BEOSIN
Blockchain Security

Smart Contract Security Audit

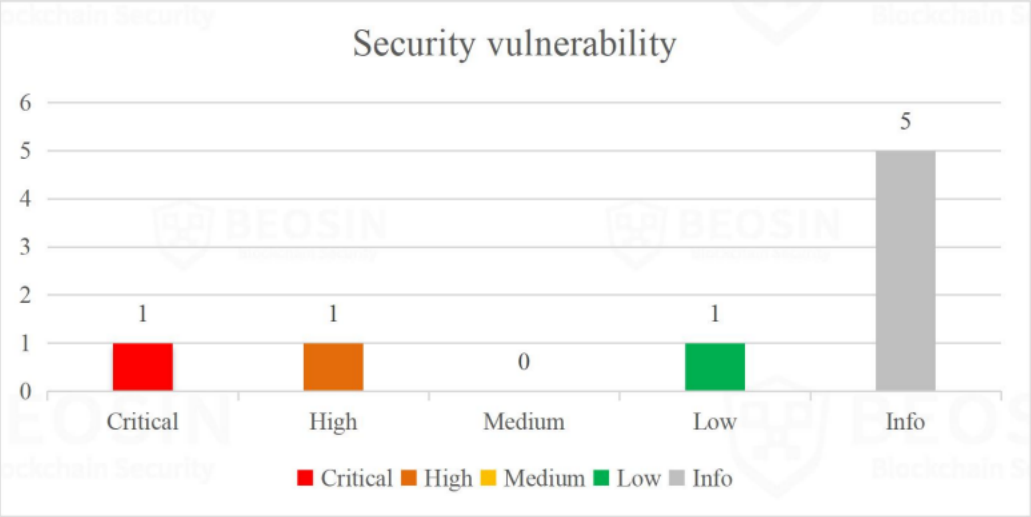
V1.3

eRon Blockchain using bas framework services

Contents

Summary of audit results

After auditing, 1 Critical-risk, 1 High-risk, 1 Medium-risk and 5 Info items were identified in the Ankr bas project. Specific audit details will be presented in the **Findings section**. Users should pay attention to the following aspects when interacting with this project:



*Notes:

- **Risk Description:**

1. If contract use the latest openzeppelin-contracts, there may be previous problems. Because the Governor in the latest openzeppelin-contracts contract has added a _castVote, it will cause the vote to still be manipulated. Please make sure to use the correct openzeppelin version.

- **Project Description:**

1. Business overview

The Staking contract implements the Validator registration function and the user stake function. Anyone can register as a Validator by pledging the corresponding funds through the Staking contract, and after registration, the Validator can only become a Validator if the Governance contract is voted on. The Governance contract can be initiated by the Validator address and must have more than two-thirds of the votes before the proposal can succeed; the RuntimeUpgrade contract is used to upgrade the system contract.

1 Overview

1.1 Project Overview

Project Name	eRon Project using bas framework services
Platform	https://*.eronscan.com

File Hash (SHA256)	Staking.sol	466e8bf3e88fb7f828bb89fb2b7c21c4e4ca6d042215a8daa1dffab0e512a6c8 ad2fdf8565190b1b9972fe91fa6fa4e044c7f783a5b0423381663f6330d20f83
	StakingPool.sol	1eca905566e42760e6cedcb0e0d9d6ad35e94b3f1d5dd8a857afe1c14cef70bd
	Injector.sol	37a7d2351fa0e9e42907231de3a54651be952c045c45562e846eb1b2787902bf
	RuntimeUpgrade.sol	5b9e85557561c1895c55b1a1b60d8b15112b1fe9864ff18c7d9db5c0dab2050f
	Governance.sol	5c76fc9e0b25d805bc0045a3ecbde8da89b577a243886d99f35a4c8637b3e234 2caf68fedf5e6ead15f496a8d06dc5c63f003e7bcb8672dd497c55745550e497

1.2 Audit Overview

Update report time: April 26

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

Audit team: Beosin Technology Co. Ltd.

2 Findings

Index	Risk description	Severity level	Status
1	A validator can vote multiple times	Critical	Fixed
2	Poorly designed *ctor* function	High	Fixed
3	User funds will not be available for withdrawal	Low	Fixed
4	The *_slashValidator* function is not rigorously judged	Info	Fixed
5	Poorly designed *undelegate* function	Info	Fixed
6	Poorly designed *_delegateTo* function	Info	Fixed
7	Missing events	Info	Fixed
8	Poorly designed *claim* function	Info	Fixed

	A validator can vote multiple times
Severity Level	**Critical**
Type	Business Security
Lines	Governance.sol#
Description	In the Governance contract, only the ValidatorOwner address can vote, but in the Staking contract, the ValidatorOwner address can be modified through the *changeValidatorOwner* function, and then you can still vote.
Recommendations	It is recommended to use validator to count the votes.
Status	Fixed.

```
68
69  function _castVote(uint256 proposalId, address account, uint8 support, string memory reason) internal virtual override onlyValidatorOwner(account) returns (uint256) {
70      address validatorAddress = _stakingContract.getValidatorByOwner(account);
71      return super._castVote(proposalId, validatorAddress, support, reason);
72  }
73
```

Figure 1 Source code of *_castVote* function (Fixed)

Poorly designed *ctor* function

Severity Level	**High**
Type	Business Security
Lines	Staking.sol#L122
Description	The *ctor* function in the staking contract should not specify initialStakes, because this function does not transfer the corresponding funds. If the validator has other users participating in the stake, it will cause the validator to withdraw the stake funds of other users.

```
118
119     function ctor(address[] calldata validators, uint256[] calldata initialStakes, uint16 commissionRate) external whenNotInitialized {
120         require(initialStakes.length == validators.length);
121         for (uint256 i = 0; i < validators.length; i++) {
122             _addValidator(validators[i], validators[i], ValidatorStatus.Active, commissionRate, initialStakes[i], 0);
123         }
124     }
125
```

Figure 2 Source code of *ctor* function

```
485
486 function _addValidator(address validatorAddress, address validatorOwner, ValidatorStatus status, uint16 commissionRate, uint256 initialStake, uint64 sinceEpoch) internal {
487     // validator commission rate
488     require(commissionRate >= COMMISSION_RATE_MIN_VALUE && commissionRate <= COMMISSION_RATE_MAX_VALUE, "Staking: bad commission rate");
489     // init validator default params
490     Validator memory validator = _validatorsMap[validatorAddress];
491     require(_validatorsMap[validatorAddress].status == ValidatorStatus.NotFound, "Staking: validator already exist");
492     validator.validatorAddress = validatorAddress;
493     validator.ownerAddress = validatorOwner;
494     validator.status = status;
495     validator.changedAt = sinceEpoch;
496     _validatorsMap[validatorAddress] = validator;
497     // save validator owner
498     require(_validatorOwners[validatorOwner] == address(0x00), "Staking: owner already in use");
499     _validatorOwners[validatorOwner] = validatorAddress;
500     // add new validator to array
501     if (status == ValidatorStatus.Active) {
502         _activeValidatorsList.push(validatorAddress);
503     }
504     // push initial validator snapshot at zero epoch with default params
505     _validatorSnapshots[validatorAddress][sinceEpoch] = ValidatorSnapshot(0, uint112(initialStake / BALANCE_COMPACT_PRECISION), 0, commissionRate);
506     // delegate initial stake to validator owner
507     ValidatorDelegation storage delegation = _validatorDelegations[validatorAddress][validatorOwner];
508     require(delegation.delegateQueue.length == 0, "Staking: delegation queue is not empty");
509     delegation.delegateQueue.push(DelegationOpDelegate(uint112(initialStake / BALANCE_COMPACT_PRECISION), sinceEpoch));
510     // emit event
511     emit ValidatorAdded(validatorAddress, validatorOwner, uint8(status), commissionRate);
512 }
513
```

Figure 3 Source code of *_addValidator* function (Fixed)

Recommendations	It is recommended to set initialStakes to zero.
Status	Fixed.

```
118
119     function ctor(address[] calldata validators, uint256[] calldata initialStakes, uint16 commissionRate) external whenNotInitialized {
120         require(initialStakes.length == validators.length);
121         uint256 totalStakes = 0;
122         for (uint256 i = 0; i < validators.length; i++) {
123             _addValidator(validators[i], validators[i], ValidatorStatus.Active, commissionRate, initialStakes[i], 0);
124             totalStakes += initialStakes[i];
125         }
126         require(address(this).balance == totalStakes, "Staking: initial stake balance mismatch");
127     }
128
```

Figure 4 Source code of *ctor* function (Fixed)

User funds will not be available for withdrawal

Severity Level	Low
Type	Business Security
Lines	Staking.sol#L313, 535-544
Description	After the validator is deleted through governance, if the validator has stake funds, the user will not be able to withdraw the funds staked on the validator.

```
513
514     function removeValidator(address account) external onlyFromGovernance virtual override {
515         _removeValidator(account);
516     }
517
518     function _removeValidatorFromActiveList(address validatorAddress) internal {
519         // find index of validator in validator set
520         int256 indexOf = -1;
521         for (uint256 i = 0; i < _activeValidatorsList.length; i++) {
522             if (_activeValidatorsList[i] != validatorAddress) continue;
523             indexOf = int256(i);
524             break;
525         }
526         // remove validator from array (since we remove only active it might not exist in the list)
527         if (indexOf >= 0) {
528             if (_activeValidatorsList.length > 1 && uint256(indexOf) != _activeValidatorsList.length - 1) {
529                 _activeValidatorsList[uint256(indexOf)] = _activeValidatorsList[_activeValidatorsList.length - 1];
530             }
531             _activeValidatorsList.pop();
532         }
533     }
534
535     function _removeValidator(address account) internal {
536         Validator memory validator = _validatorsMap[account];
537         require(validator.status != ValidatorStatus.NotFound, "Staking: validator not found");
538         // remove validator from active list if exists
539         _removeValidatorFromActiveList(account);
540         // remove from validators map
541         delete _validatorOwners[validator.ownerAddress];
542         delete _validatorsMap[account];
543         // emit event about it
544         emit ValidatorRemoved(account);
545     }
546
```

Figure 5 Source code of *_removeValidator* function (Fixed)

```
306
307 function _undelegateFrom(address toDelegator, address fromValidator, uint256 amount) internal {
308     // check minimum delegate amount
309     require(amount >= _chainConfigContract.getMinStakingAmount() && amount != 0, "Staking: amount is too low");
310     require(amount % BALANCE_COMPACT_PRECISION == 0, "Staking: amount have a remainder");
311     // make sure validator exists at least
312     Validator memory validator = _validatorsMap[fromValidator];
313     require(validator.status != ValidatorStatus.NotFound, "Staking: validator not found");
314     uint64 beforeEpoch = _nextEpoch();
315     // Lets upgrade next snapshot parameters:
316     // + find snapshot for the next epoch after current block
317     // + increase total delegated amount in the next epoch for this validator
318     // + re-save validator because last affected epoch might change
319     ValidatorSnapshot storage validatorSnapshot = _touchValidatorSnapshot(validator, beforeEpoch);
320     require(validatorSnapshot.totalDelegated >= uint112(amount / BALANCE_COMPACT_PRECISION), "Staking: insufficient balance");
321     validatorSnapshot.totalDelegated -= uint112(amount / BALANCE_COMPACT_PRECISION);
322     _validatorsMap[fromValidator] = validator;
323     // if last pending delegate has the same next epoch then its safe to just increase total
324     // staked amount because it can't affect current validator set, but otherwise we must create
325     // new record in delegation queue with the last epoch (delegations are ordered by epoch)
326     ValidatorDelegation storage delegation = _validatorDelegations[fromValidator][toDelegator];
327     require(delegation.delegateQueue.length > 0, "Staking: delegation queue is empty");
328     DelegationOpDelegate storage recentDelegateOp = delegation.delegateQueue[delegation.delegateQueue.length - 1];
329     require(recentDelegateOp.amount >= uint64(amount / BALANCE_COMPACT_PRECISION), "Staking: insufficient balance");
330     uint112 nextDelegatedAmount = recentDelegateOp.amount - uint112(amount / BALANCE_COMPACT_PRECISION);
331     if (recentDelegateOp.epoch >= beforeEpoch) {
332         // decrease total delegated amount for the next epoch
333         recentDelegateOp.amount = nextDelegatedAmount;
334     } else {
335         // there is no pending delegations, so lets create the new one with the new amount
336         delegation.delegateQueue.push(DelegationOpDelegate({epoch : beforeEpoch, amount : nextDelegatedAmount}));
337     }
338     // create new undelegate queue operation with soft lock
339     delegation.undelegateQueue.push(DelegationOpUndelegate({amount : uint112(amount / BALANCE_COMPACT_PRECISION), epoch : beforeEpoch + _chainConfigContract.getUndelegatePeriod()}));
340     // emit event with the next epoch number
341     emit Undelegated(fromValidator, toDelegator, amount, beforeEpoch);
342 }
343
```

Figure 6 Source code of *_undelegateFrom* function (Fixed)

Recommendations	It is recommended to remove the validator after the funds in the validator have been withdrawn.

Status Fixed.

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

Figure 7 Source code of *_undelegateFrom* function (Fixed)

The *_slashValidator* function is not rigorously judged

Severity Level	Info
Type	Business Security
Lines	Staking.sol#L741,743
Description	In the *_slashValidator* function, "validator.status != ValidatorStatus.NotFound" is judged, because "make sure validator was active" is also written in the comment. So the function here should judge validator.status == ValidatorStatus.Active.

```
739
740     function _slashValidator(address validatorAddress) internal {
741         // make sure validator was active
742         Validator memory validator = _validatorsMap[validatorAddress];
743         require(validator.status != ValidatorStatus.NotFound, "Staking: validator not found");
744         uint64 epoch = _currentEpoch();
745         // increase slashes for current epoch
746         ValidatorSnapshot storage currentSnapshot = _touchValidatorSnapshot(validator, epoch);
747         uint32 slashesCount = currentSnapshot.slashesCount + 1;
748         currentSnapshot.slashesCount = slashesCount;
749         // validator state might change, lets update it
750         _validatorsMap[validatorAddress] = validator;
751         // if validator has a lot of misses then put it in jail for 1 week (if epoch is 1 day)
752         if (slashesCount == _chainConfigContract.getFelonyThreshold()) {
753             validator.jailedBefore = _currentEpoch() + _chainConfigContract.getValidatorJailEpochLength();
754             validator.status = ValidatorStatus.Jail;
755             _removeValidatorFromActiveList(validatorAddress);
756             _validatorsMap[validatorAddress] = validator;
757             emit ValidatorJailed(validatorAddress, epoch);
758         }
759         // emit event
760         emit ValidatorSlashed(validatorAddress, slashesCount, epoch);
761     }
762 }
763
```

Figure 8 Source code of *_slashValidator* function (Fixed)

Recommendations	It is recommended to determine the status of the validator as active.
Status	Partially Fixed. Project party description: Validator can be slashed even if this validator is already in jail because epoch might be still active where this validator is in the active validator set. They've changed the misleading comment for this line.

```
740
741     function _slashValidator(address validatorAddress) internal {
742         // make sure validator exists
743         Validator memory validator = _validatorsMap[validatorAddress];
744         require(validator.status != ValidatorStatus.NotFound, "Staking: validator not found");
745         uint64 epoch = _currentEpoch();
746         // increase slashes for current epoch
747         ValidatorSnapshot storage currentSnapshot = _touchValidatorSnapshot(validator, epoch);
748         uint32 slashesCount = currentSnapshot.slashesCount + 1;
749         currentSnapshot.slashesCount = slashesCount;
750         // validator state might change, lets update it
751         _validatorsMap[validatorAddress] = validator;
752         // if validator has a lot of misses then put it in jail for 1 week (if epoch is 1 day)
753         if (slashesCount == _chainConfigContract.getFelonyThreshold()) {
754             validator.jailedBefore = _currentEpoch() + _chainConfigContract.getValidatorJailEpochLength();
755             validator.status = ValidatorStatus.Jail;
756             _removeValidatorFromActiveList(validatorAddress);
757             _validatorsMap[validatorAddress] = validator;
758             emit ValidatorJailed(validatorAddress, epoch);
759         }
760         // emit event
761         emit ValidatorSlashed(validatorAddress, slashesCount, epoch);
762     }
763 }
```

Figure 9 Source code of *_slashValidator* function (Fixed)

	Poorly designed *undelegate* function
Severity Level	Info
Type	Business Security
Lines	Staking.sol#L216
Description	In the *undelegate* function, there is no operation on msg.value.

```
214     }
215
216     function undelegate(address validatorAddress, uint256 amount) payable external override {
217         _undelegateFrom(msg.sender, validatorAddress, amount);
218     }
219
```

Figure 10 Source code of *undelegate* function (Fixed)

Recommendations	It is recommended to delete the payable.
Status	Fixed.

```
215
216     function undelegate(address validatorAddress, uint256 amount) external override {
217         _undelegateFrom(msg.sender, validatorAddress, amount);
218     }
219
```

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

	Poorly designed *_delegateTo* function
Severity Level	Info
Type	Business Security
Lines	Staking.sol#L277
Description	In the *_delegateTo* function of StakingPool, it is judged as "validator.status != ValidatorStatus.NotFound", which means that when the validator's status is Pending or Jail, users can also stake.

```
function _delegateTo(address fromDelegator, address toValidator, uint256 amount) internal {
    // check is minimum delegate amount
    require(amount >= _chainConfigContract.getMinStakingAmount() && amount != 0, "Staking: amount is too low");
    require(amount % BALANCE_COMPACT_PRECISION == 0, "Staking: amount have a remainder");
    // make sure amount is greater than min staking amount
    // make sure validator exists at least
    Validator memory validator = _validatorsMap[toValidator];
    require(validator.status != ValidatorStatus.NotFound, "Staking: validator not found");
    uint64 atEpoch = _nextEpoch();
    // Lets upgrade next snapshot parameters:
    // + find snapshot for the next epoch after current block
    // + increase total delegated amount in the next epoch for this validator
    // + re-save validator because last affected epoch might change
    ValidatorSnapshot storage validatorSnapshot = _touchValidatorSnapshot(validator, atEpoch);
    validatorSnapshot.totalDelegated += uint112(amount / BALANCE_COMPACT_PRECISION);
    _validatorsMap[toValidator] = validator;
    // if last pending delegate has the same next epoch then its safe to just increase total
    // staked amount because it can't affect current validator set, but otherwise we must create
    // new record in delegation queue with the last epoch (delegations are ordered by epoch)
    ValidatorDelegation storage delegation = _validatorDelegations[toValidator][fromDelegator];
    if (delegation.delegateQueue.length > 0) {
        DelegationOpDelegate storage recentDelegateOp = delegation.delegateQueue[delegation.delegateQueue.length - 1];
        // if we already have pending snapshot for the next epoch then just increase new amount,
        // otherwise create next pending snapshot. (tbh it can't be greater, but what we can do here instead?)
        if (recentDelegateOp.epoch >= atEpoch) {
            recentDelegateOp.amount += uint112(amount / BALANCE_COMPACT_PRECISION);
        } else {
            delegation.delegateQueue.push(DelegationOpDelegate({epoch : atEpoch, amount : recentDelegateOp.amount + uint112(amount / BALANCE_COMPACT_PRECISION)}));
        }
    } else {
        // there is no any delegations at al, lets create the first one
        delegation.delegateQueue.push(DelegationOpDelegate({epoch : atEpoch, amount : uint112(amount / BALANCE_COMPACT_PRECISION)}));
    }
    // emit event with the next epoch
    emit Delegated(toValidator, fromDelegator, amount, atEpoch);
}
```

Figure 12 Source code of *preMint* function (Fixed)

Recommendations	It is recommended that when the state of the Validator is active before it can be staked.
Status	Acknowledged. Project party description: They can't limit validators from being elected even if they are in jail or not active. Stakers who delegate money to jailed or inactive validators will be punished because they won't gain any rewards for it. But the validator owner might want to increase the total staked amount for his validator just to increase its position in the active validator list and be prepared for validating blocks right after the jail period ends.

	Missing events
Severity Level	Info
Type	Business Security
Lines	Staking.sol#L551-569
Description	The *_disableValidator* and *_activateValidator* functions in the Staking contract lack the corresponding event triggers,

```
551  function _activateValidator(address validatorAddress) internal {
552      Validator memory validator = _validatorsMap[validatorAddress];
553      require(_validatorsMap[validatorAddress].status == ValidatorStatus.Pending, "Staking: not pending validator");
554      _activeValidatorsList.push(validatorAddress);
555      validator.status = ValidatorStatus.Active;
556      _validatorsMap[validatorAddress] = validator;
557  }
558
559  function disableValidator(address validator) external onlyFromGovernance virtual override {
560      _disableValidator(validator);
561  }
562
563  function _disableValidator(address validatorAddress) internal {
564      Validator memory validator = _validatorsMap[validatorAddress];
565      require(_validatorsMap[validatorAddress].status == ValidatorStatus.Active, "Staking: not active validator");
566      _removeValidatorFromActiveList(validatorAddress);
567      validator.status = ValidatorStatus.Pending;
568      _validatorsMap[validatorAddress] = validator;
569  }
```

Figure 13 Source code of *_disableValidator&_activateValidator* functions (Fixed)

Recommendations	It is recommended to add their event triggers.
Status	Fixed.

```
551  function _activateValidator(address validatorAddress) internal {
552      Validator memory validator = _validatorsMap[validatorAddress];
553      require(_validatorsMap[validatorAddress].status == ValidatorStatus.Pending, "Staking: not pending validator");
554      _activeValidatorsList.push(validatorAddress);
555      validator.status = ValidatorStatus.Active;
556      _validatorsMap[validatorAddress] = validator;
557      ValidatorSnapshot storage snapshot = _touchValidatorSnapshot(validator, _nextEpoch());
558      emit ValidatorModified(validatorAddress, validator.ownerAddress, uint8(validator.status), snapshot.commissionRate);
559  }
560
561  function disableValidator(address validator) external onlyFromGovernance virtual override {
562      _disableValidator(validator);
563  }
564
565  function _disableValidator(address validatorAddress) internal {
566      Validator memory validator = _validatorsMap[validatorAddress];
567      require(_validatorsMap[validatorAddress].status == ValidatorStatus.Active, "Staking: not active validator");
568      _removeValidatorFromActiveList(validatorAddress);
569      validator.status = ValidatorStatus.Pending;
570      _validatorsMap[validatorAddress] = validator;
571      ValidatorSnapshot storage snapshot = _touchValidatorSnapshot(validator, _nextEpoch());
572      emit ValidatorModified(validatorAddress, validator.ownerAddress, uint8(validator.status), snapshot.commissionRate);
573  }
```

Figure 14 Source code of *_disableValidator&_activateValidator* functions (Fixed)

Poorly designed *claim* function

Severity Level	Info
Type	Business Security
Lines	StakingPool.sol#L166
Description	When the user does not cancel the stake, the pendingUnstake.epoch at this time is equal to zero, then the use of greater than or equal to zero here is constant.

```
160
161     function claim(address validator) external advanceStakingRewards(validator) override {
162         PendingUnstake memory pendingUnstake = _pendingUnstakes[validator][msg.sender];
163         uint256 amount = pendingUnstake.amount;
164         uint256 shares = pendingUnstake.shares;
165         // make sure user have pending unstake
166         require(pendingUnstake.epoch >= 0, "StakingPool: nothing to claim");
167         require(pendingUnstake.epoch <= _stakingContract.currentEpoch(), "StakingPool: not ready");
168         // updates shares and validator pool params
169         _stakerShares[validator][msg.sender] -= shares;
170         ValidatorPool memory validatorPool = _getValidatorPool(validator);
171         validatorPool.sharesSupply -= shares;
172         validatorPool.totalStakedAmount -= amount;
173         validatorPool.pendingUnstake -= amount;
174         _validatorPools[validator] = validatorPool;
175         // remove pending claim
176         delete _pendingUnstakes[validator][msg.sender];
177         // its safe to use call here (state is clear)
178         require(address(this).balance >= amount, "StakingPool: not enough balance");
179         payable(address(msg.sender)).transfer(amount);
180         // emit event
181         emit Claim(validator, msg.sender, amount);
182     }
183
```

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

Recommendations	It is recommended to modify it to be greater than zero.
Status	Fixed.

```
160
161     function claim(address validator) external advanceStakingRewards(validator) override {
162         PendingUnstake memory pendingUnstake = _pendingUnstakes[validator][msg.sender];
163         uint256 amount = pendingUnstake.amount;
164         uint256 shares = pendingUnstake.shares;
165         // make sure user have pending unstake
166         require(pendingUnstake.epoch > 0, "StakingPool: nothing to claim");
167         require(pendingUnstake.epoch <= _stakingContract.currentEpoch(), "StakingPool: not ready");
168         // updates shares and validator pool params
169         _stakerShares[validator][msg.sender] -= shares;
170         ValidatorPool memory validatorPool = _getValidatorPool(validator);
171         validatorPool.sharesSupply -= shares;
172         validatorPool.totalStakedAmount -= amount;
173         validatorPool.pendingUnstake -= amount;
174         _validatorPools[validator] = validatorPool;
175         // remove pending claim
176         delete _pendingUnstakes[validator][msg.sender];
177         // its safe to use call here (state is clear)
178         require(address(this).balance >= amount, "StakingPool: not enough balance");
179         payable(address(msg.sender)).transfer(amount);
180         // emit event
181         emit Claim(validator, msg.sender, amount);
182     }
183
184     receive() external payable {
185         require(address(msg.sender) == address(_stakingContract));
186     }
187 }
```

Figure 16 Source code of *claim* function (Fixed)

3 Appendix

3.1 Vulnerability Assessment Metrics and Status in Smart Contracts

3.1.1 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

3.1.2 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.

3.1.5 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.

3.2 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.

3.3 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.

BEOSIN
Blockchain Security