Multi-Gov: Cross Chain Governance Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

0kage

Hans

Assisting Auditors

October 9, 2024

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

MultiGov is a cross-chain governance system designed to enable decentralized decision-making across multiple blockchain networks. The protocol allows token holders to participate in governance activities, such as creating proposals and voting, regardless of the specific chain on which their tokens reside.

Key components of the MultiGov system include:

Hub chain: Serves as the central coordination point for cross-chain governance activities. It hosts the main governance contract (HubGovernor) and manages the aggregation of votes from different chains.

Spoke chains: Represent the various blockchain networks integrated into the MultiGov system. Each spoke chain contains contracts that facilitate local voting and communication with the hub.

Cross-Chain communication: Utilizes the Wormhole protocol to enable secure message passing between the hub and spoke chains.

Fractional Voting: Supports a sophisticated voting mechanism where users can split their voting power across different options (For, Against, Abstain).

Vote aggregation: Implements a system to collect and combine votes from multiple chains, ensuring fair representation of all token holders in the governance process.

Extensibility: Includes features for extending proposal deadlines to account for disruption in cross-chain messaging.

The protocol aims to solve the challenge of fragmented governance in multi-chain ecosystems, providing a unified decision-making platform for decentralized communities spread across different blockchain networks.

5 Audit Scope

Cyfrin conducted a security audit of the MultiGov - Crosschain Governance system, focusing specifically on the EVM-compatible codebase. The audit was performed on the codebase at commit hash 7513aa5.

The audit focused on identifying potential security vulnerabilities, logical errors, and adherence to best practices within these smart contracts. Special attention was given to the cross-chain communication mechanisms, vote counting logic, and governance processes implemented in the system.

This audit was limited only to the Solidity smart contracts and did not include any non-EVM contracts, off-chain components or front-end applications.

Following files in the `evm` folder were included in the scope of the audit:

- src/HubEvmSpokeAggregateProposer.sol
- src/HubEvmSpokeVoteDecoder.sol
- src/HubGovernor.sol
- src/HubMessageDispatcher.sol
- src/HubProposalExtender.sol
- src/HubProposalMetadata.sol
- src/HubVotePool.sol
- src/SpokeAirlock.sol
- src/SpokeMessageExecutor.sol
- src/SpokeMetadataCollector.sol
- src/SpokeVoteAggregator.sol
- src/WormholeDispatcher.sol
- src/extensions/GovernorMinimumWeightedVoteWindow.sol
- src/extensions/GovernorSettableFixedQuorum.sol
- src/lib/Checkpoints.sol
- src/lib/GovernorCountingFractional.sol
- src/lib/SpokeCountingFractional.sol
- src/interfaces/IHubVoteExtender.sol
- src/interfaces/ISpokeVoteDecoder.sol
- src/interfaces/IVoteExtender.sol
- script/DeployHubContractsBaseImpl.s.sol
- script/DeployHubContractsSepolia.s.sol
- script/DeploySpokeContractsBaseImpl.sol
- script/DeploySpokeContractsOptimismSepolia.sol

6 Executive Summary

Over the course of 11 business days, the Cyfrin team conducted an audit on the Multi-Gov: Cross Chain Governance smart contracts provided by Wormhole Foundation. In this period, a total of 17 issues were found.

MultiGov is a cross-chain governance system designed to enable decentralized decision-making across multiple blockchain networks. It allows token holders to participate in governance activities, such as creating proposals and voting, regardless of the specific chain on which their tokens reside. The system comprises a hub chain that serves as the central coordination point and multiple spoke chains that facilitate local voting and communication with the hub.

Audit revealed several low-risk and informational issues, including minor code quality improvements, fixes to deployment scripts, documentation enhancements, and potential gas optimization opportunities.

The testing suite and fuzz test setup for the MultiGov system were of exceptionally high quality, demonstrating a strong commitment to code reliability and security. To further enhance the testing process, we recommend implementing end-to-end test setups or fork tests with deployed contracts on testnets to observe the entire voting lifecycle across multiple chains.

Furthermore, the use of standardized OpenZeppelin libraries for voting checkpoints, voting power calculation, and governance processes contributed to a tight codebase with a reduced attack surface. This approach to leveraging well-audited, community-standard implementations is commendable and significantly enhances the overall security posture of the system.

Summary

Project Name	Multi-Gov: Cross Chain Governance
Repository	example-multigov
Commit	7513aa590008...
Audit Timeline	Aug 26th - Sep 9th
Methods	Manual Review, Stateful Fuzzing

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	0
Low Risk	9
Informational	6
Gas Optimizations	2
Total Issues	17

Summary of Findings

[L-1] Lack of access control in `HubProposalExtender::initialize`	Resolved
[L-2] Lack of zero address checks at multiple places	Acknowledged
[L-3] Lack of limits of vote weight window can lead to unfair voting power estimation	Acknowledged
[L-4] `HubGovernor` lacks validation on `HubProposalExtender` ownership	Resolved
[L-5] Contract accounts and account abstraction wallets face limitations in Cross-Chain proposal creation and voting power utilization	Resolved
[L-6] Cross-Chain votes can be invalidated in some scenarios due to Hub-Spoke timestamp mismatch	Acknowledged

[L-7] Potential de-synchronization of spoke registries between `HubVotePool` and `HubEvmSpokeAggregateProposer`	Acknowledged
[L-8] Lack of minimum threshold for `maxQueryTimestampOffset` in `HubEvmSpokeAggregateProposer`	Acknowledged
[L-9] Inconsistent vote weight windows enable potential voting power manipulation	Resolved
[I-1] Missing event emissions when changing critical parameters	Resolved
[I-2] Potential long-term failure in `HubGovernor::getVotes` due to uint32 Casting of Timestamps	Acknowledged
[I-3] Proposal deadline extension is allowed even in `Pending` state	Resolved
[I-4] Missing deployment script for `HubEvmSpokeAggregateProposer`	Acknowledged
[I-5] Missing implementation to cast fractional vote with signature	Acknowledged
[I-6] Inconsistent documentation related to the deployment of `SpokeMessageExecutor`	Resolved
[G-1] Hash comparison can be optimized	Resolved
[G-2] `_checkProposalEligibility` gas optimization	Acknowledged

7 Findings

7.1 Low Risk

7.1.1 Lack of access control in `HubProposalExtender::initialize`

Description: `HubProposalExtender::initialize` can only be called once. Lack of access control in this function would mean that an attacker can front-run this function to set a a malicious Governor. This can prevent the extender admin from extending duration for any proposal

Impact: Front-running initialize can potentially prevent extension of proposals

Proof of Concept: Run the following fuzz test:

```
  function testFuzz_FrontRun_HubProposalExtender_Initialize(address attacker, address
  ↪  maliciousGovernor) public {
    vm.assume(attacker != address(0) && attacker != address(this) && attacker != address(timelock));
    vm.assume(maliciousGovernor != address(0) && maliciousGovernor != address(governor));

    // Create a new HubProposalExtender
    HubProposalExtenderHarness newExtender = new HubProposalExtenderHarness(
        whitelistedExtender,
        extensionDuration,
        address(timelock),
        minimumTime,
        voteWeightWindow,
        minimumTime
    );

    // Attacker front-runs the initialization
    vm.prank(attacker);
    newExtender.initialize(payable(maliciousGovernor));

    // Assert that the malicious governor was set
    assertEq(address(newExtender.governor()), maliciousGovernor);

    // Attempt to initialize again with the intended governor (this should fail)
    vm.prank(address(timelock));
    vm.expectRevert(HubProposalExtender.AlreadyInitialized.selector);
    newExtender.initialize(payable(address(governor)));

    // Assert that the governor is still the malicious one
    assertEq(address(newExtender.governor()), maliciousGovernor);
}
```

Recommended Mitigation: Consider restricting access to `initialize` function only to the contract owner.

Wormhole Foundation Fixed in PR 177.

Cyfrin: Verified.

7.1.2 Lack of zero address checks at multiple places

Description: In general, codebase lacks input validations such as zero address checks.

`HubVotePool::constructor`

```
  constructor(address _core, address _hubGovernor, address _owner) QueryResponse(_core) Ownable(_owner)
  ↪  { //@note initialized query response with womrhole core
    hubGovernor = IGovernor(_hubGovernor);
    HubEvmSpokeVoteDecoder evmDecoder = new HubEvmSpokeVoteDecoder(_core, address(this));
    _registerQueryType(address(evmDecoder), QueryResponse.QT_ETH_CALL_WITH_FINALITY);
```

```
  }
```

HubVotePool::registerSpoke

```
  function registerSpoke(uint16 _targetChain, bytes32 _spokeVoteAddress) external {
    _checkOwner();
    _registerSpoke(_targetChain, _spokeVoteAddress);
  }
```

HubPool::setGovernor

```
 function setGovernor(address _newGovernor) external {
    _checkOwner();
    hubGovernor = IGovernor(_newGovernor);
}
```

HubProposalExtender::setVoteExtenderAdmin

```
 function setVoteExtenderAdmin(address _voteExtenderAdmin) external {
    _checkOwner();
    _setVoteExtenderAdmin(_voteExtenderAdmin); //@audit @info missing zero address check
  }
```

Recommended Mitigation: Consider adding zero address checks where applicable.

Wormhole Foundation Acknowledged.

Cyfrin: Acknowledged.

7.1.3 Lack of limits of vote weight window can lead to unfair voting power estimation

Description: There are no checks on the vote weight window duration in the `HubGovernor::setVoteWeightWindow`. Although it is a governance only function, setting very small or extremely large vote weight duration could lead to unfair voting power estimation.

```
  function setVoteWeightWindow(uint48 _weightWindow) external {
    _checkGovernance();
    _setVoteWeightWindow(_weightWindow);
  }
```

```
  function _setVoteWeightWindow(uint48 _windowLength) internal { //@audit no check on windowLength
    emit VoteWeightWindowUpdated(
      SafeCast.toUint48(voteWeightWindowLengths.upperLookup(SafeCast.toUint48(block.timestamp))),
      ↪  _windowLength
    );
    voteWeightWindowLengths.push(SafeCast.toUint96(block.timestamp), uint160(_windowLength));
  }
```

Impact: Extremely small vote weight period can attract token manipulation from users and extremely large voting period will penalize recent holders.

Recommended Mitigation: Consider adding reasonable MINIMUM and MAXIMUM limits on the vote weight period instead of completely leaving it to governance.

Wormhole Foundation Acknowledged.

Cyfrin: Acknowledged.

7.1.4 `HubGovernor` lacks validation on `HubProposalExtender` ownership

Description: The `HubGovernor` contract accepts a `HubProposalExtender` address as an immutable parameter in its constructor. However, it does not verify that this extender is owned by the `timelock` contract.

This oversight allows for the deployment of a HubGovernor instance with a HubProposalExtender that is not controlled by the governance system, potentially bypassing critical governance controls.

The deployment script in `DeployHubContractsBaseImpl.s.sol` deploy the extender with the owner as `config.voteExtenderAdmin` address and not the timelock address.

`DeployHubContractsBaseImpl.s.sol`

```
 function run()
    public
    returns (
      TimelockController,
      HubVotePool,
      HubGovernor,
      HubProposalMetadata,
      HubMessageDispatcher,
      HubProposalExtender
    )
  {
    DeploymentConfiguration memory config = _getDeploymentConfiguration();
    Vm.Wallet memory wallet = _deploymentWallet();
    vm.startBroadcast(wallet.privateKey);
    TimelockController timelock =
      new TimelockController(config.minDelay, new address[](0), new address[](0), wallet.addr);

    HubProposalExtender extender = new HubProposalExtender(
      config.voteExtenderAdmin, config.voteTimeExtension, config.voteExtenderAdmin,
      ↪  config.minimumExtensionTime
    );

   // other code
}
```

However, all tests written in `HubProposalExtender.t.sol` use the `timelock` address as the contract owner.

Impact: Changes to critical parameters such as extension duration or the vote extender admin could be made without going through the proper governance process.

Proof of Concept: Add the following to `HubGovernor.t.sol`

```
contract ExtenderOwnerNotTimelockTest is HubGovernorTest {
    address attacker = address(200);
    HubProposalExtender maliciousExtender;

    function testExtenderOwnerNotTimelock() public {
        // Attacker deploys their own HubProposalExtender
        vm.prank(attacker);
        maliciousExtender = new HubProposalExtender(
            attacker, // voteExtenderAdmin
            1 hours,  // extensionDuration
            attacker, // owner
            30 minutes // MINIMUM_EXTENSION_DURATION
        );

        // Deploy HubGovernor with the malicious extender
```

```
        HubGovernor.ConstructorParams memory params = HubGovernor.ConstructorParams({
            name: "Vulnerable Gov",
            token: token,
            timelock: timelock,
            initialVotingDelay: 1 days,
            initialVotingPeriod: 3 days,
            initialProposalThreshold: 500_000e18,
            initialQuorum: 100e18,
            hubVotePoolOwner: address(timelock),
            wormholeCore: address(wormhole),
            governorProposalExtender: address(maliciousExtender),
            initialVoteWeightWindow: VOTE_WEIGHT_WINDOW
        });

        HubGovernor vulnerableGovernor = new HubGovernor(params);

        // Verify that the deployment succeeded and the malicious extender is set
        assertEq(address(vulnerableGovernor.HUB_PROPOSAL_EXTENDER()), address(maliciousExtender));

        // Verify that the attacker owns the extender, not the timelock
        assertEq(Ownable(address(maliciousExtender)).owner(), attacker);
        assertTrue(Ownable(address(maliciousExtender)).owner() != address(timelock));

       // initialize
       maliciousExtender.initialize(payable(vulnerableGovernor));

        // Attacker can change extension duration without governance
        vm.prank(attacker);
        maliciousExtender.setExtensionDuration(2 days);

        // The change is reflected in the extender used by the governor
        assertEq(HubProposalExtender(address(vulnerableGovernor.HUB_PROPOSAL_EXTENDER())).extensionDura
        ↪  tion(), 2
        ↪  days);
    }
}
```

Recommended Mitigation: Consider adding a check in the `HubGovernor` constructor to ensure the `HubProposalExtender` is owned by the `timelock`. Alternately, consider transferring ownership of the `HubProposalExtender` to the `timelock` contract in the deployment scripts.

`HubGovernor.sol`

```
constructor(ConstructorParams memory _params) {
    // ... existing constructor logic ...

    if (Ownable(_params.governorProposalExtender).owner() != address(_params.timelock)) {
        revert ProposalExtenderNotOwnedByTimelock();
    } //@audit add this validation

    HUB_PROPOSAL_EXTENDER = IVoteExtender(_params.governorProposalExtender);

    // ... rest of the constructor ...
}
```

Wormhole Foundation Fixed in PR 179.

Cyfrin: Verified.

7.1.5 Contract accounts and account abstraction wallets face limitations in Cross-Chain proposal creation and voting power utilization

Description: Multichain governance system allows users to aggregate all their voting power that is distributed across all chains connected to the system (including the hub chain) to gather enough voting power to create a new proposal.

However, contract accounts & AA wallets holding tokens on spoke chains cannot use their voting power to create a proposal on the hub chain. Such accounts may not control the same address across all the chains where they have tokens, which means that they would not be able to use their voting power on the HubChain, unless they either transfer the tokens or delegate the votes to a trusted Externally Operated Account (EOA).

`HubEvmSpokeAggregateProposer::checkAndProposeIfEligible` checks that the caller (msg.sender) is same as the queriedAccount and reverts otherwise.

```
function _checkProposalEligibility(bytes memory _queryResponseRaw, IWormhole.Signature[] memory
↪  _signatures)
   internal
   view
   returns (bool)
{
     // code

      for (uint256 i = 0; i < _queryResponse.responses.length; i++) {
          // code...

         // Check that the address being queried is the caller
         if (_queriedAccount != msg.sender) revert InvalidCaller(msg.sender, _queriedAccount);

        // code...
     }

}
```

These accounts need to delegate their votes much before the proposal creation date for the vote weight to be considered - transferring just prior to proposal creation would lead to a temporary loss of voting power. This is because voting power is calculated as the minimum token holding over the vote weight window.

This constraint of delegating voting power long before a proposal is created introduces an associated risk, ie. delegated voting power of contracts can be misused for voting on other unrelated proposals. It increases overall trust assumptions in a decentralized voting system.

Impact: There are two risks:

1. Recently delegated voting power will be temporarily lost because of voting window constraints
2. Voting power delegated well before proposal creation can be misused to cast vote on other active proposals.

Recommended Mitigation: Consider implementing a mechanism that allows contract accounts to prove ownership across chains without requiring the same address.

Alternatively, consider documenting the constraints and risks of delegation for contract accounts and account abstraction wallets in the existing design. This would help the relevant parties plan in advance if, how & when to delegate their voting power to an EOA in the event of proposal creation.

Wormhole Foundation Fixed in PR 178

Cyfrin: Verified.

7.1.6 Cross-Chain votes can be invalidated in some scenarios due to Hub-Spoke timestamp mismatch

Description: Current implementation of the cross-chain voting mechanism in the `HubVotePool` contract does not account for the timestamp of votes cast on spoke chains. When the `HubVotePool::crossChainVote` function is

called, it uses the current timestamp to determine vote validity, rather than the timestamp when the votes were actually cast on the spoke chains.

This leads to a situation where valid votes from spoke chains can be rejected if they are processed by the hub after the voting period has ended, even if they were cast within the valid voting timeframe.

The issue stems from the use of OpenZeppelin's Governor contract, which checks the proposal's state based on the current block timestamp when castVote is called. In a cross-chain context, this doesn't account for the latency between vote casting on spoke chains and vote processing on the hub chain.

```
// In HubVotePool's crossChainVote function (simplified)
function crossChainVote(bytes memory _queryResponseRaw, IWormhole.Signature[] memory _signatures)
↪   external {
    // ... (parsing and verification)

    // This call uses the current timestamp, not the spoke chain's timestamp
    hubGovernor.castVoteWithReasonAndParams(
        _proposalId,
        UNUSED_SUPPORT_PARAM,
        "rolled-up vote from governance spoke token holders",
        _votes
    );
}

// In OpenZeppelin's Governor contract
function _castVote(
    uint256 proposalId,
    address account,
    uint8 support,
    string memory reason,
    bytes memory params
) internal virtual returns (uint256) {
    // This check uses the current timestamp
    _validateStateBitmap(proposalId, _encodeStateBitmap(ProposalState.Active));
    // ...
}
```

Impact: Valid votes from spoke chains may be discarded if processed after the voting period. The final vote tally may not accurately represent all valid votes cast across the network.

It is worthwhile to note that this risk exists only for spoke chains. While voters on the hub chain can vote upto the last minute with a guarantee that their votes will be valid, such guarantee does not exist for the spoke chain voters. Also, time to finality for most L2's is ~15 minutes and hence this edge-case will exist in the current design irrespective of the crank-turner's efficiency.

We also note that the `HubProposalExtender` has the ability to do a one-time extension of voting period to account for edge-cases related to chain-outages and scenarios where votes could not be propagated back onto the hub chain. While this definitely mitigates the risk significantly, it does not completely remove the risk of vote loss.

Proof of Concept: Add the following to `HubGovernor.t.sol`

```
contract CountHubPoolVote is HubGovernorTest {
  function test_Revert_HubPoolValidVotesRejected() external {
    uint256 proposalId = 0;
    uint16 queryChainId = 2;
    address _spokeContract= address(999);

   // set current hub vote pool
   governor.exposed_setHubVotePool(address(hubVotePool));

   // and register spoke contract
    {
```

```
    vm.startPrank(address(timelock));
      hubVotePool.registerSpoke(queryChainId, addressToBytes32(_spokeContract));
    vm.stopPrank();
  }

  {
    (, delegates) = _setGovernorAndDelegates();
    (ProposalBuilder builder) = _createArbitraryProposal();

    vm.startPrank(delegates[0]);
    proposalId =
      governor.propose(builder.targets(), builder.values(), builder.calldatas(), "hi");
    vm.stopPrank();

    _jumpToActiveProposal(proposalId);
  }


  bytes memory ethCall = QueryTest.buildEthCallWithFinalityRequestBytes(
    bytes("0x1296c33"), // random blockId: a hash of the block number
    "finalized", // finality
    1, // numCallData
    QueryTest.buildEthCallDataBytes(
      _spokeContract, abi.encodeWithSignature("proposalVotes(uint256)", proposalId)
    )
  );

  console2.log("block number %i", block.number);
  console2.log("block timestamp %i", block.timestamp);
  console2.log("voting period %i", governor.votingPeriod());
  console2.log("proposal vote start %i", governor.proposalSnapshot(proposalId));

  bytes memory ethCallResp = QueryTest.buildEthCallWithFinalityResponseBytes(
    uint64(block.number), // block number
    blockhash(block.number), // block hash
    uint64(block.timestamp), // block time US //@note send the timestamp before vote start
    1, // numResults
    QueryTest.buildEthCallResultBytes(
      abi.encode(
        proposalId,
        SpokeCountingFractional.ProposalVote({
          againstVotes: uint128(200),
          forVotes: uint128(10000),
          abstainVotes: uint128(800)
        })
      )
    ) // results
  );


  bytes memory _queryRequestBytes = QueryTest.buildOffChainQueryRequestBytes(
    VERSION, // version
    0, // nonce
    1, // num per chain requests
    abi.encodePacked(
      QueryTest.buildPerChainRequestBytes(
        queryChainId, // chainId
        hubVotePool.QT_ETH_CALL_WITH_FINALITY(),
        ethCall
      )
    )
  );
```

```
  bytes memory _resp = QueryTest.buildQueryResponseBytes(
      VERSION, // version
      OFF_CHAIN_SENDER, // sender chain id
      OFF_CHAIN_SIGNATURE, // signature
      _queryRequestBytes, // query request
      1, // num per chain responses
      abi.encodePacked(
        QueryTest.buildPerChainResponseBytes(queryChainId , hubVotePool.QT_ETH_CALL_WITH_FINALITY(),
        ↪  ethCallResp)
      )
    );

    vm.warp(governor.proposalDeadline(proposalId) + 1); // move beyond proposal deadline - no longer
    ↪  active
    IWormhole.Signature[] memory signatures = _getSignatures(_resp);

    vm.expectRevert(abi.encodeWithSelector(IGovernor.GovernorUnexpectedProposalState.selector,
    ↪  proposalId, IGovernor.ProposalState.Defeated, bytes32(1 <<
    ↪  uint8(IGovernor.ProposalState.Active))));
    hubVotePool.crossChainVote(_resp, signatures);
  }
}
```

Recommended Mitigation: Instead of immediately calling `castVote` on Governor, consider implementing a custom vote aggregation in `HubVotePool` contract. After the voting period ends, a separate function in `HubVotePool` can submit the aggregated cross-chain votes to `Governor`. This voting after voting period ends can only be allowed for `HubVotePool` within a specific time period after voting ends. Main idea from a security standpoint is to avoid a scenario where legitimate votes cast on a spoke chain are not considered in the final tally on hub chain.

Wormhole Foundation Acknowledged.

Cyfrin: Acknowledged.

7.1.7 Potential de-synchronization of spoke registries between `HubVotePool` and `HubEvmSpokeAggregateProposer`

Description: The system maintains two separate registries for spoke chain addresses: one in `HubVotePool` and another in `HubEvmSpokeAggregateProposer`.

These registries use different mechanisms and can be updated independently:

`HubVotePool` uses a checkpoint-based system:

```
mapping(uint16 emitterChain => Checkpoints.Trace256 emitterAddress) internal emitterRegistry;
```

`HubEvmSpokeAggregateProposer` uses a simple mapping:

```
mapping(uint16 wormholeChainId => address spokeVoteAggregator) public registeredSpokes;
```

The update mechanisms also are different:

- `HubVotePool` registry can be updated directly by its owner.
- `HubEvmSpokeAggregateProposer` registry is updated through governance proposals executed by HubGovernor.

This design creates a risk of the two registries becoming out of sync, potentially leading to inconsistent behavior in the cross-chain voting system.

Impact: Out-of-sync spoke registries may consider votes valid for one operation (eg. vote aggregation for proposers) but invalid for other operation (eg. hub pool voting). As a result, users might create proposals using votes that are no longer valid for the HubPool.

Recommended Mitigation: Consider having a single source of truth for both the `HubVotePool` and `HubEvmSpokeAggregateProposer` contracts. In general, a checkpoint based registry is more robust than a simple mapping used in `HubEvmSpokeAggregateProposer`, which does not track historical changes.

Wormhole Foundation Acknowledged.

Cyfrin: Acknowledged.

7.1.8 Lack of minimum threshold for `maxQueryTimestampOffset` in `HubEvmSpokeAggregateProposer`

Description: In the `HubEvmSpokeAggregateProposer` contract, the `setMaxQueryTimestampOffset` function allows the owner to set a new value for `maxQueryTimestampOffset` without enforcing a minimum threshold. The current implementation only checks that the new value is not zero:

```
function _setMaxQueryTimestampOffset(uint48 _newMaxQueryTimestampOffset) internal {
    if (_newMaxQueryTimestampOffset == 0) revert InvalidOffset();
    emit MaxQueryTimestampOffsetUpdated(maxQueryTimestampOffset, _newMaxQueryTimestampOffset);
    maxQueryTimestampOffset = _newMaxQueryTimestampOffset;
}
```

Impact: This lack of a lower bound could potentially allow the offset to be set to an impractically low value, which could disrupt the cross-chain vote aggregation for creating proposals.

Recommended Mitigation: Consider implementing a minimum threshold for the `maxQueryTimestampOffset`.

Wormhole Foundation Acknowledged.

Cyfrin: Acknowledged.

7.1.9 Inconsistent vote weight windows enable potential voting power manipulation

Description: Current implementation allows for independent setting of vote weight windows on the hub chain (via `HubGovernor`) and spoke chains (via SpokeVoteAggregator). This can lead to inconsistencies in how voting power is calculated across different chains for the same governance proposals.

In `HubGovernor.sol`, it is set via governance:

```
function setVoteWeightWindow(uint48 _weightWindow) external {
    _checkGovernance();
    _setVoteWeightWindow(_weightWindow);
}
```

In `SpokeVoteAggregator.sol`, it is set by the owner of contract:

```
function setVoteWeightWindow(uint48 _voteWeightWindow) public {
    _checkOwner();
    _setVoteWeightWindow(_voteWeightWindow);
}
```

There is no mechanism to ensure these settings remain synchronized across chains.

Impact: The inconsistency in vote weight windows can lead to voting power discrepancies. Since voting power is determined as the minimum token holding over vote weight window, having different windows could result in unfair voting representation across different spoke chains.

Recommended Mitigation: Consider making `SpokeAirlock` contract the owner of the `SpokeVoteAggregator`. All key parameter changes would then be subject to Hub Governance approval. In general, consider avoiding multiple admins for each spoke chain and delegate all authority to the Hub Governance.

Although README suggests that `SpokeAirlock` is indeed the owner of `SpokeVoteAggregator`, implementation for the same is missing in the `DeploySpokeContractsBaseImpl.sol` deployment script.

Wormhole Foundation Fixed in PR 184

Cyfrin: Verified.

7.2 Informational

7.2.1 Missing event emissions when changing critical parameters

Description: Critical parameters are changed without corresponding event emissions. This leads to a lack of overall transparency.

`HubVotePool::setGovernor` does not emit an event when governor is changed.

```
  function setGovernor(address _newGovernor) external {
    _checkOwner();
    hubGovernor = IGovernor(_newGovernor);
    //@audit @info missing event check for this critical change
  }
```

`HubProposalExtender::extendProposal` does not emit an event when a proposal is extended.

```
function extendProposal(uint256 _proposalId) external {
    uint256 exists = governor.proposalSnapshot(_proposalId);
    if (msg.sender != voteExtenderAdmin) revert AddressCannotExtendProposal();
    if (exists == 0) revert ProposalDoesNotExist();
    if (extendedDeadlines[_proposalId] != 0) revert ProposalAlreadyExtended();

    IGovernor.ProposalState state = governor.state(_proposalId);
    if (state != IGovernor.ProposalState.Active && state != IGovernor.ProposalState.Pending) {
      revert ProposalCannotBeExtended();
    }

    extendedDeadlines[_proposalId] = uint48(governor.proposalDeadline(_proposalId)) + extensionDuration;
   //@audit @info missing event emission
  }
```

Recommended Mitigation: Consider adding events when critical contract parameters are changed.

Wormhole Foundation Fixed in PR 173.

Cyfrin: Verified.

7.2.2 Potential long-term failure in `HubGovernor::getVotes` due to uint32 Casting of Timestamps

Description: In the `GovernorMinimumWeightedVoteWindow::_getVotes` function, there's a casting of the `_windowStart` value to uint32:

```
uint256 _startPos = _upperLookupRecent(_account, uint32(_windowStart), _numCheckpoints);
```

This casting limits the maximum timestamp that can be handled to 4,294,967,295 (year 2106). After this date, any attempt to cast a larger timestamp to uint32 will cause a transaction revert due to an arithmetic overflow error.

Impact: The impact of this issue is currently minimal as it will not manifest for next 80 years. After this period, the contract will fail to process votes, effectively breaking core functionality.

Recommended Mitigation: Consider replacing uint32 with uint64 in the casting operation.

Wormhole Foundation Acknowledged.

Cyfrin: Acknowledged.

7.2.3 Proposal deadline extension is allowed even in `Pending` state

Description: `HubProposalExtender::extendProposal` function allows the extension of proposal deadlines even when the proposal is in a `Pending` state. This allows the `VoteExtenderAdmin` to extend the deadline of a proposal before the voting period has even started.

However, during the `Pending` state, spoke chains cannot aggregate votes to send to the hub pool, making the extension ineffective for its intended purpose of allowing more time for vote aggregation in scenarios where Wormhole queries is not able to submit messages to the Hub chain.

```
function extendProposal(uint256 _proposalId) external {
    // ... (other checks)
    IGovernor.ProposalState state = governor.state(_proposalId);
    if (state != IGovernor.ProposalState.Active && state != IGovernor.ProposalState.Pending) {
        revert ProposalCannotBeExtended();
    }
    // ... (extension logic)
}
```

Recommended Mitigation: Consider modifying the `extendProposal` function to only allow extensions for Active proposals.

Wormhole Foundation Fixed in PR 170.

Cyfrin: Verified.

7.2.4 Missing deployment script for `HubEvmSpokeAggregateProposer`

Description: Deployment script in `DEployHubContractsBaseImpls.s.sol` does not include code for deploying `HubEvmSpokeAggregateProposer`.

Recommended Mitigation: Consider updating the script to deploy `HubEvmSpokeAggregateProposer` with documented initial values for `maxQueryTimestampOffset` parameter.

Wormhole Foundation Acknowledged.

Cyfrin: Acknowledged.

7.2.5 Missing implementation to cast fractional vote with signature

Description: On SpokeVoteAggregator, there is no function to sign a voting message with fractional voting params. While a user can cast nominal votes via signature, there is no provision to cast fractional votes via signature.

Recommended Mitigation: Consider adding a function that allows users to sign a voting message by passing fractional voting params.

Wormhole Foundation Acknowledged.

Cyfrin: Acknowledged.

7.2.6 Inconsistent documentation related to the deployment of `SpokeMessageExecutor`

Description: In the project README, the governance upgrade path for the `SpokeMessageExecutor` is given as follows

```
`SpokeMessageExecutor`: Deploy a new contract and then update the hub dispatcher to call the new spoke
↪  message executor.
```

SpokeMessageExecutor is upgradeable, so there is no need to deploy a new contract. Doing as suggested in the README infact leads to replay attack risk.

The README is outdated and inconsistent with the current system design.

Recommended Mitigation: Consider upgrading the project README to reflect the current design where `SpokeExecutor`is an upgradeable contract.

Wormhole Foundation Fixed in PR 176.

Cyfrin: Verified.

7.3 Gas Optimization

7.3.1 Hash comparison can be optimized

Description: `HubEvmSpokeDecoder::decode` does a keccak256 hash comparison to check if the blocks are finalized. One of the comparator is always constant, ie. keccak256(bytes("finalized")). Generating hashes is expensive and can be simplified by pre-calculating this value.

```
    if (keccak256(_ethCalls.requestFinality) != keccak256(bytes("finalized"))) {
      revert InvalidQueryBlock(_ethCalls.requestBlockId);
    }
```

This is gas expensive and hashing is unnecessary in this case.

Recommended Mitigation: Consider storing `keccak256(bytes("finalized"))` as a constant instead of computing every time.

Wormhole Foundation Fixed in PR 169.

Cyfrin: Verified.

7.3.2 `_checkProposalEligibility` gas optimization

Description: Current implementation of the `HubEvmSpokeAggregateProposer::_checkProposalEligibility` processes all spoke chain responses before checking if the proposal threshold is met. Additionally, the hub vote weight is added to the total vote weight after all votes from spoke chains are counted.

This approach may lead to unnecessary gas consumption in cases where the proposer's voting power on the hub chain alone, or combined with a subset of spoke chains, is sufficient to meet the proposal threshold.

Recommended Mitigation: Consider adding the hub vote weight first instead of adding it last. Modify the function to check the proposal threshold after adding the hub vote weight and after processing each spoke chain response. This allows for early termination if the threshold is met, potentially saving gas on unnecessary computations.

```
function _checkProposalEligibility(bytes memory _queryResponseRaw, IWormhole.Signature[] memory
↪  _signatures)
    internal
    view
    returns (bool)
{
    // ... (initial setup)

    uint256 _hubVoteWeight = HUB_GOVERNOR.getVotes(msg.sender, _sharedQueryBlockTime / 1_000_000);
    _totalVoteWeight = _hubVoteWeight;

    uint256 _proposalThreshold = HUB_GOVERNOR.proposalThreshold();
    if (_totalVoteWeight >= _proposalThreshold) {
        return true;
    }

    for (uint256 i = 0; i < _queryResponse.responses.length; i++) {
        // ... (process each spoke chain response)
        uint256 _voteWeight = abi.decode(_ethCalls.result[0].result, (uint256));
        _totalVoteWeight += _voteWeight;

        if (_totalVoteWeight >= _proposalThreshold) {
            return true;
        }
    }

    return false;
```

```
}
```

Wormhole Foundation Acknowledged.

Cyfrin: Acknowledged.