Flare

Security Review

Smart Contracts V2

coinspect

Smart Contracts v2

Smart Contract Security Review

Version: v240515 | Prepared for: Flare | May 2024

Security Assessment

1. Executive Summary

2.3 Solved issues & recommendations

3. Scope

4 Assessment

4.1 Security assumptions

4.2 Decentralization

4.3 Code Quality & Testing

5. Detailed Findings

FSC-01 - Voters can use any signing policy to vote for a new one

© Coinspect 2024

FSC-02 - Signing policies of certain length break the contract

FSC-03 - Relayers can set an invalid signing policy

FSC-04 - Callers can mistakenly use unsafe random values

FSC-05 - Arbitrary values for firstVotingRoundStartTs disrupt the functioning of FTSOs and Stake Mirrorings

FSC-06 - RewardEpochId size mismatch between contract and top-level-client

FSC-07 - Attacker could use submitAndPassContract to waste resources

FSC-08 - Permanent data staleness threatens financial stability of third parties

FSC-09 - Inflation receiver childs might not implement key functions

FSC-10 - Voters can have zero voting power after normalization

FSC-11 - Voter cap mismatch endangers system's voting integrity

FSC-12 - Governance calls can be return bombed

FSC-13 - Attacker can gain unfair voting power by timing first daemonization

FSC-14 - Evil voters can steal voting power through node registration

FSC-15 - Daemonize calls revert with high reward expiry offsets

FSC-16 - Tests heavily relying on mock calls could fail to reproduce adversarial scenarios

FSC-17 - Broken invariant could disrupt how third parties consume the amount of registered voters

FSC-18 - Chilling bypassed by re-delegating voting power

6. Disclaimer

© Coinspect 2024

1. Executive Summary

In **January 2024**, **Flare** engaged Coinspect to perform a source code review of its **Smart Contracts v2** repositories. The objective of the project was to evaluate the security of the smart contracts, which are one of the most important components of the Flare ecosystem.

The **Smart Contracts v2** contracts are responsible for several tasks: coordination of voters of the Flare System protocol and validation of their signatures, an implementation of a `Governance` which authorizes certain actions in other smart contracts, inflation and rewards management in the network among other core features of Flare.

Solved	Caution Advised	Resolution Pending
High 2	High 0	High 0
Medium 3	Medium 0	Medium 0
Low 3	Low 0	Low 0
No Risk 10	No Risk 0	No Risk 0
Total **18**	Total **0**	Total **0**

In this security assessment, Coinspect identified 2 high-risk, 3 medium-risk and 3 low-risk issues. Also, 10 no-risk issues are included. These no-risk issues are recommendations aiming to enhance the project's long-term security and integrity. The

reported issues with more impact mainly relate to several cases where evil voters could get unfair advantage over others. For example, by manipulating their vote power, bypassing penalties and using older signing policies (voting rules).

© Coinspect 2024

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-01	Voters can use any signing policy to vote for a new one	High
FSC-14	Evil voters can steal voting power through node registration	High
FSC-02	Signing policies of certain length break the contract	Medium
FSC-03	Relayers can set an invalid signing policy	Medium
FSC-13	Attacker can gain unfair voting power by timing first daemonization	Medium
FSC-04	Callers can mistakenly use unsafe random values	Low
FSC-08	Permanent data staleness threatens financial stability of third parties	Low
FSC-11	Voter cap mismatch endangers system's voting integrity	Low
FSC-05	Arbitrary values for firstVotingRoundStartTs disrupt the functioning of FTSOs and Stake Mirrorings	None
FSC-06	RewardEpochId size mismatch between contract and top-level-client	None
FSC-07	Attacker could use submitAndPassContract to waste resources	None
FSC-09	Inflation receiver childs might not implement key functions	None

FSC-10	Voters can have zero voting power after normalization	None
FSC-12	Governance calls can be return bombed	None
FSC-15	Daemonize calls revert with high reward expiry offsets	None
FSC-16	Tests heavily relying on mock calls could fail to reproduce adversarial scenarios	None
FSC-17	Broken invariant could disrupt how third parties consume the amount of registered voters	None
FSC-18	Chilling bypassed by re-delegating voting power	None

3. Scope

The initial scope was focused on the Flare Smart Contracts V2 repository at commit `e84833be205602322def5eb08a658d8bba42c6bc`.

On April 9, at Flare's request, we expanded the scope to include additional contracts: `PollingFTSO`, `PollingFoundation`, `PChainStakeMirrorVerifier`, and `ValidatorRewardsOffersManager`. These were reviewed at commit `57013ccb27d9d93fc901f864993c7589eef19230`. It is important to note that this commit was not fully reviewed; our evaluation was limited to the contracts specified above, none of which presented issues.

© Coinspect 2024

4 Assessment

This review evaluates the security of **Flare V2 Smart Contracts**, the core infrastructure of Flare on which other protocols rely.

This infrastructure allows to:

- Calculate, set and query a list of top voters
- Generate a random number, e.g., consumed when picking a vote power block
- Vote for new signing policies, validator up-time and reward distributions
- Claim rewards
- Relay voting results that derive from off-chain calculations. This feature also enables mirroring the voting results to other chains
- Mirror the stakes made on the **P-Chain**, to get the underlying voting power derived from those stakes
- Publish and query data for price-feeds

It implements a voting mechanism used by users that accrue the top voting power from the community. The voting weight of each user is calculated by adding delegations of `WNat` and stakes in the **P-Chain**, using a checkpoint system. Past stakes and delegations within a specific reward epoch are retrieved at a specific vote power block. This block is a random block in the range of the epoch's blocks calculated using a safe random number.

This system requires to be always updated as it makes several critical time-sensitive calculations. This heartbeat is made by the **Flare Daemon**, that daemonizes the system once per block. On top of each block, calling `daemonize()` performs the following core actions:

- Track down which epochs are ready to be voted
- Allow reward claiming of past epochs
- Cleanup older epochs
- Get random numbers and set vote power blocks
- Update voting power
- Prepare and update the set of voters that can have subsidized calls in commit-reveal scheme

4.1 Security assumptions

© Coinspect 2024

For this security assessment, Coinspect made the following assumptions:

- The `governanceSettings`, `initialGovernance` and `addressUpdater`, `signingPolicySetter` and other data is correctly set on deployment. All other core variables that have a setter can change in the future and could adopt any allowed value by the contract.
- The `flareSystemManager` address can only be used in certain conditions protected by consensus software.
- The majority of voters are non-malicious and do not collude.

4.2 Decentralization

The voting integrity of the protocol depends on the assumption that voters do not collude. This assumption is important as colluding voters could break how the system works by simply voting corrupted or malicious data. For voters that misbehave, the governance has the privilege to chill (pause) their voting power for a specific amount of epochs.

A base governance along with its executor is implemented and many actions of the protocol are privileged. Those privileged actions are mainly adding or removing configurations, upgrading contract addresses and setting the value for core operational variables. Although the governance is considered trusted, those setters that change critical variables have also in place range checks to prevent breaking the system with forbidden values.

The protocol has multiple contracts that communicate with each other. Because of this, the targets (addresses) on each contract can be updated by the governance. This operational aspect is critical, since the whole system might work unexpectedly in the event of not updating the target's addresses when required. It is relevant to mention that setting the wrong contracts (e.g., bad addresses or outdated versions) might cause irreversible impact of unknown risk in the whole system.

4.3 Code Quality & Testing

Coinspect observed the project's code quality is high, taking recommendations of security assessments made for the previous version. Also, the codebase includes relevant comments and NatSpec.

Regarding the testing suite, most tests rely on mock calls and fail to reproduce threat scenarios that might derive from the interaction between contracts. Coinspect

suggests including more adversarial integration tests that do not use mock calls.

5. Detailed Findings

FSC-01

Voters can use any signing policy to vote for a new one

Location

```
contracts/protocol/implementation/Relay.sol
```

Description

A set of voters that had, at any point, reached the threshold needed to pass votes can reuse the signing policy that gave them such powers to relay a new signing policy. This means they can finalize a new signing policy that gives them power again even when they have no stake anymore or were penalized by the system.

© Coinspect 2024

The problem stems from the fact that the timing checks intended to disallow relayers from providing a siging policy that does not match the message's current `votingRoundId` are not not performed when relaying signing policies.

Messages for subprotocols must follow three precise timing rules:

1. A message must not indicate a signing policy for a reward epoch bigger than what is calculated from its `votingRoundId`
2. If a message indicates a singing policy for the same reward epoch that calculated from its `votingRoundId`, that signing policy must have been initialized.
3. If a message indicates a signing policy for a reward epoch that is behind what is calculated from its `votingRoundId`, then this was the last signing policy available and the message is valid with an increased threshold *or* the indicated singing policy was the one active on that reward epoch.

These invariants are crucial to prevent relayers from providing signing policies either from the future or from the past. Unfortunately, this check is not done on the finalization of singing policies themselves, as they are inside a conditional that checks if the `protocolId` is bigger than zero: `if gt(protocolId, 0)`.

What this means is that a set of voters that for *any* reason got a majority of the votes at any point in time can relay a new, arbitrary, singing policy for the next reward epoch. This even applies to voters that get a majority for a non-initialized signing policy.

Therefore, the strategies to prevent evil majorities -- such as slashing or blacklists -- will not be effective.

To be exploitable, the `noSigningPolicyRelay` flag must be set to `false`.

Recommendation

Ensure the finalization of new signing policies is contingent upon the validation against the last active and authorized signing policy's parameters, including voters, weight, and other relevant metadata.

Status

Fixed on commit `421fa87c16af78416bd61bdbc97559006249d07f`.

A check on `relay()` was added that allows using only the last initialized policy for old policies.

FSC-02

Signing policies of certain length break the contract

Location

`contracts/protocol/implementation/Relay.sol`

Description

Setting a signing policy of a specific length will break the relay contract because the hash calculated by the `top-level-client` and the contract will differ for the same signing policy.

To understand the issue, it is important to take into account that a key invariant of the relay contract is that the hash of the
signing policy passed in the `calldata` is the same as the one stored in the `toSigningPolicyHash` mapping:

```
            if iszero(
               eq(
                   mload(add(memPtr, M_2_signingPolicyHashTmp)),
```

```
                    mload(add(memPtr, M_3_existingSigningPolicyHashTmp))
                )
            ) {
                revertWithMessage(memPtr, "Signing policy hash
mismatch", 28)
            }
```

To calculate the contents of the `calldata` signinig policy, the `calculateSigningPolicyHash` is used:

```
            function calculateSigningPolicyHash(
                _memPos,
                _calldataPos,
                _policyLength
            ) -> _policyHash {
                // first byte
                calldatacopy(_memPos, _calldataPos, 32)
                // all but last 32-byte word
                let endPos := add(_calldataPos, mul(div(_policyLength,
32), 32))
                for {
                    let pos := add(_calldataPos, 32)
                } lt(pos, endPos) {
                    pos := add(pos, 32)
                } {
                    calldatacopy(add(_memPos, M_1), pos, 32)
                    mstore(_memPos, keccak256(_memPos, 64))
                }

                // handle the remaining bytes
                mstore(add(_memPos, M_1), 0)
                calldatacopy(add(_memPos, M_1), endPos,
mod(_policyLength, 32)) // remaining bytes
                mstore(_memPos, keccak256(_memPos, 64))
                _policyHash := mload(_memPos)
            }
```

This function has one specific behavior which is not found in the `top-level-client` implementation: the function in `Relay.sol` **always** pads the signing policy with zero bytes until its length is a multiple of 32. It does this even when the signing policy length is already a multiple of 32.

Contrast this with `top-level-behavior`, which pads the signing policy only if needed:

```
func SigningPolicyHash(signingPolicy []byte) []byte {
        if len(signingPolicy)%32 != 0 {
                signingPolicy = append(signingPolicy, make([]byte, 32-
len(signingPolicy)%32)...)
        }
        hash := crypto.Keccak256(signingPolicy[:32],
signingPolicy[32:64])
        for i := 2; i < len(signingPolicy)/32; i++ {
                hash = crypto.Keccak256(hash, signingPolicy[i*32:
```

© Coinspect 2024

```
(i+1)*32])
        }
        return hash
}
```

All in all, this results in the two hashes not coinciding for the same singing policy. While the off chain program will calculate the hash of `[data]`, the smart contract will calculate the hash of `[data : 32-zero-bytes]`.

This mismatch likely impacts several interactions between the components. One of the most important is the process of signing a new policy: when the Top Level Client calls `SignNewSigningPolicy`, it will send its interpretation of the `newSigningPolicyHash`, which will cause the transaction to revert as the first check done by `SignNewSigningPolicy` in the `flareSystemManager.sol` smart contract is:

```
        require(_newSigningPolicyHash != bytes32(0) &&
_getSigningPolicyHash(_rewardEpochId) == _newSigningPolicyHash,
            "new signing policy hash invalid");
```

The only reason why this issue is of low-likelihood is that signing policies have a size of `43 + 22 * len(voters)`. By mere chance, the result is never a multiple of 32. Nevertheless, even a slight change of length in the signing policy to `42 + 22 * len(voters)` would make this issue likely to occur.

It is worth highlighting as well that the hashing system in the smart contracts has the same collision problem that was described in the `top-level-client` review under ID `TOP-05`.

Recommendation

Make sure the hashing of the two system matches in every scenario.

Status

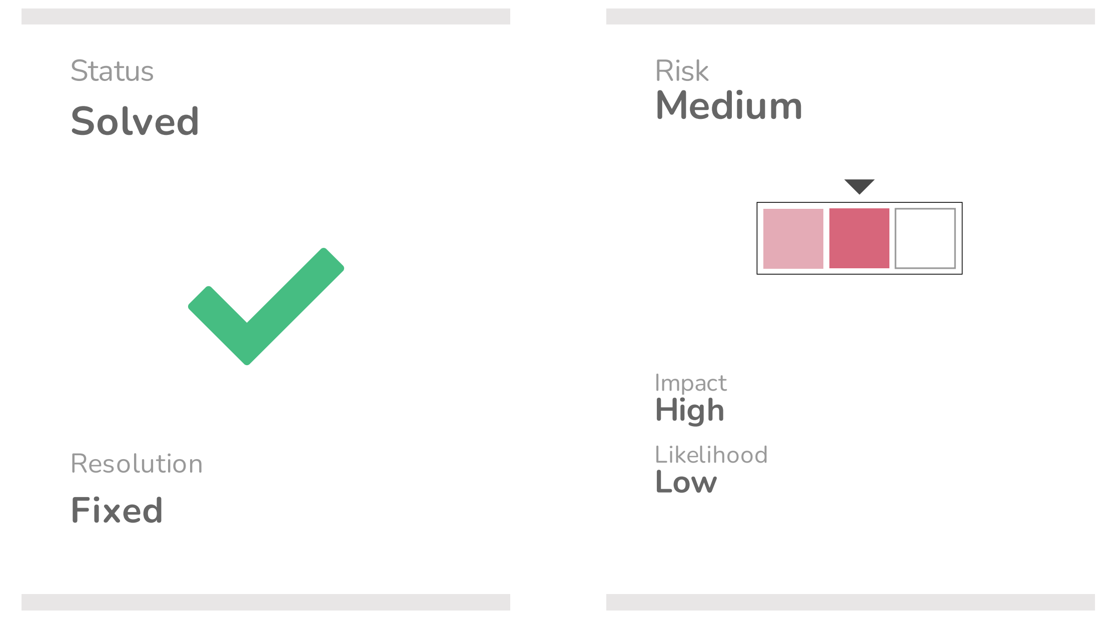
Fixed on commit `421fa87c16af78416bd61bdbc97559006249d07f`.

The hashing function now only considers non-zero remaining bytes.

© Coinspect 2024

FSC-03

Relayers can set an invalid signing policy



Location

`contracts/protocol/implementation/Relay.sol`

Description

Both voters and the `signingPolicySetter` can set an invalid singing policy which would result in service disruption.

In particular, there is no check that the `weights` of the voters are above the `threshold`. If this condition is not met, finalizing new messages will be impossible.

It is worth noting that there are other checks to prevent signing policies from being invalid, for example by making sure that the `voters` array is not empty.

Recommendation

Check the the sum of weights of the policy reached the threshold.

© Coinspect 2024

Status

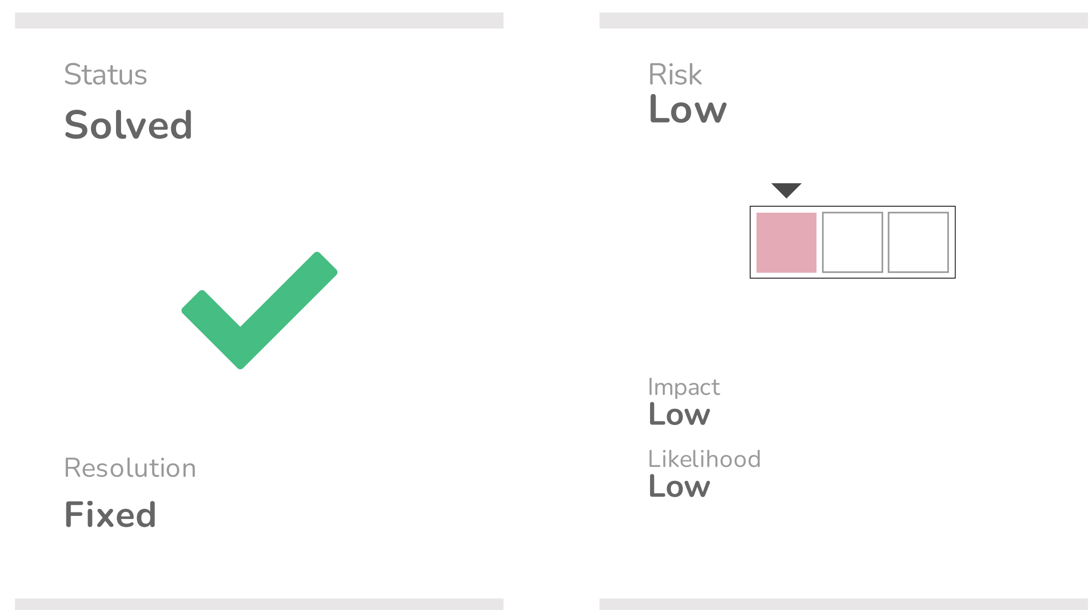
Fixed on commit `88c2a762ce44a2bdc59666d4cada6c216d02c2ac`.

A check to prevent adding policies with less weight than its threshold was added to `setSigningPolicy()`.

© Coinspect 2024

FSC-04

Callers can mistakenly use unsafe random values



Location

```
contracts/protocol/implementation/Submission.sol
contracts/protocol/implementation/Relay.sol
```

Description

The random-providing API of the contracts invites callers to use an unsafe random value, as its main method `getCurrentRandom()` does not indicate that the return value might be unsafe.

For the `Submission.sol` contract, there is a triplet of methods that can return a random value. `getCurrentRandom()`, `getCurrentRandomWithQuality` and `getCurrentRandomWithQualityAndTimestamp()`. None of this methods is labeled as unsafe.

For the `Relay.sol` contract, the only method is `getCurrentRandom()`. While this method returns the information necessary to check if the random is safe to use,

© Coinspect 2024

the caller is still expected to manually check if the returned values are safe.

Recommendation

Provide an API that clearly distinguishes between safe and unsafe variants of the methods.

The safe variants should only return values if they are safe to use by checking interanlly that `isSecureRandom` is set to true.

Unsafe variants should be labeled with a name such as `unsafe` or clearly documented as such and callers can use them if they do not need a random number.

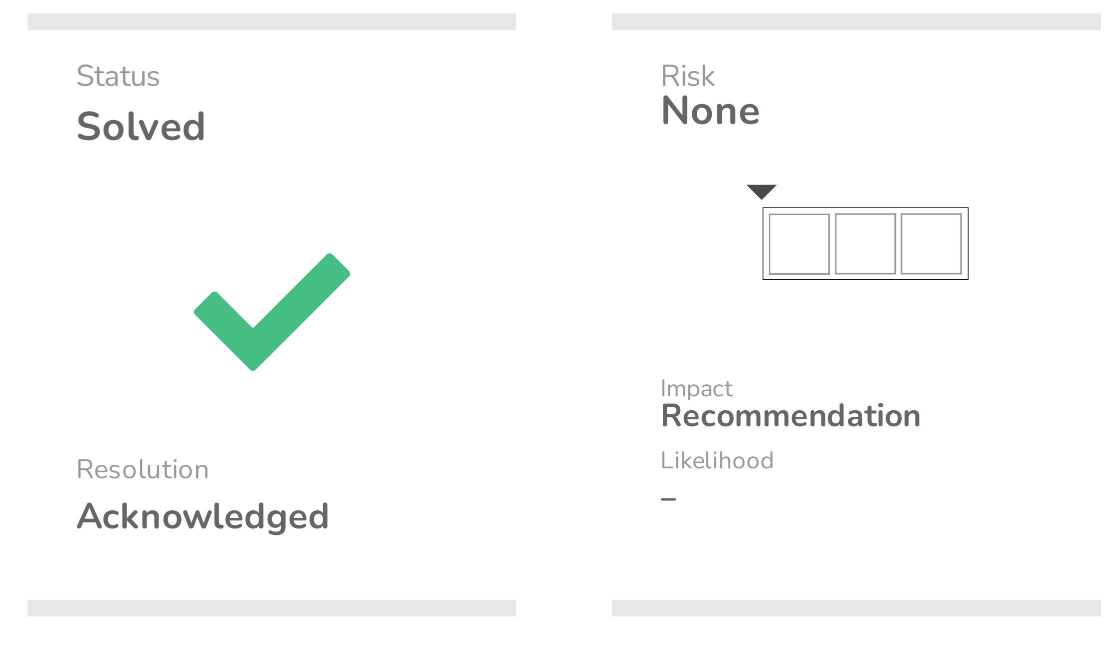
Status

Fixed on commit `103929a40650c5d8c427e97acad88172da4c31a3`. Relevant comments were added to the NatSpec on commit `8b5248d6d3a97a7cfc55f4072f445ac93fe8fc11`.

The function `getCurrentRandom()` now reverts if the random is unsafe. For possibly unsafe call, users can still call `getCurrentRandomWithQuality()`.

FSC-05

Arbitrary values for firstVotingRoundStartTs disrupt the functioning of FTSOs and Stake Mirrorings



Location

`contracts/protocol/implementation/Relay.sol`

Description

A misconfiguration in the deployment of the Relay contract will trigger a revert in other parts of the protocol restricting key functionalities such as publishing FTSO feeds or mirroring stakes.

Additionally, because `relay()` is time-agnostic and depends on round IDs, setting an overly high `firstVotingRoundStartTs` causes a mismatch between this value and the actual start timestamp.

`constructor()`:

```
stateData.firstVotingRoundStartTs = _firstVotingRoundStartTs;
```

When `firstVotingRoundStartTs` is greater than the timestamp, the following function reverts:

```
/**
    * @inheritdoc IRelay
    */
function getVotingRoundId(uint256 _timestamp) external view returns
(uint256) {
    require(_timestamp >= stateData.firstVotingRoundStartTs, "before
the start");
    return (_timestamp - stateData.firstVotingRoundStartTs) /
stateData.votingEpochDurationSeconds;
}
```

By setting an overly high `firstVotingRoundStartTs` value, the FTSO Feed Publisher contract will fail to publish feeds:

```
    function _getMinVotingRoundId() internal view returns(uint256
_minVotingRoundId) {
        uint256 currentVotingRoundId =
relay.getVotingRoundId(block.timestamp);
        if (currentVotingRoundId > feedsHistorySize) {
            _minVotingRoundId = currentVotingRoundId -
feedsHistorySize;
        }
    }
```

Recommendation

Ensure before the deployment that the `firstVotingRoundStartTs` matches the one specified by the initial signing policy.

Proof of Concept

The following test shows how it is impossible to get the current voting round ID after a misconfiguration of the Relay contract. Also, shows how it is possible to make relay calls regardless the first voting round start value.

When trying to get the current `votingRoundId`, the contract reverts.

```
  it("Should relay a message and not fail to get current
votingRoundId", async () => {
```

```
    // This test requires to modify   const firstVotingRoundStartSec =
4294967295; from the setups made in before()
    const messageHash = ProtocolMessageMerkleRoot.hash(messageData);
    const signatures = await generateSignatures(
      accountPrivateKeys,
      messageHash,
      N / 2 + 1
    );

    const relayMessage = {
      signingPolicy: signingPolicyData,
      signatures,
      protocolMessageMerkleRoot: messageData,
    };

    const fullData = RelayMessage.encode(relayMessage);
    const receipt = await web3.eth.sendTransaction({
      from: signers[0].address,
      to: relay.address,
      data: selector + fullData.slice(2),
    })
    await expectEvent.inTransaction(receipt!.transactionHash, relay,
"ProtocolMessageRelayed", {
      protocolId: toBN(messageData.protocolId),
      votingRoundId: toBN(messageData.votingRoundId),
      isSecureRandom: messageData.isSecureRandom,
      merkleRoot: merkleRoot,
    });
    console.log("Gas used:", receipt?.gasUsed?.toString());
    const confirmedMerkleRoot = await
relay.getConfirmedMerkleRoot(messageData.protocolId,
messageData.votingRoundId);
    expect(confirmedMerkleRoot).to.equal(merkleRoot);

    let stateData = await relay.stateData();

expect(stateData.randomNumberProtocolId.toString()).to.be.equal(message
Data.protocolId.toString());

expect(stateData.randomVotingRoundId.toString()).to.be.equal(messageDat
a.votingRoundId.toString());

expect(stateData.isSecureRandom.toString()).to.be.equal(messageData.isS
ecureRandom.toString());

    expect(RelayMessage.decode(fullData)).not.to.throw;
    const decodedRelayMessage = RelayMessage.decode(fullData);

    expect(RelayMessage.equals(relayMessage,
decodedRelayMessage)).to.be.true;

    let blockTimestamp = (await
ethers.provider.getBlock('latest')).timestamp;
    // The following call should not fail under normal conditions
    console.log('Trying to get the current voting round Id...')
    await relay.getVotingRoundId(blockTimestamp)
  });
```

© Coinspect 2024

Status

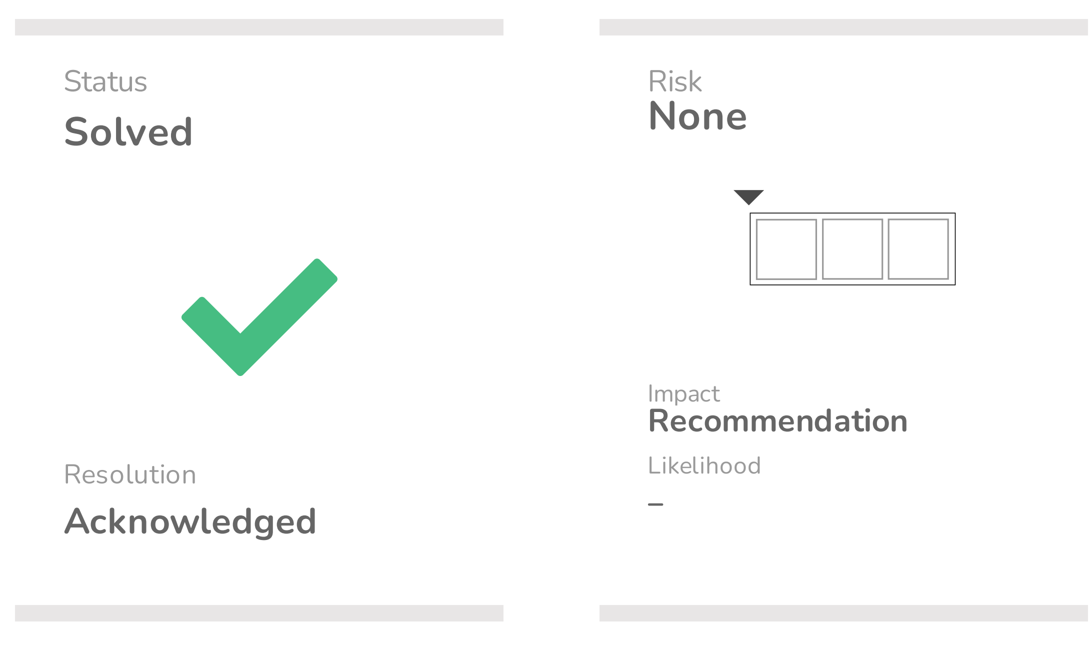
Acknowledged.

The Flare Team recognizes the importance of the configuration settings during the initial deployment. They have committed to implementing stringent precautions to ensure that these settings are carefully managed and correctly implemented to avoid potential issues.

© Coinspect 2024

FSC-06

RewardEpochId size mismatch between contract and top-level-client



Location

`contracts/protocol/implementation/Relay.sol`

Description

The `RewardEpochId` is assumed to be of 3 bytes in the `Relay.sol` contract, but the `top-level-client` encodes it as a `uint32` which occupies 4 bytes.

The mismatch is apparent when comparing the mask used in `Relay.sol` to extract it and the encoding code of the `top-level-client`.

```
uint256 private constant MD_MASK_rewardEpochId = 0xffffff;
```

```
    epochBytes := shared.Uint32toBytes(uint32(s.rewardEpochId))
```

© Coinspect 2024

This issue is informational as the mismatch would only be apparent for epoch IDs of size bigger than `2**24-1`.

Recommendation

Make sure that the off chain component and the on chain contract agree on the size of the structures they must share.

Status

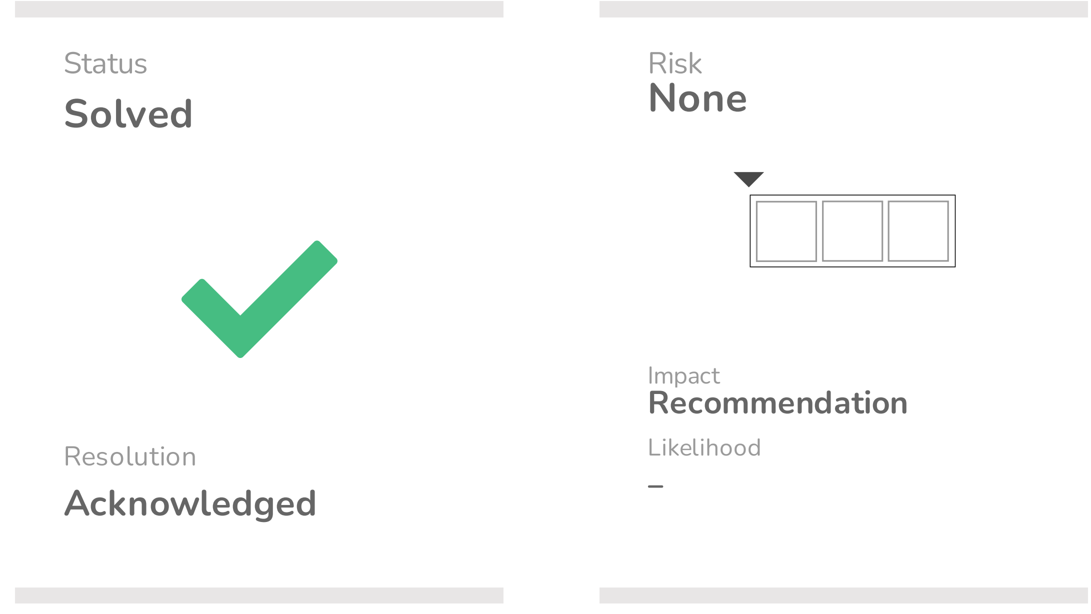
Acknowledged.

The Flare Team has acknowledged this issue and is considering the addition of a check statement in the Flare Systems Client. This adjustment would occur when the size required to accommodate the `rewardEpochId` exceeds current specifications. However, it's important to note that the current 3-byte size is expected to be sufficient for the system's anticipated lifetime.

© Coinspect 2024

FSC-07

Attacker could use submitAndPassContract to waste resources



Location

```
contracts/protocol/implementation/Submission.sol
```

Description

The Submission contract's method `submitAndPass` calls a configurable method of a configurable address. Because these calls are subsidized, if the called method returns an arbitrary amount of bytes, an attacker could use this subsidy to force the contract to use a large amount of gas without incurring in a big cost themselves.

This transactions would then have to be replicated by all peers.

Recommendation

Make sure the Governance is aware of this risk and that the selected call through `submitAndPass` cannot return arbitrary bytes amounts.

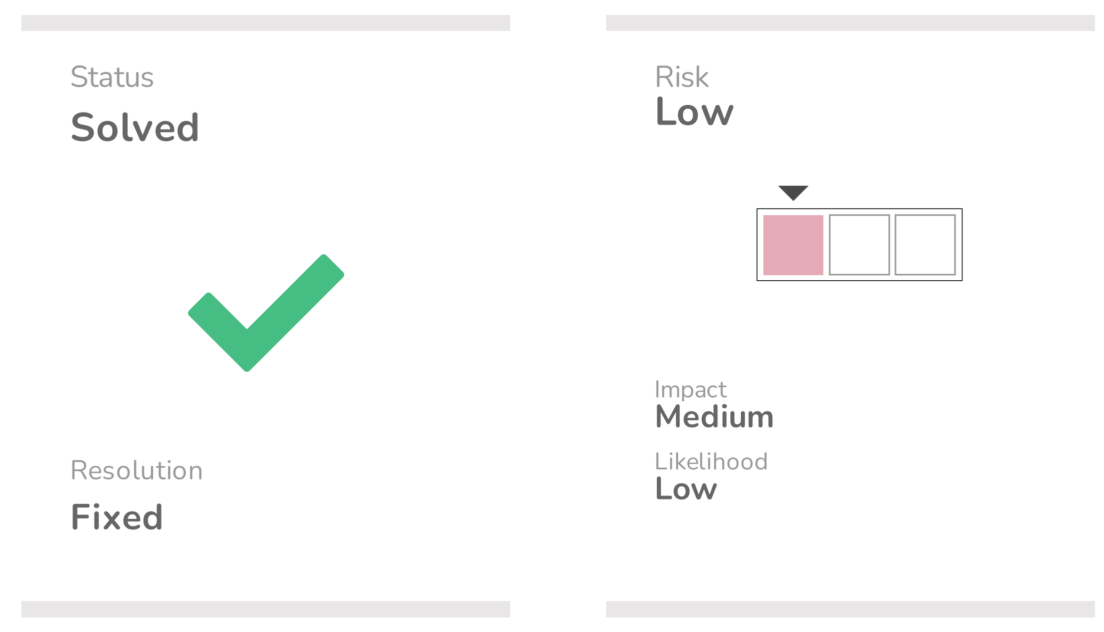
Status

Acknowledged.

The Flare Team stated that they are aware of the underlying risks related to the `submitAndPass` contract, and that they plan to provide its implementation when the FTSO Fast Updates system is reviewed.

FSC-08

Permanent data staleness threatens financial stability of third parties



Location

`contracts/ftso/implementation/FtsoFeedPublisher.sol`

Description

The current FTSO feed can be manipulated to always return the same data, potentially disrupting how third parties rely on FTSO data for financial calculations based on a price feed. Specifically, the `getCurrentFeed()` method could be forced to consistently return the same data, with no possibility of updating it in production other than creating a new feed for this asset.

When publishing feed data, this check determines whether the `lastFeeds` mapping should be updated:

```
            bool addLastFeed = feed.votingRoundId >
lastFeeds[feed.name].votingRoundId;
```

© Coinspect 2024

The publisher can pass an overly high value for `feed.votingRoundId`, setting the latest feed's data:

```
                if (addLastFeed) {
                    lastFeeds[feed.name] = feed;
                }
```

Once a feed with a voting round id value set to `type(uint32).max` (as voting round ids are 32-bit variables) is established, no subsequent voting round ID will be greater than this value. Consequently, `addLastFeed` will always be false for any other round id, preventing updates to the `lastFeeds` mapping.

Feeds can be updated by anyone passing the feed proofs or by the feed publisher (proofs are not required).

Recommendation

Ensure that last feeds can be updated up to a maximum number of rounds in the future.

Status

Fixed on commit `800405c8d6fa607a20ab62099e45bed91c32d07c`.

A check to verify that the round id is not in the future was added to the FTSO feed publish process.

Proof of Concept

The following scenario demonstrates how a feed can be made permanently stale. This is achieved by publishing a feed with an excessively high voting round ID. Subsequent assertions illustrate that even after attempting to publish a new feed, the value reported remains unchanged from the original.

Output

```
[FAIL. Reason: Assertion failed.] testCoinspectMakeFeedStale() (gas:
171687)
Logs:
```

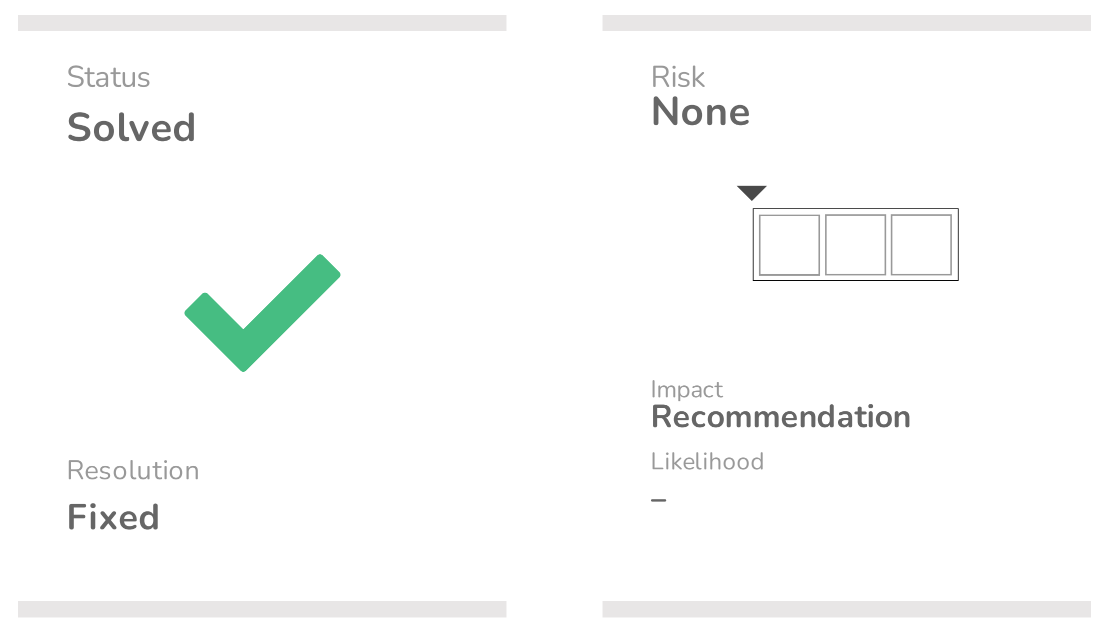
```
  Error: a == b not satisfied [uint]
        Left: 4294967295
       Right: 3
  Error: a == b not satisfied [int]
        Left: 100
       Right: 200
  Error: a == b not satisfied [uint]
        Left: 1000
       Right: 2000
  Error: a == b not satisfied [int]
        Left: 2
       Right: 4
```

Test

```
    function testCoinspectMakeFeedStale() public {
        testSetFeedsPublisher();
        _mockGetVotingRoundId(block.timestamp, 4);

        uint32 roundId = type(uint32).max;
        IFtsoFeedPublisher.Feed memory feed =
            IFtsoFeedPublisher.Feed(roundId, feedName1, int32(100),
uint16(1000), int8(2));

        IFtsoFeedPublisher.Feed[] memory feeds = new
IFtsoFeedPublisher.Feed[](1);
        feeds[0] = feed;
        vm.prank(feedsPublisher);
        ftsoFeedPublisher.publishFeeds(feeds);

        IFtsoFeedPublisher.Feed memory getFeed =
ftsoFeedPublisher.getCurrentFeed(feedName1);
        assertEq(getFeed.votingRoundId, roundId);
        assertEq(getFeed.name, feedName1);
        assertEq(getFeed.value, int32(100));
        assertEq(getFeed.turnoutBIPS, uint16(1000));
        assertEq(getFeed.decimals, int8(2));

        // Then, when setting for the next epoch, getCurrentFeed will
still return the same
        roundId = 3;
        feed = IFtsoFeedPublisher.Feed(roundId, feedName1, int32(200),
uint16(2000), int8(4));
        feeds[0] = feed;
        vm.prank(feedsPublisher);
        ftsoFeedPublisher.publishFeeds(feeds);

        getFeed = ftsoFeedPublisher.getCurrentFeed(feedName1);
        // all the following assertions will fail because it will still
return the previous feed
        assertEq(getFeed.votingRoundId, roundId);
        assertEq(getFeed.name, feedName1);
        assertEq(getFeed.value, int32(200));
        assertEq(getFeed.turnoutBIPS, uint16(2000));
        assertEq(getFeed.decimals, int8(4));
    }
```

FSC-09

Inflation receiver childs might not implement key functions



Location

`contracts/inflation/implementation/InflationReceiver.sol`

Description

Declaring virtual functions without providing an implementation increases the risk that contracts extending the `InflationReceiver` might not override these functions with the necessary logic. This oversight is possible because the contract will still compile without these implementations.

The `InflationReceiver` contract, which handles inflation proceeds, currently implements two key virtual functions as empty instead of leaving them purely abstract:

```
/**
 * Method that is called when new daily inflation is authorized.
 */
```

```
    function _setDailyAuthorizedInflation(uint256 _toAuthorizeWei)
internal virtual {}

    /**
     * Method that is called when new inflation is received.
     */
    function _receiveInflation() internal virtual {}
```

Those two functions allow the inheritor to execute further logic when receiving inflation proceeds or setting the daily authorization inflation.

Recommendation

Make those two functions abstract. This will require overriding and implementing them when inheriting the abstract contract.

Status

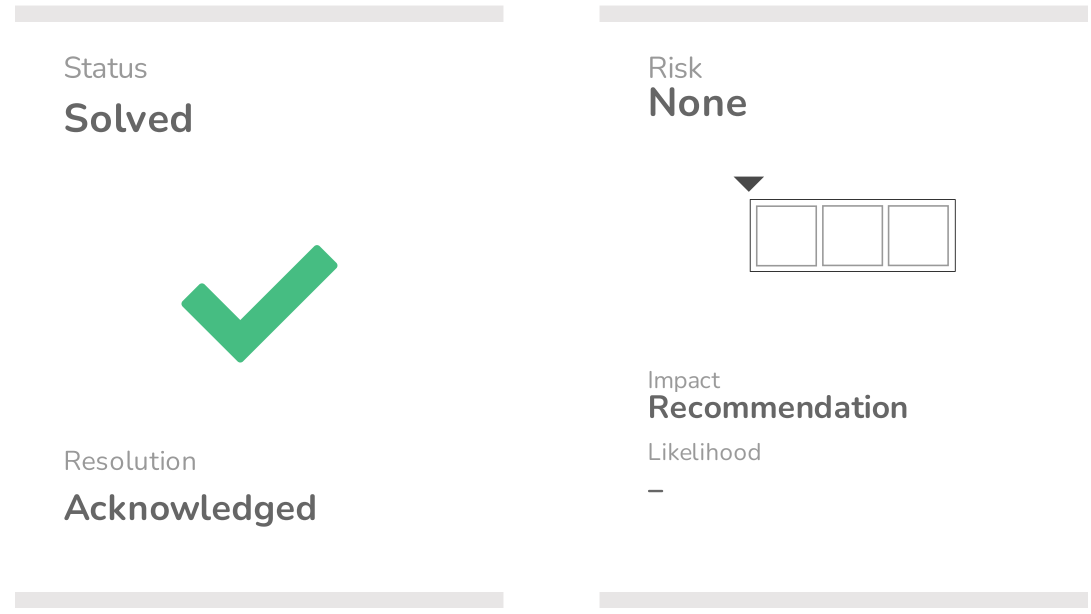
Fixed on commit `481fe3e0d263286f0fe39f36882a5aa8d0264e59`.

The mentioned functions are now abstract.

© Coinspect 2024

FSC-10

Voters can have zero voting power after normalization



Location

contracts/protocol/implementation/VoterRegistry.sol

Description

Voting weight snapshots for a signing policy can yield in zero voting power for voters that have less weight, disrupting the weight accumulation when signing for rewards.

When initializing the next singning policy, a snapshot of each voter's weight is made via `createSigningPolicySnapshot()`. This process also normalizes the weight in a `uint16` base:

```
    _normalisedWeights[i] = uint16((weights[i] * UINT16_MAX) /
weightsSum); // weights[i] <= weightsSum
    _normalisedWeightsSum += _normalisedWeights[i];
```

© Coinspect 2024

When registering a voter it can be seen that each user's weight is a `uint256` number:

```
    uint256 weight =
flareSystemsCalculator.calculateRegistrationWeight(_voter,
_rewardEpochId, votePowerBlock);
```

Voters with less weight might end up having zero voting power after normalization, making their votes worthless when signing for rewards via `FlareSystemsManager.signRewards()`:

```
        (address voter, uint16 weight) =
            voterRegistry.getVoterWithNormalisedWeight(_rewardEpochId,
signingPolicyAddress); // <--- weight could be zero
        require(voter != address(0), "signature invalid");
        require(state.rewardVotes[messageHash].voters[voter].signTs ==
0, "voter already signed");
        // save voter's timestamp and block number
        state.rewardVotes[messageHash].voters[voter] =
VoterData(block.timestamp.toUint64(), block.number.toUint64());
        // check if signing threshold is reached
        bool thresholdReached =
state.rewardVotes[messageHash].accumulatedWeight + weight >
state.threshold;
```

Recommendation

Consider using a bigger data type for the normalized weights.

Status

Acknowledged.

The Flare Team has acknowledged this scenario, noting that the chosen normalization/scaling factor was specifically designed to minimize relaying costs.

Proof of Concept

This scenario demonstrates a situation where only one account accrues all the normalized voting weight, effectively leaving other accounts with zero normalized weight. It involves creating a snapshot for a signing policy, which then normalizes each user's weight and displays the outcome.

Prerequisites

The following modifications were made to the internal function `_createInitialVoters()` in `VoterRegistry.t.sol`, allowing for the setup of different starting voting powers for each user:

```
    // weights
    // @Coinspect Review modifications
    uint256 initialVoterWeight = (i == _num - 1) ? type(uint32).max :
10000 * (i + 1);
    console.log("Assigning weight: %s", initialVoterWeight);
    initialVotersWeights.push(initialVoterWeight);
```

Output

```
  Assigning weight: 10000
  Assigning weight: 20000
  Assigning weight: 30000
  Assigning weight: 4294967295

 Getting normalized weights calling  getVoterWithNormalisedWeight()
  Norm Weight for user: 0x785E2a241F1e59c0264e0Ae81CaD20F178919efd = 0
  Norm Weight for user: 0x9A346D6F41D268704c28c396Aa81b14683223653 = 0
  Norm Weight for user: 0xC5aFb58ba22614335242A1e57bCcf0bFD97F15eD = 0
  Norm Weight for user: 0x826c9ab8aF8944f59e96D7d524a3F76c2AfA900c =
65534
```

Test

```
    function testCoinspect_ZeroVotingPowerAfterNormalization() public {
        // derived from
        IVoterRegistry.Signature memory signature;

        _mockGetCurrentEpochId(0);
        _mockGetVoterAddressesAt();
        _mockGetPublicKeyOfAt();
        _mockGetVoterRegistrationData(10, true);
        _mockVoterWeights();
        vm.prank(mockFlareSystemsManager);

voterRegistry.setNewSigningPolicyInitializationStartBlockNumber(1);

        uint256 weightsSum = 0;
        for (uint256 i = 0; i < initialVoters.length; i++) {
            signature = _createSigningPolicyAddressSignature(i, 1);
            voterRegistry.registerVoter(initialVoters[i], signature);
            weightsSum += initialVotersWeights[i];
        }

        // create signing policy snapshot
        vm.mockCall(
```

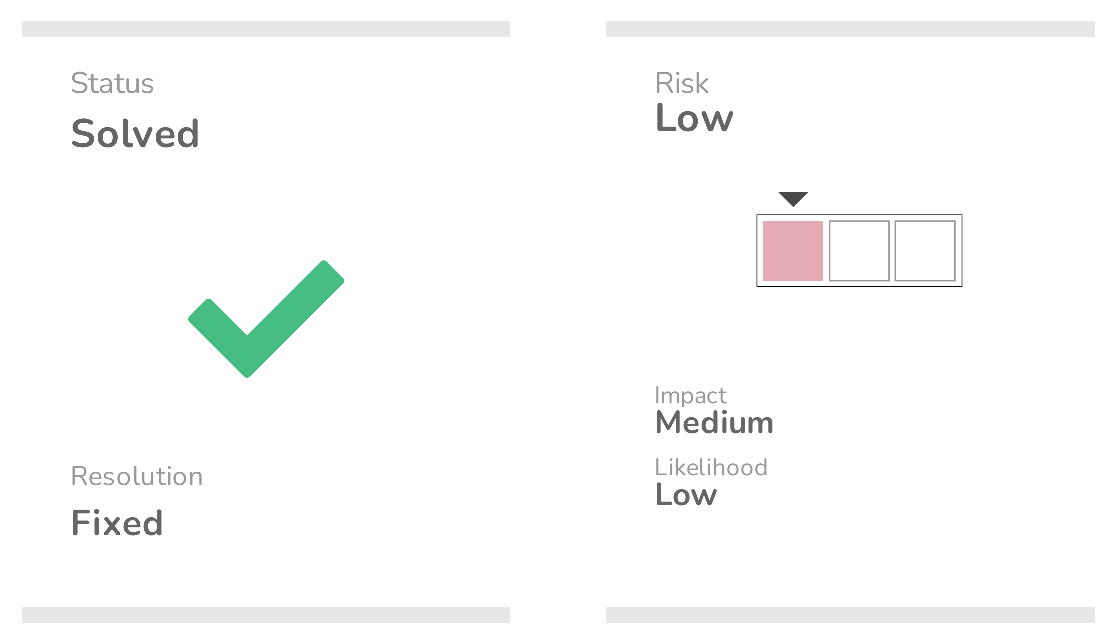
```
            mockEntityManager,
            abi.encodeWithSelector(
                IIEntityManager.getSigningPolicyAddresses.selector,
                initialVoters,

voterRegistry.newSigningPolicyInitializationStartBlockNumber(1)
            ),
            abi.encode(initialSigningPolicyAddresses)
        );
        vm.mockCall(
            mockEntityManager,
            abi.encodeWithSelector(
                IIEntityManager.getPublicKeys.selector,
                initialVoters,

voterRegistry.newSigningPolicyInitializationStartBlockNumber(1)
            ),
            abi.encode(initialPublicKeyParts1, initialPublicKeyParts2)
        );
        vm.prank(mockFlareSystemsManager);
        (address[] memory signPolAddresses, uint16[] memory
normWeights, uint16 normWeightsSum) =
            voterRegistry.createSigningPolicySnapshot(1);

        assertEq(initialSigningPolicyAddresses.length,
signPolAddresses.length);
        uint16 sum = 0;
        uint256 voterWeight;
        uint16 normVoterWeight;
        address _gVWNWVoter;
        uint16 _gVWNWNormalizedWeigt;

        console.log("\n Getting normalized weights calling
getVoterWithNormalisedWeight()");
        for (uint256 i = 0; i < initialVoters.length; i++) {
            assertEq(signPolAddresses[i],
initialSigningPolicyAddresses[i]);
            voterWeight = initialVotersWeights[i];
            normVoterWeight = uint16(voterWeight * UINT16_MAX /
weightsSum);
            assertEq(normWeights[i], normVoterWeight);
            sum += normVoterWeight;

            vm.mockCall(
                mockEntityManager,
                abi.encodeWithSelector(

IEntityManager.getVoterForSigningPolicyAddress.selector,
                    initialSigningPolicyAddresses[i],

voterRegistry.newSigningPolicyInitializationStartBlockNumber(1)
                ),
                abi.encode(initialVoters[i])
            );
            (_gVWNWVoter, _gVWNWNormalizedWeigt) =
                voterRegistry.getVoterWithNormalisedWeight(1,
initialSigningPolicyAddresses[i]);
            console.log("Norm Weight for user: %s = %s", _gVWNWVoter,
_gVWNWNormalizedWeigt);
        }
```

```
        assertEq(sum, normWeightsSum);

        (uint128 _sum, uint16 _normSum, uint16 _normSumPub) =
voterRegistry.getWeightsSums(1);
        assertEq(_sum, weightsSum);
        assertEq(_normSum, normWeightsSum);
        // only voter0 registered public key
        assertEq(_normSumPub, uint16(initialVotersWeights[0] *
UINT16_MAX / weightsSum));
    }
```

© Coinspect 2024

FSC-11

Voter cap mismatch endangers system's voting integrity



Location

`contracts/protocol/implementation/FlareSystemsManager.sol`

Description

The maximum amount of voters allowed for registration can be smaller than the minimum number of votes checked on the System Manager, as a result, voting registration will always be enabled until the minimum duration in blocks or time passes.

In the context of the System Manager, it is checked that the voters registered exceeds the minimum number of voters:

```
    function _isVoterRegistrationEnabled(uint256 _rewardEpoch,
RewardEpochState storage _state)
        internal
        view
        returns (bool)
```

```
    {
        return _state.randomAcquisitionEndTs != 0
            && (
                block.timestamp <= _state.randomAcquisitionEndTs +
voterRegistrationMinDurationSeconds
                    || block.number <= _state.randomAcquisitionEndBlock
+ voterRegistrationMinDurationBlocks
                    ||
voterRegistry.getNumberOfRegisteredVoters(_rewardEpoch) <
signingPolicyMinNumberOfVoters
            );
    }
```

However, in the Voter Registry there is a cap for the amount of maximum voters that can be registered:

```
        if (length < maxVoters) {
            // we can just add a new one
            votersAndWeights.voters.push(_voter);
            votersAndWeights.weights[_voter] = weight;
        } else {
            // find minimum to kick out (if needed)
            ...
        }
```

The described scenario could happen since there are no checks that guarantee that the `signingPolicyMinNumberOfVoters` is smaller than `voterRegistry.maxVoters`.

Recommendation

Check that the `signingPolicyMinNumberOfVoters` is smaller than `voterRegistry.maxVoters` when updating the setting.

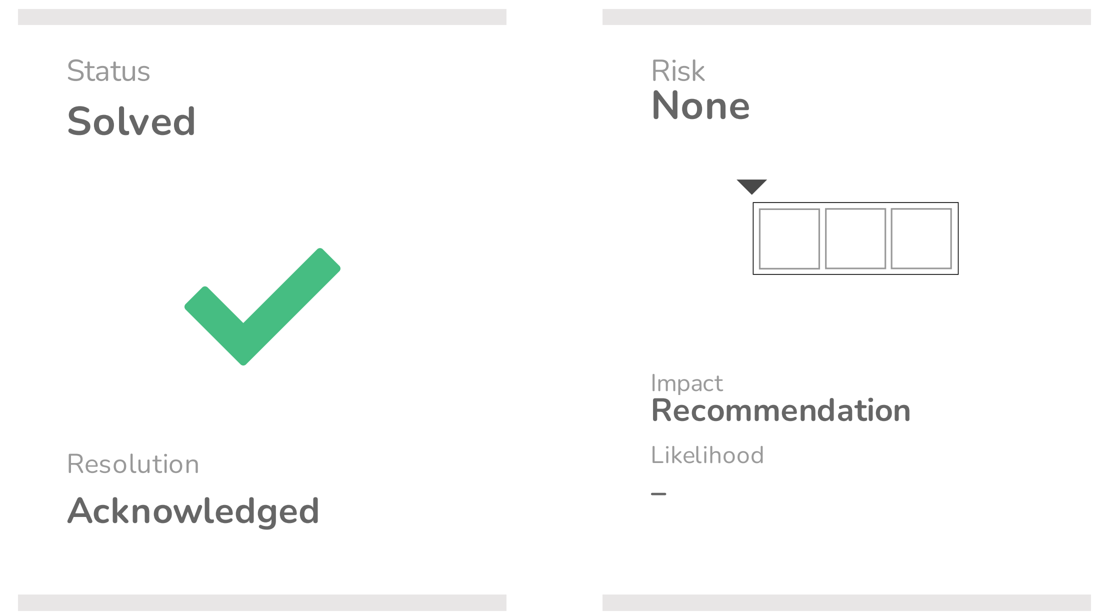
Status

Fixed on commit `8e4be77f53e603453ebe2ba06c189a7b63420f18`.

A check preventing the mentioned scenario was added into the `VoterRegistry` and `SystemManager` contracts.

FSC-12

Governance calls can be return bombed



Location

`contracts/governance/implementation/GovernedBase.sol`

Description

Governance calls copy all the return data into memory when processing a revert message, as a result, outstanding amounts of gas might be consumed if the callee return-bombs the caller (Governance):

```
    function _passReturnOrRevert(bool _success) private pure {
        // pass exact return or revert data - needs to be done in
assembly
        //solhint-disable-next-line no-inline-assembly
        assembly {
            let size := returndatasize()
            let ptr := mload(0x40)
            mstore(0x40, add(ptr, size))
            returndatacopy(ptr, 0, size)
            if _success {
                return(ptr, size)
            }
```

```
            revert(ptr, size)
        }
    }
```

As all the return data is copied into memory, it could increase the gas spent by expanding the memory used with a long revert message.

Recommendation

Add a return data size limit. Nomad implemented this in their ExcessivelySafeCall Repository.

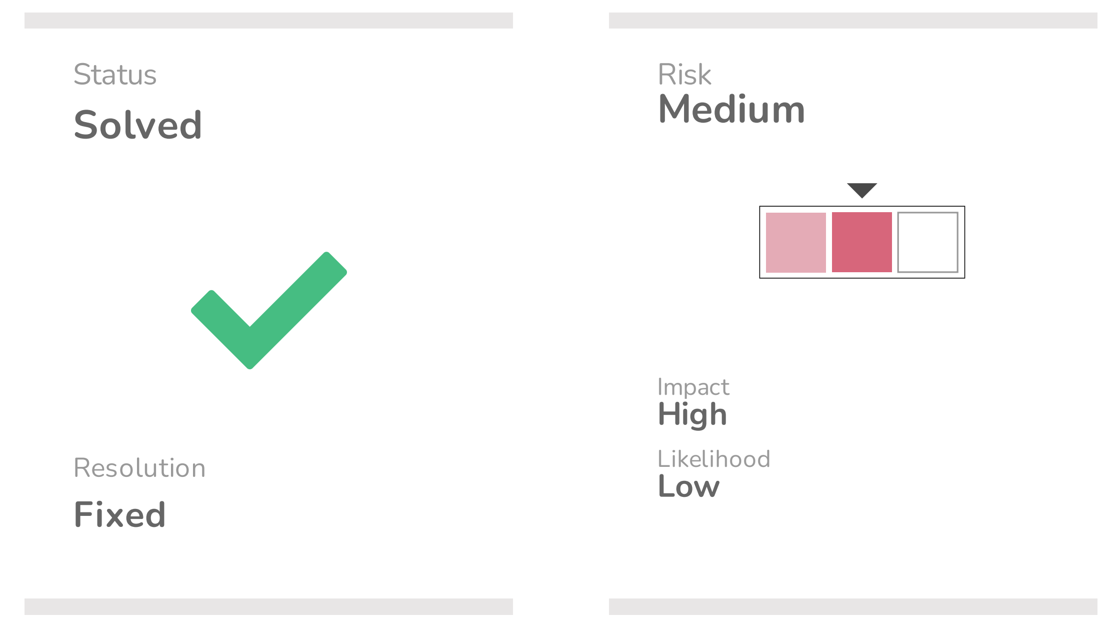
Status

Acknowledged.

The Flare Team stated that there would be no chance of return-bombs since all contracts planned to be called from are known and trusted.

FSC-13

Attacker can gain unfair voting power by timing first daemonization



Location

`contracts/protocol/implementation/FlareSystemsManager.sol`

Description

Attackers can try to predict when the first call to `daemonize()` will be made after the contract's deployment and manipulate their stakes, delegations, or `WNat` balance, to unfairly increase their initial voting power.

The Systems Manager contract uses `initialRandomVotePowerBlockSelectionSize` to set the vote power block when the random number was unsafe. This variable is meant to be used only in the beginning to select the initial vote power block:

```
    function _selectVotePowerBlock(uint24 _nextRewardEpochId, uint256
_random)
        internal view
        returns(uint64 _votePowerBlock)
    {
```

© Coinspect 2024

```
        // randomTs > state.randomAcquisitionStartTs && isSecureRandom
== true
        uint64 startBlock = rewardEpochState[_nextRewardEpochId -
1].randomAcquisitionStartBlock;
        // 0 < endBlock < block.number
        uint64 endBlock =
rewardEpochState[_nextRewardEpochId].randomAcquisitionStartBlock;
        uint64 numberOfBlocks;
        if (startBlock == 0) {
            // endBlock > 0 && initialRandomVotePowerBlockSelectionSize
> 0
            numberOfBlocks = Math.min(endBlock,
initialRandomVotePowerBlockSelectionSize).toUint64();
        } else {
            // endBlock > startBlock
            numberOfBlocks = endBlock - startBlock;
        }

        //slither-disable-next-line weak-prng
        uint256 votePowerBlocksAgo = _random % numberOfBlocks; //
numberOfBlocks > 0
        _votePowerBlock = endBlock - votePowerBlocksAgo.toUint64();
    }
```

Since the value for initialRandomVotePowerBlockSelectionSize according to the deployment config files (deployment/utils/deploy-contracts.ts) is 1, it means that votePowerBlocksAgo will always be one block before the first daemonization's block, which is set as randomAcquisitionStartBlock.

The test provided by the project, testSelectVotePowerBlockUnsecureRandom, demonstrates how the first vote power block can be easily predicted by knowing the block number at which the first daemonization occurred (block 199 in this case). Additionally, the initial configurations used in this test (initialRandomVotePowerBlockSelectionSize set to 5) do not align with the configurations intended for deployment (initialRandomVotePowerBlockSelectionSize set to 1).

Test provided by the project

```
    // use current vote power block; initial reward epoch -> use
unsecure random
    function testSelectVotePowerBlockUnsecureRandom() public {
        uint64 currentTime = uint64(block.timestamp) +
REWARD_EPOCH_DURATION_IN_SEC - 2 * 3600;
        vm.warp(currentTime);

        _mockToSigningPolicyHash(1, bytes32(0));

        // start random acquisition
        vm.roll(199);
        vm.startPrank(flareDaemon);
        flareSystemsManager.daemonize();
```

```
        // select vote power block
        // move to the end of acquisition period and don't get secure
random
        vm.roll(block.number + 15000 + 1);
        vm.warp(block.timestamp + uint64(8 * 60 * 60 + 1));
        vm.mockCall(
            mockRelay,
            abi.encodeWithSelector(IRelay.getRandomNumber.selector),
            abi.encode(123, false, currentTime + 1)
        );

        vm.expectEmit();
        emit VotePowerBlockSelected(1, 196, uint64(block.timestamp));
        flareSystemsManager.daemonize();
        // voter registration started
        // endBlock = 199, _initialRandomVotePowerBlockSelectionSize =
5
        // numberOfBlocks = 5, random (=123) % 5 = 3 -> vote power
block = 199 - 3 = 196
        assertEq(flareSystemsManager.getVotePowerBlock(1), 196);
    }
```

Recommendation

Coinsider restricting sensitive actions that depend on the vote power block while the system is bootstrapping.

Status

Fixed on commit `e577e9c6c230f6e8ea202d340545219d06284756`.

The Flare Team provided a script that will run to provide the initial random number as well as including each chain's config parameters, mitigating the risk of this issue.

FSC-14

Evil voters can steal voting power through node registration

Location

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

Description

Attackers can target users mirroring a node in the stake mirroring service and register it directly in the Entity Manager. This grants attackers with outstanding voting power they don't own, effectively stealing it from unsuspecting users.

The node ID registration process is needed when a user wants to claim a node as theirs, and can only be done once:

```
    function registerNodeId(bytes20 _nodeId) external {
        require(nodeIdRegistered[_nodeId].addressAtNow() == address(0),
"node id already registered");
        register[msg.sender].nodeIds.addRemoveNodeId(_nodeId, true,
maxNodeIdsPerEntity);
```

```
        nodeIdRegistered[_nodeId].setAddress(msg.sender);
        emit NodeIdRegistered(msg.sender, _nodeId);
    }
```

Having nodes registered enables a new stream of accruing voting weight accounted by the System Calculator:

```
        bytes20[] memory nodeIds = entityManager.getNodeIdsOfAt(_voter,
_votePowerBlockNumber);
        uint256[] memory nodeWeights;
        if (address(pChainStakeMirror) != address(0)) {
            nodeWeights = pChainStakeMirror.batchVotePowerOfAt(nodeIds,
_votePowerBlockNumber);
            for (uint256 i = 0; i < nodeWeights.length; i++) {
                if (_rewardEpochId >=
voterRegistry.chilledUntilRewardEpochId(nodeIds[i])) {
                    _registrationWeight += nodeWeights[i];
                } else {
                    nodeWeights[i] = 0;
                }
            }
        }
```

The underlying node's voting power is retrieved from the stake mirroring contract with `batchVotePowerOfAt(nodeIdList, _blockNumber)`. This function returns the vote power of a certain array of nodes, regardless of the user that mirrored it. The mirroring contract has another method that returns the voting power of a node but filters the owner that made its mirroring, `votePowerFromToAt(_owner, _nodeId, _blockNumber)` (not used in any calculation of the System Calculator).

As a consequence, the evil voter doing this registering attack receives the voting power yielded by a node they do not own, for the time that it's still active in the mirroring service.

It is worth mentioning that attackers can do this by scanning the victim's `StakeData` (containing the `nodeId`) passed when mirroring a stake. For those stakes that are already mirrored but not registered in the Entity Manager, attackers can rely on either scanning the chain's state or filtering events to get those node IDs.

Recommendation

Ideally, ensure that node IDs can only be registered by their owners by requiring a proof of posession of their node IDs. This would likely require modifying the Flare node signing mechanism or update the Flare's VM with yet novel BLS precompiles.

© Coinspect 2024

Alternatively, clearly document how the staking and registering process should be made and how it mitigates the risks mentioned on this issue. Also, consider mentioning how the proof of possession process might change in the future if support for new precompiles or signing mechanisms are added to the Flare Ecosystem.

Status

Partially fixed on commit `ffe36b1cd2419486447ae66d116e7f8ff5bc9984`.

A Node Possession Verifier contract was added aiming to fix this issue. Coinspect found that the minimum key length required was too small to ensure authentication. Additionally, there is no domain separator in the signature.

The contract is also quite complex, as the ASN.1 format chosen for the key needs to be interpreted in Solidity, adding to the attack surface and the gas costs for users. This is likely due to the usage of a RSA key intended to be used in other contexts, which as repurposed to be useful for proof of possession.

A way to further mitigate the risks associated with the complexity of this solution is to only use the signatures in case of a dispute for some address: in the best case scenario, the documentation should guide users towards using the system in a non-risky way. But if a mistake happens, users can reclaim an address by showing their signature.

Fixed on commit `c5102126d9e6744552dcab2d463b6de68cd3ac11`.

The minimum certificate size is now `512`. Additionally, `32-byte` zero prefixing was added to the message. Additionally, the Flare Team stated that they are aware of the underlying risks of performing these type of validations on-chain. Because of this, they rely on the fact that the Node Posession Verifier contract can be replaced by the Governance using a setter in the `EntityManager`.

Proof of Concept

The following tests shows how the `user2` registers the `nodeId1`, property of `user1`. As a result, the `user2` registers the `nodeId1` as theirs, leaving the `user1` empty handed.

Output

```
  User1: 0x29E3b139f4393aDda86303fcdAa35F60Bb7092bF
  User2: 0x537C8f3d3E18dF5517a58B3fB9D9143697996802

Node Id1 to register:
  0x01398d9895c8b40fed627cb1663f4b4a0c7d4ce3


  Error: C1 Failed
  Error: a == b not satisfied [uint]
        Left: 0
       Right: 1
  Error: C3 Failed
  Error: a == b not satisfied [uint]
        Left: 0
       Right: 1
  Error: C5 Failed
  Error: a == b not satisfied [address]
        Left: 0x537c8f3d3e18df5517a58b3fb9d9143697996802
       Right: 0x29e3b139f4393adda86303fcdaa35f60bb7092bf
```

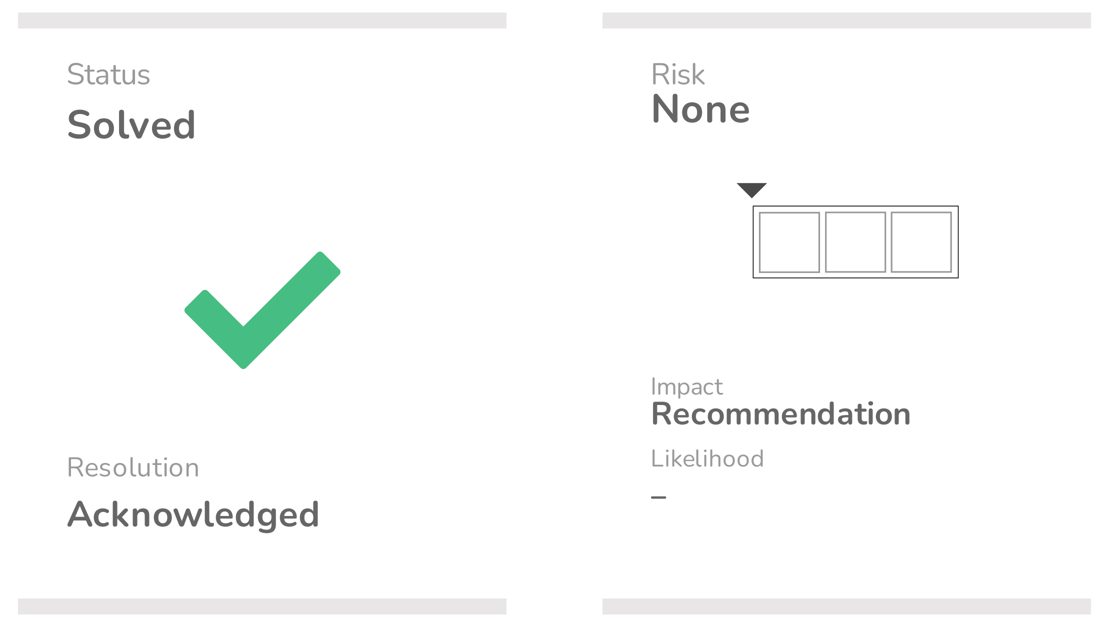
Test

```
    function testCoinspectRegisterOthersNodeId() public {
        vm.roll(100);
        assertEq(entityManager.getNodeIdsOfAt(user1,
block.number).length, 0);
        assertEq(entityManager.getNodeIdsOf(user1).length, 0);
        assertEq(entityManager.getVoterForNodeId(nodeId1,
block.number), address(0));

        console.log("User1: %s", user1);
        console.log("User2: %s", user2);

        console.log("\nNode Id1 to register:");
        console2.logBytes20(nodeId1);
        console.log("\n");

        // User 2 registers the nodeId1 as theirs
        vm.expectEmit();
        emit NodeIdRegistered(user2, nodeId1);
        vm.roll(101);
        vm.prank(user2);
        entityManager.registerNodeId(nodeId1);

        // When the call of user 1 is executed, it reverts
        // as it is trying to register the same node id again
        vm.startPrank(user1);
        vm.expectRevert("node id already registered");
        entityManager.registerNodeId(nodeId1);
        vm.stopPrank();

        // No node was registered for User1
        bytes20[] memory nodeIds = entityManager.getNodeIdsOfAt(user1,
101);
        assertEq(nodeIds.length, 1, "C1 Failed");
        // assertEq(nodeIds[0], nodeId1);
```

```
        assertEq(entityManager.getNodeIdsOfAt(user1, 100).length, 0,
"C2 Failed");
        assertEq(entityManager.getNodeIdsOf(user1).length, 1, "C3
Failed");
        assertEq(entityManager.getVoterForNodeId(nodeId1, 100),
address(0), "C4 Failed");
        assertEq(entityManager.getVoterForNodeId(nodeId1,
block.number), user1, "C5 Failed");

        // The Node1 is registered by the User 2 (the following
assertions will pass)
        nodeIds = entityManager.getNodeIdsOfAt(user2, 101);
        assertEq(nodeIds.length, 1);
        assertEq(nodeIds[0], nodeId1);
        assertEq(entityManager.getNodeIdsOfAt(user2, 100).length, 0);
        assertEq(entityManager.getNodeIdsOf(user2).length, 1);
        assertEq(entityManager.getVoterForNodeId(nodeId1, 100),
address(0));
        assertEq(entityManager.getVoterForNodeId(nodeId1,
block.number), user2);
    }
```

FSC-15

Daemonize calls revert with high reward expiry offsets



Location

`contracts/protocol/implementation/FlareSystemsManager.sol`

Description

Setting an overly high value for the reward expiry offset will trigger a revert on every `daemonize` call that attempts to cleanup expired reward epochs.

The `updateSettings` function allows the governance to modify how Flare's core operates. Checks are in place to prevent assigning forbidden or out-of-bounds values to important variables among those that can be updated. This is not the case for `rewardExpiryOffsetSeconds`, that can take any value up to `type(uint32).max`:

```
    function _updateSettings(Settings memory _settings) internal {
        require(_settings.signingPolicyThresholdPPM <= PPM_MAX,
 "threshold too high");
```

```
        require(_settings.signingPolicyMinNumberOfVoters > 0, "zero
voters");

        randomAcquisitionMaxDurationSeconds =
_settings.randomAcquisitionMaxDurationSeconds;
        randomAcquisitionMaxDurationBlocks =
_settings.randomAcquisitionMaxDurationBlocks;
        newSigningPolicyInitializationStartSeconds =
_settings.newSigningPolicyInitializationStartSeconds;
        newSigningPolicyMinNumberOfVotingRoundsDelay =
_settings.newSigningPolicyMinNumberOfVotingRoundsDelay;
        rewardExpiryOffsetSeconds =
_settings.rewardExpiryOffsetSeconds;
        voterRegistrationMinDurationSeconds =
_settings.voterRegistrationMinDurationSeconds;
        voterRegistrationMinDurationBlocks =
_settings.voterRegistrationMinDurationBlocks;
        submitUptimeVoteMinDurationSeconds =
_settings.submitUptimeVoteMinDurationSeconds;
        submitUptimeVoteMinDurationBlocks =
_settings.submitUptimeVoteMinDurationBlocks;
        signingPolicyThresholdPPM =
_settings.signingPolicyThresholdPPM;
        signingPolicyMinNumberOfVoters =
_settings.signingPolicyMinNumberOfVoters;
    }
```

Then, the expiry threshold is calculated inside the `daemonize` call that attemps to close an expired epoch. This internal call will trigger an underflow when the `rewardExpiryOffsetSeconds` is greater than the current timestamp:

```
    function _closeExpiredRewardEpochs(uint24 _currentRewardEpochId)
internal {
        uint256 expiryThreshold = block.timestamp -
rewardExpiryOffsetSeconds;
        {...}
    }
```

Recommendation

Add a range check in `_updateSettings()` for `rewardExpiryOffsetSeconds`.

Status

Acknowledged.

The Flare Team stated that the values are 7 days on testnets and 90 days on mainnet for the V1 and they will keep the same values for this V2.

© Coinspect 2024

Proof of Concept

The following test shows how setting a high value for the `rewardExpiryOffsetSeconds` triggers an underflow inside `daemonize()` when attempting to close expired epochs.

Output

```
    [FAIL. Reason: Arithmetic over/underflow]
testCoinspectVoterCleanupDaemonizeRevert() (gas: 281257)
```

Test

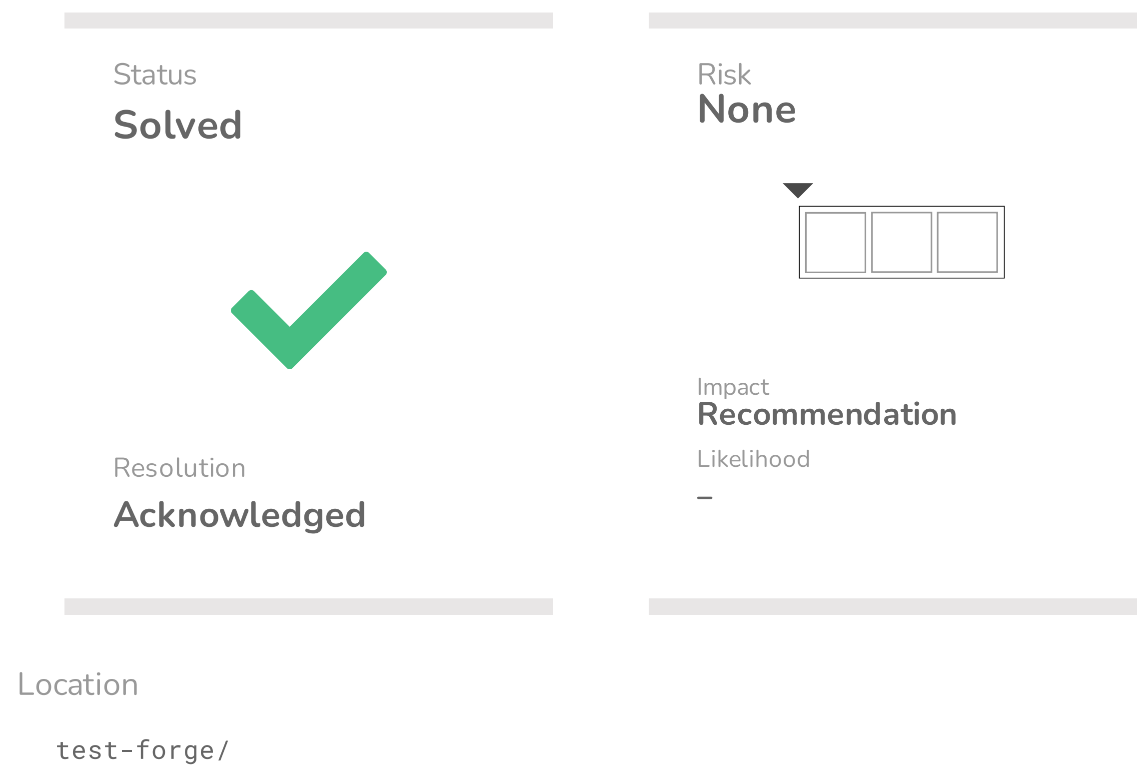
```
    function testCoinspectVoterCleanupDaemonizeRevert() public {
        // Set an overly high value for rewardExpiryOffsetSeconds
        settings = FlareSystemsManager.Settings(1, 2, 3, 4, 5, 6, 7, 8,
9, 10, type(uint32).max);

        vm.prank(governance);
        flareSystemsManager.updateSettings(settings);

        // We care about this value
        assertEq(flareSystemsManager.rewardExpiryOffsetSeconds(),
type(uint32).max);

        vm.mockCall(
            mockRewardManager,
abi.encodeWithSelector(IIRewardManager.closeExpiredRewardEpoch.selector
), abi.encode()
        );

        vm.mockCall(
            mockCleanupBlockNumberManager,

abi.encodeWithSelector(IICleanupBlockNumberManager.setCleanUpBlockNumbe
r.selector),
            abi.encode()
        );

        vm.prank(governance);
        flareSystemsManager.setTriggerExpirationAndCleanup(true);
        assertEq(flareSystemsManager.triggerExpirationAndCleanup(),
true);

        _initializeSigningPolicyAndMoveToNewEpoch(1);
        vm.startPrank(flareDaemon);
        flareSystemsManager.daemonize();

        // start reward epoch 2
        _initializeSigningPolicyAndMoveToNewEpoch(2);
```

© Coinspect 2024

```
        vm.prank(flareDaemon);
        flareSystemsManager.daemonize();

        // start reward epoch 3 and close epoch 1: will underflow
        assertEq(flareSystemsManager.rewardEpochIdToExpireNext(), 1);
        _initializeSigningPolicyAndMoveToNewEpoch(3);
        vm.prank(flareDaemon);
        flareSystemsManager.daemonize();
    }
```

FSC-16

Tests heavily relying on mock calls could fail to reproduce adversarial scenarios



Description

The set of tests made using Foundry heavily rely on mock calls made to other contracts. Although this unit testing approach is not incorrect, it could happen that several adversarial scenarios might not be properly tested.

For some contracts (e.g. the Voter Registry, System Calculator and Entity Manager), it could be relevant to make each unit test linking and deploying all the actual contracts instead of using Foundry `mockCalls`. This approach reduces the chances of encountering issues in production that derive from the interactions between protocol's contracts.

© Coinspect 2024

Recommendation

Include more adversarial tests that do not rely on Foundry's `mockCall`.

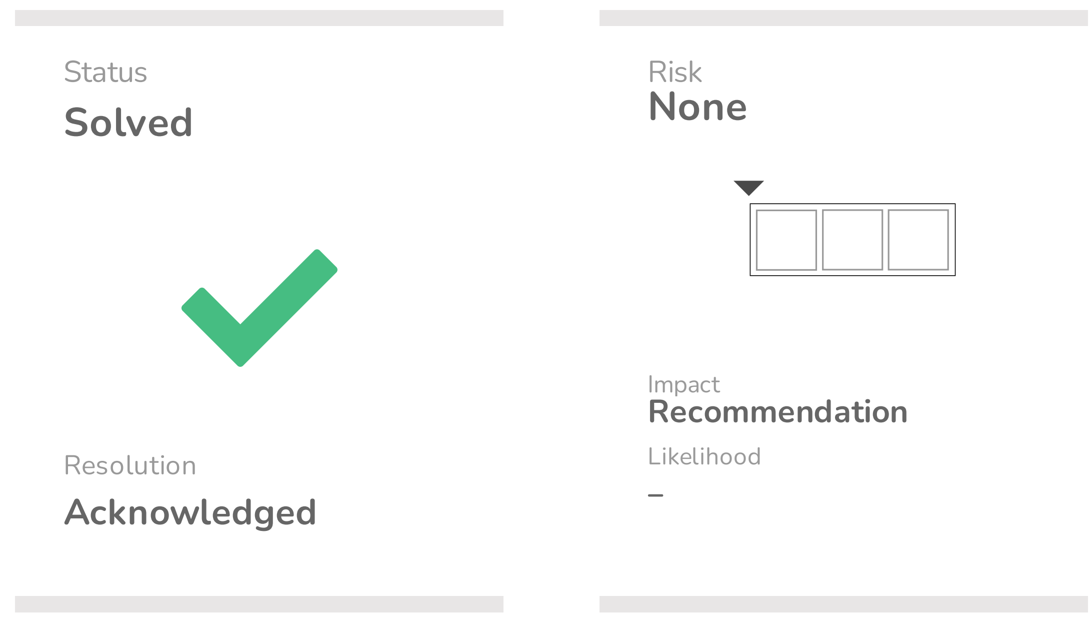
Status

Acknowledged.

The Flare Team stated that they will add more integration between the Foundry Tests.

FSC-17

Broken invariant could disrupt how third parties consume the amount of registered voters



Location

`contracts/protocol/implementation/VoterRegistry.sol`

Description

The functioning of third parties consuming `maxVoters()` could be disrupted as the actual amount of voters could be higher than the reported value.

Full voter registries for a specific epoch, will keep this size across all the epoch and will never decrease only replacing voters with less power. When the governance decreases the `maxVoters`'s value during the registering process and the voter's registry is already full, the following invariant is broken:

```
getNumberOfRegisteredVoters(_rewardEpochId) <= maxVoters
```

```
    function setMaxVoters(uint256 _maxVoters) external onlyGovernance {
        require(_maxVoters <= UINT16_MAX, "_maxVoters too high");
        maxVoters = _maxVoters;
    }
```

This happens because `_registerVoter()` does not remove any voter, replacing those with less voting power with the new ones:

```
        uint256 length = votersAndWeights.voters.length;

        if (length < maxVoters) {
            // we can just add a new one
            votersAndWeights.voters.push(_voter);
            votersAndWeights.weights[_voter] = weight;
        } else {
            // find minimum to kick out (if needed)
```

Recommendation

Document this behavior in the Voter Registry contract. Alternatively, implement a way to remove voters with less voting power to keep the invariant (`amount of registered voters <= maxVoters`).

Status

Acknowledged.

The Flare Team stated this will have no impact since:

- `maxVoters` is only due to change by the governance when registration is not enabled and
- this value will probably only increase

Proof of Concept

The following scenario shows how a `maxVoters` decrease when the voter registry for an epoch is full, breaks the before mentioned invariant. This leaves the system with more voters than the specified `maxVoters`.

Output

```
  Registered voter 0x785E2a241F1e59c0264e0Ae81CaD20F178919efd
  Registered voter 0x9A346D6F41D268704c28c396Aa81b14683223653
  Registered voter 0xC5aFb58ba22614335242A1e57bCcf0bFD97F15eD
  Registered voter 0x826c9ab8aF8944f59e96D7d524a3F76c2AfA900c
  Registered voter 0xECd279Ee12da9D5Ea2E6485963AD39135185F167
  Registered voter 0xe71BDF95c944F4668331625d52fBB9eb101Da602
  Number of registered voters: 6

Voters registered:
  0x785E2a241F1e59c0264e0Ae81CaD20F178919efd
  0x9A346D6F41D268704c28c396Aa81b14683223653
  0xC5aFb58ba22614335242A1e57bCcf0bFD97F15eD
  0x826c9ab8aF8944f59e96D7d524a3F76c2AfA900c
  0xECd279Ee12da9D5Ea2E6485963AD39135185F167
  0xe71BDF95c944F4668331625d52fBB9eb101Da602


  Registered voter 0x90FE592f338ef93bD0150f76A1673eF97a2219c6
  Registered voter 0xe8070EC806CAb48AD3909864D1Ee04c1C366cf7a
  Number of registered voters: 6

Voters registered:
  0x90FE592f338ef93bD0150f76A1673eF97a2219c6
  0xe8070EC806CAb48AD3909864D1Ee04c1C366cf7a
  0xC5aFb58ba22614335242A1e57bCcf0bFD97F15eD
  0x826c9ab8aF8944f59e96D7d524a3F76c2AfA900c
  0xECd279Ee12da9D5Ea2E6485963AD39135185F167
  0xe71BDF95c944F4668331625d52fBB9eb101Da602
```

Test

```
  function testCoinspectRegisterVoters() public {
        // Set Up created first 4 voters, [0,1,2,3]
        _createInitialVotersFromIndex(6, 4); // create 6 more voters
starting from index 4

        IVoterRegistry.Signature memory signature;

        _mockGetCurrentEpochId(0);
        _mockGetVoterAddressesAt();
        _mockGetPublicKeyOfAt();
        _mockGetVoterRegistrationData(10, true);
        _mockVoterWeights();

        // The max voters is 6
        vm.prank(governance);
        voterRegistry.setMaxVoters(6);

        vm.prank(mockFlareSystemsManager);

voterRegistry.setNewSigningPolicyInitializationStartBlockNumber(1);

        // Register the first 6 voters (have less voting power than the
rest)
        for (uint256 i = 0; i < 6; i++) {
            signature = _createSigningPolicyAddressSignature(i, 1);
```

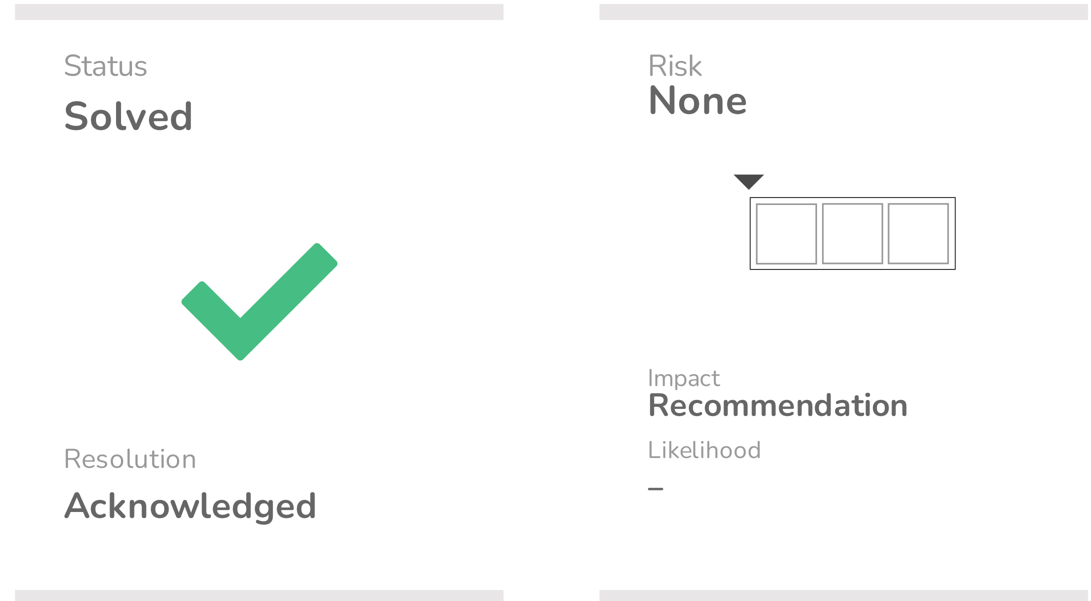
```
            vm.expectEmit();
            emit VoterRegistered(
                initialVoters[i],
                uint24(1),
                initialSigningPolicyAddresses[i],
                initialSubmitAddresses[i],
                initialSubmitSignaturesAddresses[i],
                initialPublicKeyParts1[i],
                initialPublicKeyParts2[i],
                initialVotersWeights[i]
            );
            voterRegistry.registerVoter(initialVoters[i], signature);
            console.log("Registered voter %s", initialVoters[i]);
        }

        uint256 numOfRegisteredVoters =
voterRegistry.getNumberOfRegisteredVoters(1);
        console.log("Number of registered voters: %s",
numOfRegisteredVoters);
        address[] memory _registeredVoters =
voterRegistry.getRegisteredVoters(1);
        console.log("\nVoters registered:");
        for (uint256 i = 0; i < numOfRegisteredVoters; i++) {
            console.log(_registeredVoters[i]);
        }

        console.log("\n");

        // Then maxVoters is reduced to 3
        vm.prank(governance);
        voterRegistry.setMaxVoters(3);

        for (uint256 i = 6; i < 8; i++) {
            signature = _createSigningPolicyAddressSignature(i, 1);
            vm.expectEmit();
            emit VoterRegistered(
                initialVoters[i],
                uint24(1),
                initialSigningPolicyAddresses[i],
                initialSubmitAddresses[i],
                initialSubmitSignaturesAddresses[i],
                initialPublicKeyParts1[i],
                initialPublicKeyParts2[i],
                initialVotersWeights[i]
            );
            voterRegistry.registerVoter(initialVoters[i], signature);
            console.log("Registered voter %s", initialVoters[i]);
        }

        numOfRegisteredVoters =
voterRegistry.getNumberOfRegisteredVoters(1);
        console.log("Number of registered voters: %s",
numOfRegisteredVoters);

        _registeredVoters = voterRegistry.getRegisteredVoters(1);
        console.log("\nVoters registered:");
        for (uint256 i = 0; i < numOfRegisteredVoters; i++) {
            console.log(_registeredVoters[i]);
        }
    }
```

With `_createInitialVotersFromIndex(uint256 _amt, uint256 _fromIndex)`:

```
    function _createInitialVotersFromIndex(uint256 _amt, uint256
_fromIndex) internal {
        uint256 _num = _amt + _fromIndex;
        for (uint256 i = _fromIndex; 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
            // @Coinspect Review modifications
            uint256 initialVoterWeight = 10000 * (i + 1);
            initialVotersWeights.push(initialVoterWeight);

            // 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))));
            }
        }
    }
```

FSC-18

Chilling bypassed by re-delegating voting power



Location

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

Description

Users can change the account they delegate to, effectively bypassing the chilling process for upcoming epochs.

Governance can chill the delegation address of a voter for a specific number of epochs. During this period, any voting power derived from their nodes or delegations is nullified. For example, if a voter is chilled for `2` reward epochs at epoch `3`, they will be chilled until epoch `5`. This means that starting from epoch `6`, they will recover their voting power.

```
function chill(
    bytes20[] calldata _beneficiaryList,
```

```
        uint256 _noOfRewardEpochs
    )
        external onlyGovernance
        returns(
            uint256 _untilRewardEpochId
        )
    {
        uint256 currentRewardEpochId =
flareSystemsManager.getCurrentRewardEpochId();
        _untilRewardEpochId = currentRewardEpochId + _noOfRewardEpochs
+ 1;
        for(uint256 i = 0; i < _beneficiaryList.length; i++) {
            chilledUntilRewardEpochId[_beneficiaryList[i]] =
_untilRewardEpochId;
            emit BeneficiaryChilled(_beneficiaryList[i],
_untilRewardEpochId);
        }
    }
```

When registering a voter, `calculateRegistrationWeight` from the System Calculator is called. This function accounts for the voter's weight, only if they are not chilled for the current reward epoch:

```
        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;
        }
```

A voter can control three accounts: one holding the WNat balance (delegator), another acting as the voter, and a third as the delegation address (delegated). If the delegation address is chilled for a long period, the delegator may re-delegate the WNat balance to a different delegation address registered for another voter, all controlled by the same user, effectively bypassing the chilling penalty.

This issue is considered low-risk since the attacker must accumulate a significant amount of tokens in the delegator account. Moreover, if they manage this attack, there will be another 99 voters in the set that could outweigh any malicious vote performed by this bad actor.

A voter can initially control three accounts. One holding the **WNat** balance (delegator), another one acting as the voter and a last one which is the delegation address (delegated). If that delegation address is chilled for a long time, then the delegator re-delegates the **WNat** balance to a different delegation address

© Coinspect 2024

registered for a different voter (all controlled by the same user), effectively bypassing the chilling penalty.

This issue is considered to have no-risk since the attacker has to accumulate a high amount of tokens in the delegator account. Moreover, if they manage to make this attack, there will be another 99 voters in the set that could outweigh any malicious vote performed by this evil actor.

Additionally, in `calculateRegistrationWeight`, there are no requirements for voters to have nodes registered. This calculation aggregates the voting power derived from registered nodes and **WNat** delegations, allowing voters to receive all their weight from **WNat** delegations without having any node registered.

Recommendation

Document that the chilling process should not be fully trusted in cases where one voter has enough voting power without relying on third-party delegations.

Status

Acknowledged.

Although this issue is considered feasible, the Flare Team stated that there are no plans to replace the WNat contract. Also, Flare considers that since re-delegation usually takes time, this issue could only affect voting power from a chilled data provider for a limited duration.

Proof of Concept

The scenario below shows an attacker who controls an account with WNat. From that account, they delegate to the delegation address of a voter (also under their control). When that voter's delegation address is chilled, for the next epochs they revoke the WNat delegation and re-delegate to a new Sybil delegation address they control. As a result, they manage to bypass being chilled by creating a new pair of `(SybilVoter, SybilDelegation)`.

Output

```
  Delegating to Voter's delegation address
0x8CCaea58b4062d79EDa8E9E14C3D46C7655E8482...
  Chilling Voter 0x4DAfB91f6682136032C004768837e60Bc099E52C...
  Calculating registration weight of Voter:
0x4DAfB91f6682136032C004768837e60Bc099E52C ...
  Registration weight for voter = 0

Re-delegating to SybilVoter's delegation address
0x9D8511BA640540f7706c6Cd58138C3aFbFFA8dC0...
  Calculating registration weight of SybilVoter:
0x1606a175DD34f6ee99424f8e220f22144D70A112 ...
  Registration weight for SybilVoter = 5372
  Error: The voting power is non zero
  Error: a == b not satisfied [uint]
        Left: 5372
       Right: 0
```

Test

```
        function testCoinspectBypassChillStatus() public {
        // The attacker controls an account that makes and revokes WNat
delegations
        // Then, creates multiple pairs of (Voter, DelegationAddress)
that rotates when
        // their voter is chilled. This attack does not require any
active node to work.
        //
        // At this point the attacker controls 1 + N * (2) accounts
where:
        // One account is the owner of WNat, used to delegate
        // N: The amount of sets of (voter, delegationAddress)

        bytes20[] memory nodeIds = new bytes20[](0);
        uint256[] memory nodeWeights = new uint256[](0);

        // Voter: address that is part of a signing policy
        // Delegation: address that receives the delegated voting
weight, assigned to the Voter
        address voter = makeAddr("voter");
        address delegationAddress = makeAddr("delegation");

        address sybilVoter = makeAddr("sybilVoter");
        address sybilDelegationAddress =
makeAddr("sybilDelegationAddress");

        uint24 rewardEpochId = 12345;
        uint256 votePowerBlockNumber = 1234567;

        // Simulate that the delegationAddress is chilled forever
        console.log("Delegating to Voter's delegation address %s...",
delegationAddress);
        vm.mockCall(
            address(calculator.voterRegistry()),

abi.encodeWithSelector(IVoterRegistry.chilledUntilRewardEpochId.selecto
r, bytes20(delegationAddress)),
```

© Coinspect 2024

```
            abi.encode(type(uint256).max - rewardEpochId)
        );

        // MockCalls to Entity Manager
        // The votePowerBlock for rewardEpochId = 12345
        _mockCallsToPrepareWeightRegistrationCalculation(
            voter, delegationAddress, rewardEpochId,
votePowerBlockNumber, nodeIds, nodeWeights
        );

        // Check the voter's weight
        console.log("Chilling Voter %s...", voter);

        console.log("Calculating registration weight of Voter: %s ...",
voter);

        vm.prank(address(calculatorNoMirroring.voterRegistry()));
        uint256 registrationWeight =
calculator.calculateRegistrationWeight(voter, rewardEpochId,
votePowerBlockNumber);

        assertEq(registrationWeight, 0); // the voter was chilled
        console.log("Registration weight for voter = %s",
registrationWeight);

        // Then, for the next epochs the attacker activates a new set
of (SybilVoter, SybilDelegation)
        // and revokes the delegations from made to the
DelegationAddress from the previous epochs.
        rewardEpochId = rewardEpochId + 2;
        votePowerBlockNumber = votePowerBlockNumber + 100;

        // Simulate that the SybilDelegation not chilled
        console.log("\nRe-delegating to SybilVoter's delegation address
%s...", sybilDelegationAddress);

        vm.mockCall(
            address(calculator.voterRegistry()),

abi.encodeWithSelector(IVoterRegistry.chilledUntilRewardEpochId.selecto
r, bytes20(sybilDelegationAddress)),
            abi.encode(0)
        );

        // The votePowerBlock + 100 for rewardEpochId = 12345 + 2
        _mockCallsToPrepareWeightRegistrationCalculation(
            sybilVoter, sybilDelegationAddress, rewardEpochId,
votePowerBlockNumber, nodeIds, nodeWeights
        );

        // Check the new voter's weight
        console.log("Calculating registration weight of SybilVoter: %s
...", sybilVoter);

        vm.prank(address(calculatorNoMirroring.voterRegistry()));
        registrationWeight =
calculator.calculateRegistrationWeight(sybilVoter, rewardEpochId,
votePowerBlockNumber);
        console.log("Registration weight for SybilVoter = %s",
registrationWeight);
```

```
        assertEq(registrationWeight, 0, "The voting power is non
zero");
    }
```

With _mockCallsToPrepareWeightRegistrationCalculation():

```
    function _mockCallsToPrepareWeightRegistrationCalculation(
        address _voter,
        address _delegation,
        uint256 _rwEpochId,
        uint256 _votePowerBlockNumber,
        bytes20[] memory nodeIds,
        uint256[] memory nodeWeights
    ) internal {
        vm.mockCall(
            address(calculator.entityManager()),

abi.encodeWithSelector(IEntityManager.getNodeIdsOfAt.selector, _voter,
_votePowerBlockNumber),
            abi.encode(nodeIds)
        );

        vm.mockCall(
            address(calculator.pChainStakeMirror()),
            abi.encodeWithSelector(

bytes4(keccak256("batchVotePowerOfAt(bytes20[],uint256)")), nodeIds,
_votePowerBlockNumber
            ),
            abi.encode(nodeWeights)
        );

        vm.mockCall(
            address(calculator.entityManager()),

abi.encodeWithSelector(IEntityManager.getDelegationAddressOfAt.selector
, _voter, _votePowerBlockNumber),
            abi.encode(_delegation)
        );

        vm.mockCall(
            address(calculator.wNat()),

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

        vm.mockCall(
            address(calculator.wNat()),
            abi.encodeWithSelector(
                bytes4(keccak256("votePowerOfAt(address,uint256)")),
_delegation, _votePowerBlockNumber
            ),
            abi.encode(WNAT_WEIGHT)
        );
```

```
        vm.mockCall(
            address(calculator.wNatDelegationFee()),

abi.encodeWithSelector(IWNatDelegationFee.getVoterFeePercentage.selecto
r, _voter, _rwEpochId),
            abi.encode(DELEGATION_FEE_BIPS)
        );
    }
```

6. Disclaimer

The information presented in this document is provided "as is" and without warranty. The present security audit does not cover any on-chain systems or frontends that communicate with the network, nor the general operational security of the organization that developed the code.

© Coinspect 2024