validators on the Ronin chain vote on deposit requests.

(4) When the vote is passed, transfer the voucher token corresponding to the deposit funds to the user's address on the Ronin chain.

6.2 Withdrawal Feature of Cross-Chain Bridge

Users can send transactions to the RoninGatewayV2 contract on the Ronin chain and wait for validators on the Ronin chain to vote and sign. After the signature is passed, the user can withdraw money on the main chain with the corresponding signature. It should be noted that, unlike deposits, there are restrictions on the withdrawal amount when making withdrawals in Ronin. The specific restrictions are as follows:

. Withdrawal tiers

There are 3 withdrawal tiers with different level of threshold required to withdraw. This is what we propose initially

Tier	Withdrawal Value	Threshold
Tier 1	-	The normal withdrawal/deposit threshold
Tier 2	>= `highTierThreshold(token)`	Applied the special threshold for high-tier withdrawals
Tier 3	>= `lockedThreshold(token)`	Applied the special threshold for high-tier withdrawals, one additional human review to unlock the fund

Figure 67 Withdrawal Limit

The specific process of withdrawal is as follows:

(1) The user deposits funds to the RoninGatewayV2 on Ronin and triggers the WithdrawalRequested event.

(2) The bridge worker monitors the WithdrawalRequested event and signs the withdrawn WithdrawalRequested event.

(3) The user collects the corresponding signatures of the WithdrawalRequested event, and takes these signatures and corresponding withdrawal information to Mainchain for withdrawal.

(4) When withdrawing money, the signature will be verified, and the withdrawal amount will also be judged; if the signature is correct and the withdrawal amount meets the conditions, the corresponding deposit funds will be transferred to the user's address on the Mainchain chain and the withdrawal will be triggered Withdraw event.

(5) After the verifiers on the Ronin chain listen to the Withdraw event, they will vote on the Withdraw event. If the vote is passed, it means that the withdrawal event has been successfully confirmed on the Ronin chain. so far, the entire withdrawal process is over.

6.3 Migration Feature of Cross-Chain Bridge

When the funds in MainchainGatewayV2 on the Mainchain chain need to be migrated, the migrator will call the migration function on the Ronin chain to migrate funds.

Historical Vulnerability Detection

Ronin is a fork of Go-Ethereum Ver.1.10.13 from over 2 years ago. Through this audit, it was found that there were vulnerabilities that are the same as those in the historical version of Go-Ethereum. While development of Ronin has diverged from Ethereum, it is still highly recommended that the project party continue to place enough security resources to continuously check and fix the existence of vulnerabilities and issues that have been exposed by Ethereum to prevent damage to Ronin nodes.

Consensus Contracts Audit Categories

No.	Categories	Subitems
1	Coding Conventions	Compiler Version Security
		Deprecated Items
		Redundant Code
		require/assert Usage
		Gas Consumption
2	General Vulnerability	Integer Overflow/Underflow
		Reentrancy
		Pseudo-random Number Generator (PRNG)
		Transaction-Ordering Dependence
		DoS (Denial of Service)
		Function Call Permissions
		call/delegatecall Security
		Returned Value Security
		tx.origin Usage
		Replay Attack
		Overriding Variables
		Third-party Protocol Interface Consistency
3	Business Security	Business Logics
		Business Implementations
		Manipulable Token Price
		Centralized Asset Control
		Asset Tradability
		Arbitrage Attack

Beosin classified the security issues of smart contracts into three categories: Coding Conventions, General Vulnerability, Business Security. Their specific definitions are as follows:

- **Coding Conventions**

Audit whether smart contracts follow recommended language security coding practices. For example, smart contracts developed in Solidity language should fix the compiler version and do not use deprecated keywords.

- **General Vulnerability**

General Vulnerability include some common vulnerabilities that may appear in smart contract projects. These vulnerabilities are mainly related to the characteristics of the smart contract itself, such as integer overflow/underflow and denial of service attacks.

- **Business Security**

Business security is mainly related to some issues related to the business realized by each project, and has a relatively strong pertinence. For example, whether the lock-up plan in the code match the white paper, or the flash loan attack caused by the incorrect setting of the price acquisition oracle.

*Note that the project may suffer stake losses due to the integrated third-party protocol. This is not something Beosin can control. Business security requires the participation of the project party. The project party and users need to stay vigilant at all times.

Appendix

1.1 Vulnerability Assessment Metrics and Status in Smart Contracts

1.1.1 Metrics

In order to objectively assess the severity level of vulnerabilities in blockchain systems, this report provides detailed assessment metrics for security vulnerabilities in smart contracts with reference to CVSS 3.1 (Common Vulnerability Scoring System Ver 3.1).

According to the severity level of vulnerability, the vulnerabilities are classified into four levels: "critical", "high", "medium" and "low". It mainly relies on the degree of impact and likelihood of exploitation of the vulnerability, supplemented by other comprehensive factors to determine of the severity level.

Likelihood \ Impact	Severe	High	Medium	Low
Probable	Critical	High	Medium	Low
Possible	High	High	Medium	Low
Unlikely	Medium	Medium	Low	Info
Rare	Low	Low	Info	Info

1.1.2 Degree of impact

- **Severe**

Severe impact generally refers to the vulnerability can have a serious impact on the confidentiality, integrity, availability of smart contracts or their economic model, which can cause substantial economic losses to the contract business system, large-scale data disruption, loss of authority management, failure of key functions, loss of credibility, or indirectly affect the operation of other smart contracts associated with it and cause substantial losses, as well as other severe and mostly irreversible harm.

- **High**

High impact generally refers to the vulnerability can have a relatively serious impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a greater economic loss, local functional unavailability, loss of credibility and other impact to the contract business system.

- **Medium**

Medium impact generally refers to the vulnerability can have a relatively minor impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a small amount of economic loss to the contract business system, individual business unavailability and other impact.

- **Low**

Low impact generally refers to the vulnerability can have a minor impact on the smart contract, which can pose certain security threat to the contract business system and needs to be improved.

3.1.4 Likelihood of Exploitation

- **Probable**

Probable likelihood generally means that the cost required to exploit the vulnerability is low, with no special exploitation threshold, and the vulnerability can be triggered consistently.

- **Possible**

Possible likelihood generally means that exploiting such vulnerability requires a certain cost, or there are certain conditions for exploitation, and the vulnerability is not easily and consistently triggered.

- **Unlikely**

Unlikely likelihood generally means that the vulnerability requires a high cost, or the exploitation conditions are very demanding and the vulnerability is highly difficult to trigger.

- **Rare**

Rare likelihood generally means that the vulnerability requires an extremely high cost or the conditions for exploitation are extremely difficult to achieve.

1.1.5 Fix Results Status

Status	Description
Fixed	The project party fully fixes a vulnerability.
Partially Fixed	The project party did not fully fix the issue, but only mitigated the issue.
Acknowledged	The project party confirms and chooses to ignore the issue.

1.2 Disclaimer

The Audit Report issued by Beosin is related to the services agreed in the relevant service agreement. The Project Party or the Served Party (hereinafter referred to as the "Served Party") can only be used within the conditions and scope agreed in the service agreement. Other third parties shall not transmit, disclose, quote, rely on or tamper with the Audit Report issued for any purpose.

The Audit Report issued by Beosin is made solely for the code, and any description, expression or wording contained therein shall not be interpreted as affirmation or confirmation of the project, nor shall any warranty or guarantee be given as to the absolute flawlessness of the code analyzed, the code team, the business model or legal compliance.

The Audit Report issued by Beosin is only based on the code provided by the Served Party and the technology currently available to Beosin. However, due to the technical limitations of any organization, and in the event that the code provided by the Served Party is missing information, tampered with, deleted, hidden or subsequently altered, the audit report may still fail to fully enumerate all the risks.

The Audit Report issued by Beosin in no way provides investment advice on any project, nor should it be utilized as investment suggestions of any type. This report represents an extensive evaluation process designed to help our customers improve code quality while mitigating the high risks in blockchain.

1.3 About Beosin

Beosin is the first institution in the world specializing in the construction of blockchain security ecosystem. The core team members are all professors, postdocs, PhDs, and Internet elites from world-renowned academic institutions. Beosin has more than 20 years of research in formal verification technology, trusted computing, mobile security and kernel security, with overseas experience in studying and collaborating in project research at well-known universities. Through the security audit and defense deployment of more than 2,000 smart contracts, over 50 public blockchains and wallets, and nearly 100 exchanges worldwide, Beosin has accumulated rich experience in security attack and defense of the blockchain field, and has developed several security products specifically for blockchain.

BEOSIN
Blockchain Security

Official Website
https://www.beosin.com
Telegram
https://t.me/+dD8Bnqd133RmNWNl
Twitter
https://twitter.com/Beosin_com
Email
Contact@beosin.com