DiamondDao protocol security report

Version 1.0

Maroutis

December 6, 2024

DiamondDao protocol security report

Maroutis

December, 2024

DiamondDao Audit Report

Prepared by: Maroutis

Table of Contents

Protocol Summary

Disclaimer

Maroutis makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of 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.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Details

Scope

```
contracts/
--- DiamondDao.sol
contracts/interfaces/
contracts/library/
```

Issues found

Severity	Number of issues found
High	0
Medium	3

Severity	Number of issues found
Low	4
Info	11
Total	18

Documentation Review Report:

Key Findings

> *"DMD Decentralized Autonomous Organization facilitates decentralized decision-making, ownership, and governance within the DMD ecosystem. The platform introduces a new tool that supports a voting-based decision-making process and promotes transparency in operations."*

- The `DiamondDao` contract implements a decentralized governance mechanism allowing proposals to be created, voted on, finalized, and executed.
- Validator candidates are recognized and can participate in voting based on their staked amount, aligning with the documentation's emphasis on staking and validator roles.

> *"User without a connected wallet: This user has access to the DMD DAO proposal list, DAO voting results, and historical proposals. However, they are unable to create proposals or vote in the decision-making process."*

> *"Validator Candidate with a connected wallet: This user meets the criteria to become a validator candidate within the DMD ecosystem and takes a central role in DAO voting. Validator candidates can create proposals and vote on any active proposals, including their own, during the voting phase."*

> *"DMD token holder with a connected wallet: … These users can stake on one or several validator candidates, granting them the authority to vote on their behalf."*

> Access the governance section of the platform to review and vote on proposals submitted by the community. While every DMD holder can create a proposal, only validator candidates can directly participate in the voting process.

- Any DMD token holder can create proposals.

- The DAO enforces role-based access controls, ensuring that only validator candidates can vote on proposals, while general token holders can stake on validators.
- Functions for creating and voting on proposals are restricted to validator candidates, preventing unauthorized users from performing these actions, in line with the documentation.
- The staking functionality allows DMD token holders to delegate their stake to validators, influencing voting power as described. However, this functionality is not included in this part of the audited codebase.

> *"There are two main voting phases in the DMD DAO: the proposal phase and the voting phase. Each phase lasts 14 days, with the voting phase starting immediately after the proposal phase concludes."*

> *"During the proposal phase, DMD holders can create proposals that are shared within the community. During the voting phase, the creation of new proposals is not allowed. However, every validator candidate can vote on the proposals from the Active list, except for the dismissed ones."*

- The `DiamondDao` contract manages distinct Proposal and Voting phases, each lasting the specified duration, ensuring that proposal creation and voting occur in their designated timeframes.
- The contract enforces that new proposals cannot be created during the Voting phase and that voting can only occur on active proposals, aligning with the documented behavior.
- Transitions between phases are handled by the `switchPhase` function, maintaining the correct sequence.

> *"There are three types of proposals supported by the DMD DAO community: open proposals, ecosystem parameter change proposals, and contract upgrade proposals."*

> *"An open proposal requires 33% of total DAO weight participation and 33% of exceeding yes votes."*

> *"An ecosystem parameters' change proposal requires 33% of total DAO weight participation and 33% of yes votes exceeding no votes."*

> *"A contract upgrade proposal requires 50% of total DAO weight participation and 50% of yes votes exceeding no votes."*

- The DAO categorizes proposals into Open, Ecosystem Parameter Change, and Contract Upgrade types based on the calldata and target contracts.
- The `countVotes` and quorum calculations enforce the participation and exceeding yes votes thresholds as specified for each proposal type.

- Depending on the proposal type, the execution logic appropriately handles fund transfers, parameter changes, or contract upgrades, ensuring that each proposal type's requirements are met before execution.
- Both the participation % and the exceeding Yes vote % are the same. The `quorumReached` function only checks for the exceeding Yes votes as it is enough for both conditions to be true or false.

> *"During this step, proposal fees paid for proposal creation are refunded to the creator for accepted proposals only. If the proposer fails to finalize their voting, any other ecosystem participant can finalize it by accessing the Historic Proposals list."*

- The DAO enforces a fixed proposal fee during the proposal creation process and refunds it upon successful finalization of accepted proposals, adhering to the documentation.
- The `finalize` function manages the refund of proposal fees for accepted proposals and the allocation of fees to the reinsert pot for declined proposals.

> *"The history of proposals is stored on the blockchain and is accessible in a separate tab of the DAO UI. Users can inspect historic proposals and finalize or execute those that require action."*

- Users can finalize any unfinalized proposal even after some time has passed.

Documentation findings misaligned with the codebase

> *"Contract upgrade proposals allow for updates to any contract owned by the DAO."*

- During deployment, the ownership of the `ProxyAdmin` remains with the deployer (`owner`) instead of being transferred to the DAO.

> *"Voting results are based on two main indexes: Exceeding Yes Answers % and Participation %. Each validator candidate has their own Voting Power (%), which depends on the weight of their stake within the entire DAO DMD possession."*

- The `countVotes` function uses a snapshot of votes but recalculates quorum based on the current `totalStakingAmount` instead of the snapshot taken at the end of the Voting phase.

> *"There are three types of proposals supported by the DMD DAO community: open proposals, ecosystem parameter change proposals, and contract upgrade proposals."*

- In the codebase, each proposal has one overall Type. However, many calldatas of different types can be stacked in a single proposal.

> *"Each proposal type has a minimum required Exceeding Yes Answers value that must be reached for the proposal to be accepted."*

- When `totalStakedAmount` is zero or extremely low, proposals will always pass as the quorum calculations are proportional to the total staked amount.

> *"Every accepted proposal needs to be executed to implement the decision. Execution can occur during the new DAO phase following the phase in which the proposal was created and finalized. … There is a 28-day timeframe for proposal execution (the duration of the next proposal and voting phase)."*

> *"The proposal fee is fixed at 20 DMD for all proposal types. This fee can be adjusted by submitting an ecosystem parameter change proposal."*

- The proposal fee is set at deploy time and chosen by the deployer. In the existing hardhat tests, the fee chosen is 50 DMD.

> *"Finalization requires the payment of network fees."* A proposal can be dismissed by its creator during the proposal phase; however, both the creation (proposal) fee and the network fee will be forfeited.

- There is no such thing as network fees in the codebase. Only the proposal fee is created
- When canceling a proposal, the proposer forfeits his proposal fee only.

Testing Report

Results from static analysis tools

Static analysis reports do not showcase any severe issues. Some tools propose some code improvements for better gas usage and logic implementation.

Results from existing tests

The existing Hardhat tests achieve a high level of coverage and validation. The coverage is over 90% for the contracts in scope. A total of 86 tests pass successfully, covering a broad range of functionalities including initialization, phase transitions, proposal creation, voting, finalization, execution, and reentrancy protection. These tests validate both valid and invalid scenarios, ensuring that the DAO behaves correctly under normal conditions and when presented with invalid inputs.

The tests follow a simple logic. Basically, for every functionality (initialization, switchPhase, propose, cancel, vote…):

- The first few tests check for invalid arguments
- The following tests then validate the correct execution of every functionality
- Then the last tests validate the correct end states.

For more details of the functionalities covered:

- **Initialization:** Checks invalid arguments, start timestamps, and prevents re-initialization.
- **Phases:** Ensures that `switchPhase` transitions between Proposal/Voting phases correctly, updates proposal states, and respects timing assumptions.
- **Proposals:** Includes tests for invalid proposals, limits on new proposals, and ensuring no proposals are created during the wrong phase.
- **Voting:** Validates that only active validators can vote, ensures votes are recorded and tallied properly, and confirms that voting results match expected outcomes.
- **Finalization & Execution:** Verifies correct statistics updates, fee refunds, validate thresholds, and the ability to execute accepted proposals under normal conditions.
- **Reentrancy:** Confirms that calls to `execute` are guarded against reentrancy and that the DAO can self-upgrade under typical assumptions.

Identified gaps and improvements

While the Hardhat test suite already provided broad coverage and validated the main functionalities, the following gaps were noted before introducing the additional Foundry tests:

1. **Phase and timing edge cases:**
   Original tests did not deeply explore scenarios where `switchPhase` is called late, potentially leading to shortened phases and immediate subsequent transitions.
2. **Ownership and permission checks:**
   Functions guarded by `onlyOwner` in `ValueGuards` were never tested under conditions where an owner is properly set.
3. **Complex quorum scenarios and stake manipulations:**
   While the main tests confirm that voting and quorum mechanics work under normal conditions, they did not cover stale snapshots vs. current `totalStakingAmount`, allowing for stake manipulation after voting ends.
4. **Multiple calldatas proposals:**
   The original tests assumed straightforward proposals. More intricate proposals with multiple calldatas and different intended types were not explored.

5. **Upgrade path validation:**
   The tests did not confirm whether governance could autonomously upgrade the DAO if the `ProxyAdmin` remained controlled by a privileged external party.
6. **More edge cases**: Edge cases like empty total stake amount are not considered.

Additional tests cases

Subsequently, additional Foundry tests were introduced to explore edge cases and scenarios not thoroughly covered by the initial Hardhat tests :

1. **Phase delays:**
   - **`testSwitchPhaseDurationReducedDueToDelay`**: Confirming that calling `switchPhase` after the current phase's end can shorten the subsequent phase duration, potentially allowing rapid phase transitions.
   - **`testVotingIsCannotBeDoneAfterManyStateTransition`**: Ensures voting on older proposals cannot occur after multiple transitions.
2. **Proposal finalization & execution windows:**
   - **`testPreventFinalizationBeforeVotingPhaseEnds`**: Ensures proposals cannot be finalized prematurely.
   - **`testMissedFinalizeCanStillBeFinalizedDuringNextPhase`**: Demonstrates that unfinalized proposals remain finalizable in subsequent phases, even several phases later.
   - **`testExecutionWithin28DayWindow1`** & **`testExecutionWithin28DayWindow2`** : Validate executing proposals within the allowed window post-finalization.
   - **`testPreventExecutionOutside28DayWindow`**: Shows that proposals become inexecuable after the allowed window. Even if this was already tested, the idea is to showcase that only `execute` is prevented but finalization is possible.
3. **Quorum & total stake manipulation:**
   - **`testQuorumStakeManipulationBetweenVotingAndSnapshot`**: Highlights that votes are snapshotted at Voting phase end, but quorum uses current `totalStakingAmount`. Post vote total stake changes can alter results.
   - **`testQuorumCalculationWithZeroOrSmallTotalStake`**: In the edge case where total stake is zero or negligible, proposals almost always pass.
4. **Proposals with multiple calldata:**

- **testProposalTypeDeterminedByLastCalldataOrUpgrade**: When multiple calldatas are present, the final proposal type may not reflect all intended actions. The last or a failing call may force a `ContractUpgrade` type, overshadowing `Open` or `EcosystemParameterChange` types.

5. **Owner initialization & OnlyOwner functions:**

   - **testOwnerNotInitialized**: The `owner()` variable is never set, making `onlyOwner` functions in `ValueGuards` permanently inaccessible. This either necessitates a governance workaround (upgrading the contract to assign an owner) or indicates a design flaw.

6. **ProxyAdmin Ownership:**

   - **testDaoCannotBeUpgradedBecauseProxyAdminIsCentralized**: Shows that if the `ProxyAdmin` is not transferred to the DAO, governance cannot upgrade the contract. This creates a centralization point.

Additional foundry test suite

To add the foundry setup:

- Run the following command `npm install --save-dev @nomicfoundation/hardhat-foundry`
- Then import it in your Hardhat config: `import "@nomicfoundation/hardhat-foundry";`
- To complete the setup, run `npx hardhat init-foundry`

Consider copying the following diff content below in an empty file and applying the changes with `git apply ...`.

Diff details

```
diff --git a/test/foundry/DiamondDao.t.sol b/test/foundry/DiamondDao.t.sol
new file mode 100644
index 0000000..800cdc5
--- /dev/null
+++ b/test/foundry/DiamondDao.t.sol
@@ -0,0 +1,915 @@
+// SPDX-License-Identifier: MIT
+pragma solidity ^0.8.0;
+
+import "forge-std/Test.sol";
```

```
+import "../../contracts/DiamondDao.sol";
+import "../../contracts/mocks/MockValidatorSetHbbft.sol";
+import "../../contracts/mocks/MockStakingHbbft.sol";
+import {
+    DaoPhase,
+    Phase,
+    Proposal,
+    ProposalState,
+    ProposalStatistic,
+    Vote,
+    VoteRecord,
+    VotingResult,
+    ProposalType
+} from "../../contracts/library/DaoStructs.sol";
+
+import {TransparentUpgradeableProxy} from "@openzeppelin/contracts/
    proxy/transparent/TransparentUpgradeableProxy.sol";
+import {ProxyAdmin} from "@openzeppelin//contracts/proxy/transparent/
    ProxyAdmin.sol";
+
+contract DiamondDaoTest is Test {
+    DiamondDao dao;
+    MockValidatorSetHbbft mockValidatorSet;
+    MockStakingHbbft mockStaking;
+    // ProxyAdmin proxyAdmin;
+
+    address reinsertPot = address(0x1);
+    address immutable owner = makeAddr("owner");
+    uint256 createProposalFee = 50 ether;
+    uint256 governancePotValue = 500 ether;
+
+    address[] users;
+
+    function setUp() public {
+        // Initialize users
+        for (uint256 i = 0; i < 20; i++) {
+            address user = vm.addr(i + 1);
+            users.push(user);
+            vm.deal(user, 1000 ether);
+        }
+
+        // Deploy mock contracts
+        mockValidatorSet = new MockValidatorSetHbbft();
+        mockStaking = new MockStakingHbbft(address(mockValidatorSet));
+
+        // Deploy the implementation contract
+        DiamondDao daoImplementation = new DiamondDao();
+
+        // Prepare initializer data
+        bytes memory initializerData = abi.encodeWithSignature(
+            "initialize(address,address,address,address,uint256,uint64
```

```
        )",
60  +             address(mockValidatorSet),
61  +             address(mockStaking),
62  +             reinsertPot,
63  +             address(0), // _txPermission (ethers.ZeroAddress)
64  +             createProposalFee,
65  +             uint64(block.timestamp + 1)
66  +         );
67  +
68  +         // Deploy the proxy contract
69  +         TransparentUpgradeableProxy daoProxy =
70  +             new TransparentUpgradeableProxy(address(daoImplementation)
        , owner, initializerData);
71  +
72  +         // Attach the DAO interface to the proxy address
73  +         dao = DiamondDao(payable(address(daoProxy)));
74  +     }
75  +     // Helper Functions
76  +
77  +     function createProposal(
78  +         address proposer,
79  +         string memory description,
80  +         address[] memory targets,
81  +         uint256[] memory values,
82  +         bytes[] memory calldatas
83  +     ) internal returns (uint256 proposalId) {
84  +         vm.startPrank(proposer);
85  +         // Ensure proposer has enough funds
86  +         vm.deal(proposer, createProposalFee);
87  +         proposalId = dao.hashProposal(targets, values, calldatas,
        description);
88  +         dao.propose{value: createProposalFee}(targets, values,
        calldatas, "title", description, "url");
89  +         vm.stopPrank();
90  +     }
91  +
92  +     function switchPhase() internal {
93  +         (, uint64 end,, Phase currentPhase) = dao.daoPhase();
94  +         vm.warp(end + 1);
95  +         dao.switchPhase();
96  +     }
97  +
98  +     function addValidatorsStake(address[] memory validators, uint256
        stakeAmount) internal {
99  +         for (uint256 i = 0; i < validators.length; i++) {
100 +             mockValidatorSet.add(validators[i], validators[i], true);
101 +             mockStaking.setStake(validators[i], stakeAmount);
102 +         }
103 +     }
104 +
105 +     function vote(uint256 proposalId, address[] memory voters, Vote
```

```
        voteType) internal {
106 +           for (uint256 i = 0; i < voters.length; i++) {
107 +               vm.prank(voters[i]);
108 +               dao.vote(proposalId, voteType);
109 +           }
110 +       }
111 +
112 +       // Tests
113 +
114 +       function testVotingIsCannotBeDoneAfterManyStateTransition() public
        {
115 +           address proposer = users[2];
116 +
117 +           // Create a proposal
118 +           address[] memory targets = new address[](1);
119 +           targets[0] = users[1];
120 +
121 +           uint256[] memory values = new uint256[](1);
122 +           values[0] = 100 ether;
123 +
124 +           bytes[] memory callDatas = new bytes[](1);
125 +           callDatas[0] = "";
126 +
127 +           uint256 proposalId = createProposal(proposer, "Test Proposal",
        targets, values, callDatas);
128 +
129 +           // Add validators
130 +           address[] memory votersList = new address[](5);
131 +           for (uint256 i = 0; i < 5; i++) {
132 +               votersList[i] = users[i + 3];
133 +           }
134 +           addValidatorsStake(votersList, 10 ether);
135 +
136 +           // Switch to Voting phase
137 +           switchPhase();
138 +
139 +           (,,, Phase phase) = dao.daoPhase();
140 +           assertEq(uint256(phase), 1); // Voting phase
141 +           // Voter 1 abstains
142 +           vm.prank(votersList[0]);
143 +           dao.vote(proposalId, Vote.Abstain);
144 +
145 +           // Switch phase again proposal --> Voting --> next proposal
146 +           switchPhase();
147 +           // Cannot vote in this phase.
148 +           (,,, phase) = dao.daoPhase();
149 +           assertEq(uint256(phase), 0); // proposal
150 +           vm.expectRevert(abi.encodeWithSelector(IDiamondDao.
        UnavailableInCurrentPhase.selector, phase));
151 +           // Voter 2 votes Yes
152 +           vm.prank(votersList[1]);
```

```
153 +         dao.vote(proposalId, Vote.Yes);
154 +
155 +         // Switch phase again proposal --> Voting --> next proposal
    --> next Voting
156 +         switchPhase();
157 +         // Cannot vote anymore because the proposal state is # Active
    even if the state phase is correct.
158 +         (,,, phase) = dao.daoPhase();
159 +         assertEq(uint256(phase), 1); // Voting
160 +         vm.expectRevert(
161 +             abi.encodeWithSelector(
162 +                 IDiamondDao.UnexpectedProposalState.selector,
    proposalId, dao.getProposal(proposalId).state
163 +             )
164 +         );
165 +         // Voter 3 votes No
166 +         vm.prank(votersList[2]);
167 +         dao.vote(proposalId, Vote.No);
168 +     }
169 +
170 +     function testPreventFinalizationBeforeVotingPhaseEnds() public {
171 +         address proposer = users[2];
172 +
173 +         // Create a proposal
174 +         address[] memory targets = new address[](1);
175 +         targets[0] = users[1];
176 +
177 +         uint256[] memory values = new uint256[](1);
178 +         values[0] = 100 ether;
179 +
180 +         bytes[] memory callDatas = new bytes[](1);
181 +         callDatas[0] = "";
182 +
183 +         uint256 proposalId = createProposal(proposer, "Test Proposal",
     targets, values, callDatas);
184 +
185 +         ProposalState state = dao.getProposal(proposalId).state;
186 +
187 +         // Attempt to finalize during Proposal phase
188 +         vm.expectRevert(abi.encodeWithSelector(IDiamondDao.
    UnexpectedProposalState.selector, proposalId, state));
189 +         dao.finalize(proposalId);
190 +
191 +         // Switch to Voting phase
192 +         switchPhase();
193 +
194 +         // Attempt to finalize during Voting phase
195 +         vm.expectRevert(
196 +             abi.encodeWithSelector(
197 +                 IDiamondDao.UnexpectedProposalState.selector,
    proposalId, dao.getProposal(proposalId).state
```

```
198 +             )
199 +         );
200 +         dao.finalize(proposalId);
201 +
202 +         // Need to switch phase again to be able to finalize the
    proposal (proposal --> Voting --> next proposal).
203 +         switchPhase();
204 +         dao.finalize(proposalId);
205 +     }
206 +
207 +     function testMissedFinalizeCanStillBeFinalizedDuringNextPhase()
    public {
208 +         address proposer = users[2];
209 +
210 +         // Create a proposal
211 +         address[] memory targets = new address[](1);
212 +         targets[0] = users[1];
213 +
214 +         uint256[] memory values = new uint256[](1);
215 +         values[0] = 100 ether;
216 +
217 +         bytes[] memory callDatas = new bytes[](1);
218 +         callDatas[0] = "";
219 +
220 +         uint256 proposalId = createProposal(proposer, "Test Proposal",
     targets, values, callDatas);
221 +
222 +         assertEq(uint256(dao.getProposal(proposalId).state), 0);
223 +         switchPhase(); // proposal --> Voting
224 +         assertEq(uint256(dao.getProposal(proposalId).state), 2);
225 +         switchPhase(); // proposal --> Voting --> next proposal
226 +
227 +         assertEq(uint256(dao.getProposal(proposalId).state), 3);
228 +
229 +         // We assume user failed or forgot to finalize in this phase.
230 +         // Someone called switchPhase again
231 +         // Notice that the state of the proposal stays equal to
    VotingFinished. So it can be called at any time. Even after more
    than 28 days
232 +
233 +         switchPhase(); // proposal --> Voting --> next proposal -->
    next Voting
234 +         assertEq(uint256(dao.getProposal(proposalId).state), 3);
235 +         assertEq(block.timestamp, 3 * (dao.DAO_PHASE_DURATION() + 1) +
     2);
236 +
237 +         // We can call switchPhase many times and finalize would still
     work
238 +         switchPhase(); // proposal --> Voting --> next proposal -->
    next Voting --> next proposal(1)
239 +         switchPhase(); // proposal --> Voting --> next proposal -->
```

```
        next Voting --> next proposal(2) --> next Voting(2)
240 +         assertEq(uint256(dao.getProposal(proposalId).state), 3);
241 +
242 +         // Finalization happens after 56 days since the end of the
        first voting phase
243 +         dao.finalize(proposalId);
244 +     }
245 +
246 +     function testExecutionWithin28DayWindow1() public {
247 +         address proposer = users[2];
248 +
249 +         // Create a proposal
250 +         address[] memory targets = new address[](1);
251 +         targets[0] = users[1];
252 +
253 +         uint256[] memory values = new uint256[](1);
254 +         values[0] = 100 ether;
255 +
256 +         bytes[] memory callDatas = new bytes[](1);
257 +         callDatas[0] = "";
258 +
259 +         uint256 proposalId = createProposal(proposer, "Test Proposal",
        targets, values, callDatas);
260 +
261 +         // Add validators
262 +         address[] memory votersList = new address[](5);
263 +         for (uint256 i = 0; i < 5; i++) {
264 +             votersList[i] = users[i + 3];
265 +         }
266 +         addValidatorsStake(votersList, 10 ether);
267 +
268 +         // Switch to Voting phase
269 +         switchPhase();
270 +
271 +         // Voters vote Yes
272 +         vote(proposalId, votersList, Vote.Yes);
273 +
274 +         // Switch phase to end Voting/ next proposal
275 +         switchPhase();
276 +
277 +         // Finalize proposal
278 +         dao.finalize(proposalId);
279 +
280 +         // Attempt to execute the proposal
281 +         address(dao).call{value: 100 ether}("");
282 +         vm.prank(proposer);
283 +         dao.execute(proposalId);
284 +     }
285 +
286 +     function testExecutionWithin28DayWindow2() public {
287 +         address proposer = users[2];
```

```
288 +
289 +         // Create a proposal
290 +         address[] memory targets = new address[](1);
291 +         targets[0] = users[1];
292 +
293 +         uint256[] memory values = new uint256[](1);
294 +         values[0] = 100 ether;
295 +
296 +         bytes[] memory callDatas = new bytes[](1);
297 +         callDatas[0] = "";
298 +
299 +         uint256 proposalId = createProposal(proposer, "Test Proposal",
     targets, values, callDatas);
300 +         values[0] = 10 ether;
301 +         uint256 secondProposalId = createProposal(proposer, "Test
    Proposal", targets, values, callDatas);
302 +
303 +         // Add validators
304 +         address[] memory votersList = new address[](5);
305 +         for (uint256 i = 0; i < 5; i++) {
306 +             votersList[i] = users[i + 3];
307 +         }
308 +         addValidatorsStake(votersList, 10 ether);
309 +
310 +         // Switch to Voting phase
311 +         switchPhase();
312 +
313 +         // Voters vote Yes
314 +         vote(proposalId, votersList, Vote.Yes);
315 +         vote(secondProposalId, votersList, Vote.Yes);
316 +
317 +         // Switch phase to end Voting/ next proposal
318 +         switchPhase();
319 +
320 +         // Finalize proposal
321 +         dao.finalize(proposalId);
322 +         dao.finalize(secondProposalId);
323 +
324 +         // Attempt to execute the proposal
325 +         address(dao).call{value: 100 ether}("");
326 +         vm.prank(proposer);
327 +         dao.execute(proposalId);
328 +
329 +         // For the second proposal, we switch again, 28 days total
    since the voting phase.
330 +         // // Switch phases
331 +         switchPhase(); // proposal --> Voting --> next proposal -->
    next Voting
332 +         // Attempt to execute the proposal
333 +         address(dao).call{value: 10 ether}("");
334 +         vm.prank(proposer);
```

```
335 +         dao.execute(secondProposalId);
336 +     }
337 +
338 +     function testPreventExecutionOutside28DayWindow() public {
339 +         address proposer = users[2];
340 +
341 +         // Create a proposal
342 +         address[] memory targets = new address[](1);
343 +         targets[0] = users[1];
344 +
345 +         uint256[] memory values = new uint256[](1);
346 +         values[0] = 100 ether;
347 +
348 +         bytes[] memory callDatas = new bytes[](1);
349 +         callDatas[0] = "";
350 +
351 +         uint256 proposalId = createProposal(proposer, "Test Proposal",
     targets, values, callDatas);
352 +
353 +         // Add validators
354 +         address[] memory votersList = new address[](5);
355 +         for (uint256 i = 0; i < 5; i++) {
356 +             votersList[i] = users[i + 3];
357 +         }
358 +         addValidatorsStake(votersList, 10 ether);
359 +
360 +         // Switch to Voting phase
361 +         switchPhase(); // proposal --> Voting
362 +
363 +         // Voters vote Yes
364 +         vote(proposalId, votersList, Vote.Yes);
365 +         address(dao).call{value: 100 ether}("");
366 +
367 +         // Switch phase to end Voting/ next proposal
368 +         switchPhase(); // proposal --> Voting --> next proposal
369 +
370 +         // Switch phases
371 +         switchPhase(); // proposal --> Voting --> next proposal -->
    next Voting
372 +         switchPhase(); // proposal --> Voting --> next proposal -->
    next Voting --> next proposal(2)
373 +
374 +         // Finalize proposal is possible
375 +         dao.finalize(proposalId);
376 +
377 +         // Attempt to execute the proposal is not possible
378 +         vm.expectRevert(abi.encodeWithSelector(IDiamondDao.
    OutsideExecutionWindow.selector, proposalId));
379 +         vm.prank(proposer);
380 +         dao.execute(proposalId);
381 +
```

```
382 +         // Switch again; but still not possible
383 +         switchPhase(); // proposal --> Voting --> next proposal -->
    next Voting --> next proposal(2) --> next Voting (2)
384 +         vm.expectRevert(abi.encodeWithSelector(IDiamondDao.
    OutsideExecutionWindow.selector, proposalId));
385 +         vm.prank(proposer);
386 +         dao.execute(proposalId);
387 +     }
388 +
389 +     function testQuorumStakeManipulationBetweenVotingAndSnapshot()
    public {
390 +         address proposer = users[2];
391 +         address voter = users[3];
392 +
393 +         // Set initial stake
394 +         mockValidatorSet.add(voter, voter, true);
395 +         mockValidatorSet.add(users[4], users[4], true);
396 +         mockStaking.setStake(voter, 10 ether);
397 +
398 +         assertEq(mockStaking.totalStakedAmount(), 10 ether);
399 +
400 +         // Create a proposal
401 +         address[] memory targets = new address[](1);
402 +         targets[0] = users[1];
403 +
404 +         uint256[] memory values = new uint256[](1);
405 +         values[0] = 100 ether;
406 +
407 +         bytes[] memory callDatas = new bytes[](1);
408 +         callDatas[0] = "";
409 +
410 +         uint256 proposalId = createProposal(proposer, "Test Proposal",
     targets, values, callDatas);
411 +
412 +         // Switch to Voting phase
413 +         switchPhase();
414 +
415 +         // Voter casts a vote
416 +         vm.prank(voter);
417 +         dao.vote(proposalId, Vote.Yes);
418 +
419 +         // Switch phase to end Voting
420 +         switchPhase(); //@note users stake snapshot is set here but
    not the total staked amount
421 +
422 +         VotingResult memory res = dao.countVotes(proposalId);
423 +         uint256 requiredExceeding = mockStaking.totalStakedAmount() *
    (33 * 100) / 10000;
424 +         assertGt(res.stakeYes, res.stakeNo + requiredExceeding); //
    Proposal is passing at this point
425 +
```

```
426 +         // A user massively increases their stake before finalization
427 +         mockStaking.setStake(users[4], 1000 ether);
428 +
429 +         // Get the vote counts
430 +         VotingResult memory result = dao.countVotes(proposalId);
431 +         requiredExceeding = mockStaking.totalStakedAmount() * (33 *
    100) / 10000;
432 +         assertLt(result.stakeYes, result.stakeNo + requiredExceeding);
     // Now the proposal is not passing
433 +         // @note that this will also prevent validators who voted yes
    from increasing their staked amount in order to not decline the
    proposal due to increase in total staking
434 +         // while users who voted no will be incentivized to increase
    their stake
435 +         // users voting abstain will also negatively impact the vote
    if their stake increase.
436 +         // should be using a timepoint for the totalSupply also like
    in here https://github.com/OpenZeppelin/openzeppelin-contracts/blob/
    a5c4cd8182103aa96c2147433bf1bfb8fde63ca9/contracts/governance/
    extensions/GovernorVotesQuorumFraction.sol#L69-L74
437 +
438 +         // Finalize proposal
439 +         dao.finalize(proposalId);
440 +
441 +         // Check if the increased stake affected voting power
442 +         assertEq(uint256(dao.getProposal(proposalId).state), uint256(
    ProposalState.Declined)); // not passing
443 +     }
444 +     // Upgrades
445 +     // 1. Test for ProposalType.Open
446 +     function testExecuteOpenProposal() public {
447 +         address proposer = users[2];
448 +         address recipient = users[3];
449 +         uint256 amount = 100 ether;
450 +
451 +         // Fund the DAO governance pot
452 +         address(dao).call{value: amount}("");
453 +
454 +         // Create a proposal to transfer funds to recipient
455 +         address[] memory targets = new address[](1);
456 +         targets[0] = recipient;
457 +
458 +         uint256[] memory values = new uint256[](1);
459 +         values[0] = amount;
460 +
461 +         bytes[] memory callDatas = new bytes[](1);
462 +         callDatas[0] = "";
463 +
464 +         uint256 proposalId = createProposal(proposer, "Transfer funds
    to recipient", targets, values, callDatas);
465 +
```

```
466 +         // Add validators
467 +         address[] memory votersList = new address[](5);
468 +         for (uint256 i = 0; i < 5; i++) {
469 +             votersList[i] = users[i + 4];
470 +         }
471 +         addValidatorsStake(votersList, 10 ether);
472 +
473 +         // Switch to Voting phase
474 +         switchPhase();
475 +
476 +         // Voters vote Yes
477 +         vote(proposalId, votersList, Vote.Yes);
478 +
479 +         // Switch phase to end Voting
480 +         switchPhase();
481 +
482 +         // Finalize proposal
483 +         dao.finalize(proposalId);
484 +
485 +         // Verify that the proposal type is Open
486 +         Proposal memory proposal = dao.getProposal(proposalId);
487 +         assertEq(uint256(proposal.proposalType), uint256(ProposalType.
    Open));
488 +
489 +         // Check recipient balance before execution
490 +         uint256 recipientBalanceBefore = recipient.balance;
491 +
492 +         // Execute proposal
493 +         vm.prank(proposer);
494 +         dao.execute(proposalId);
495 +
496 +         // Check recipient balance after execution
497 +         uint256 recipientBalanceAfter = recipient.balance;
498 +
499 +         // Verify that the funds were transferred
500 +         assertEq(recipientBalanceAfter - recipientBalanceBefore,
    amount);
501 +
502 +         // Verify that the governance pot decreased
503 +         assertEq(address(dao).balance, 0);
504 +     }
505 +
506 +     // 2. Test for ProposalType.EcosystemParameterChange
507 +     function testExecuteEcosystemParameterChangeProposal() public {
508 +         address proposer = users[2];
509 +         uint256 newDelegatorMinStake = 150 ether;
510 +
511 +         //set setChangeAbleParameters in Mock contract
512 +         string memory setter = "setDelegatorMinStake(uint256)";
513 +         string memory getter = "delegatorMinStake()";
514 +         uint256[] memory params = new uint256[](5);
```

```
515 +         params[0] = 5000000000000000000;
516 +         params[1] = 10000000000000000000;
517 +         params[2] = 15000000000000000000;
518 +         params[3] = 20000000000000000000;
519 +         params[4] = 25000000000000000000;
520 +         mockStaking.setAllowedChangeableParameter(setter, getter,
    params);
521 +
522 +         // Prepare calldata to change delegatorMinStake
523 +         bytes memory calldata_ = abi.encodeWithSignature("
    setDelegatorMinStake(uint256)", newDelegatorMinStake);
524 +
525 +         address[] memory targets = new address[](1);
526 +         targets[0] = address(mockStaking);
527 +
528 +         uint256[] memory values = new uint256[](1);
529 +         values[0] = 0;
530 +
531 +         bytes[] memory callDatas = new bytes[](1);
532 +         callDatas[0] = calldata_;
533 +
534 +         uint256 proposalId = createProposal(proposer, "Change
    delegatorMinStake", targets, values, callDatas);
535 +
536 +         // Add validators
537 +         address[] memory votersList = new address[](5);
538 +         for (uint256 i = 0; i < 5; i++) {
539 +             votersList[i] = users[i + 3];
540 +         }
541 +         addValidatorsStake(votersList, 10 ether);
542 +
543 +         // Switch to Voting phase
544 +         switchPhase();
545 +
546 +         // Voters vote Yes
547 +         vote(proposalId, votersList, Vote.Yes);
548 +
549 +         // Switch phase to end Voting
550 +         switchPhase();
551 +
552 +         // Finalize proposal
553 +         dao.finalize(proposalId);
554 +
555 +         // Verify that the proposal type is EcosystemParameterChange
556 +         Proposal memory proposal = dao.getProposal(proposalId);
557 +         assertEq(uint256(proposal.proposalType), uint256(ProposalType.
    EcosystemParameterChange));
558 +
559 +         // Execute proposal
560 +         vm.prank(proposer);
561 +         dao.execute(proposalId);
```

```
562 +
563 +         // Verify that the parameter was updated
564 +         assertEq(mockStaking.delegatorMinStake(), newDelegatorMinStake
    );
565 +     }
566 +
567 +     // // 3. Test for ProposalType.ContractUpgrade
568 +     function testExecuteContractUpgradeProposal() public {
569 +         address proposer = users[2];
570 +
571 +         // Deploy new implementation
572 +         DiamondDao newImplementation = new DiamondDao();
573 +         // Get proxyAdmin from storage
574 +         bytes32 proxyAdmin =
575 +             vm.load(address(dao), bytes32(uint256(0
    xb53127684a568b3173ae13b9f8a6016e243e63b6e8ee1178d6a717850b5d6103)))
    ;
576 +
577 +         vm.prank(owner);
578 +         ProxyAdmin(address(uint160(uint256(proxyAdmin)))).
    transferOwnership(address(dao));
579 +
580 +         // Prepare calldata to call upgrade on ProxyAdmin
581 +         bytes memory calldata_ =
582 +             abi.encodeWithSelector(ProxyAdmin.upgradeAndCall.selector,
     address(dao), address(newImplementation), "");
583 +
584 +         address[] memory targets = new address[](1);
585 +         targets[0] = address(uint160(uint256(proxyAdmin)));
586 +
587 +         uint256[] memory values = new uint256[](1);
588 +         values[0] = 0;
589 +
590 +         bytes[] memory callDatas = new bytes[](1);
591 +         callDatas[0] = calldata_;
592 +
593 +         uint256 proposalId = createProposal(proposer, "Upgrade DAO
    Contract", targets, values, callDatas);
594 +
595 +         // Add validators
596 +         address[] memory votersList = new address[](5);
597 +         for (uint256 i = 0; i < 5; i++) {
598 +             votersList[i] = users[i + 3];
599 +         }
600 +         addValidatorsStake(votersList, 10 ether);
601 +
602 +         // Switch to Voting phase
603 +         switchPhase();
604 +
605 +         // Voters vote Yes
606 +         vote(proposalId, votersList, Vote.Yes);
```

```
607 +
608 +         // Switch phase to end Voting
609 +         switchPhase();
610 +
611 +         // Finalize proposal
612 +         dao.finalize(proposalId);
613 +
614 +         // Verify that the proposal type is ContractUpgrade
615 +         Proposal memory proposal = dao.getProposal(proposalId);
616 +         assertEq(uint256(proposal.proposalType), uint256(ProposalType.
        ContractUpgrade));
617 +
618 +         // Execute proposal
619 +         vm.prank(proposer);
620 +         dao.execute(proposalId);
621 +
622 +         // Verify that the DAO implementation was upgraded
623 +         bytes32 impl =
624 +             vm.load(address(dao), bytes32(uint256(0
        x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc)))
        ;
625 +         address implementation = address(uint160(uint256(impl)));
626 +         assertEq(implementation, address(newImplementation));
627 +     }
628 +
629 +     // Edge cases
630 +     // Test with zero total stake
631 +     function testQuorumCalculationWithZeroOrSmallTotalStake() public {
632 +         address proposer = users[2];
633 +
634 +         // Create a proposal
635 +         address[] memory targets = new address[](1);
636 +         targets[0] = users[1];
637 +
638 +         uint256[] memory values = new uint256[](1);
639 +         values[0] = 100 ether;
640 +
641 +         bytes[] memory callDatas = new bytes[](1);
642 +         callDatas[0] = "";
643 +
644 +         uint256 proposalId = createProposal(proposer, "Test Proposal",
         targets, values, callDatas);
645 +
646 +         // No validators added, total stake is zero
647 +
648 +         // Switch to Voting phase
649 +         switchPhase();
650 +
651 +         // No votes cast
652 +
653 +         // Switch phase to end Voting
```

```
654 +         switchPhase();
655 +
656 +         // Finalize proposal
657 +         dao.finalize(proposalId);
658 +
659 +         // Check proposal state
660 +         Proposal memory proposal = dao.getProposal(proposalId);
661 +
662 +         // Proposal is accepted
663 +         // @note Basically, when total stakes are 0 or very small at
        the time of quorum calculation, any proposal will get accepted (as
        long as Yes >= No)
664 +         assertEq(uint256(proposal.state), uint256(ProposalState.
        Accepted));
665 +     }
666 +
667 +     function testSwitchPhaseDurationReducedDueToDelay() public {
668 +         uint64 DAO_PHASE_DURATION = dao.DAO_PHASE_DURATION();
669 +
670 +         // Get the current phase end time
671 +         (, uint64 currentEnd,,) = dao.daoPhase();
672 +
673 +         // Simulate a delay: Advance time by 1 hour after the current
        phase end
674 +         uint64 timeAfterEnd = currentEnd + 1 hours;
675 +         vm.warp(timeAfterEnd);
676 +
677 +         // Assume switchPhase is called after the delay
678 +         dao.switchPhase();
679 +
680 +         // Get the new phase start and end times
681 +         (uint64 newStart, uint64 newEnd,,) = dao.daoPhase();
682 +
683 +         // The new phase end is set to newStart + DAO_PHASE_DURATION
684 +         uint64 expectedEnd = newStart + DAO_PHASE_DURATION;
685 +
686 +         // However, since block.timestamp > newStart, the actual
        duration remaining is less than 2 weeks
687 +
688 +         // Calculate the remaining duration of the new phase
689 +         uint64 remainingDuration = newEnd > uint64(block.timestamp) ?
        newEnd - uint64(block.timestamp) : 0;
690 +
691 +         // The remaining duration should be less than
        DAO_PHASE_DURATION
692 +         assertTrue(remainingDuration < DAO_PHASE_DURATION);
693 +
694 +         // Verify that the new phase will last less than the expected
        DAO_PHASE_DURATION
695 +         uint64 expectedRemainingDuration = expectedEnd - uint64(block.
        timestamp);
```

```
696 +         assertEq(remainingDuration, expectedRemainingDuration);
697 +
698 +         // Logging for more clarity
699 +         emit log_named_uint("Current Timestamp:", uint64(block.
    timestamp));
700 +         emit log_named_uint("New Phase Start:", newStart);
701 +         emit log_named_uint("New Phase End:", newEnd);
702 +         emit log_named_uint("DAO_PHASE_DURATION(2 weeks):",
    DAO_PHASE_DURATION);
703 +         emit log_named_uint("Remaining Duration of New Phase:",
    remainingDuration);
704 +     }
705 +
706 +     function testProposalTypeDeterminedByLastCalldataOrUpgrade()
    public {
707 +         address proposer = makeAddr("Proposer");
708 +
709 +         // calldata: Open
710 +         bytes memory calldata1 = "";
711 +
712 +         // calldata: ContractUpgrade
713 +         bytes memory calldata2 = abi.encodeWithSignature(
714 +             "setIsCoreContract(address,bool)", address(mockStaking),
    false
715 +         );
716 +
717 +         // calldata: EcosystemParameterChange
718 +         bytes memory calldata3 = abi.encodeWithSignature(
719 +             "setCreateProposalFee(uint256)", 40 ether
720 +         );
721 +
722 +         // Targets
723 +         address[] memory targets = new address[](3);
724 +         targets[0] = proposer;
725 +         targets[1] = address(dao);
726 +         targets[2] = address(dao);
727 +
728 +         uint256[] memory values = new uint256[](3);
729 +         values[0] = 20 ether;
730 +         values[0] = 0;
731 +         values[1] = 0;
732 +
733 +         address(dao).call{value: 20 ether}("");
734 +
735 +         bytes[] memory callDatas = new bytes[](3);
736 +         callDatas[0] = calldata1;
737 +         callDatas[1] = calldata2;
738 +         callDatas[2] = calldata3;
739 +
740 +         uint256 proposalId = createProposal(
741 +             proposer,
```

```
742 +            "Test multiple calldatas",
743 +            targets,
744 +            values,
745 +            callDatas
746 +        );
747 +
748 +        // Add validators
749 +        address[] memory votersList = new address[](5);
750 +        for (uint256 i = 0; i < 5; i++) {
751 +            votersList[i] = users[i + 4];
752 +        }
753 +        addValidatorsStake(votersList, 10 ether);
754 +
755 +        // Switch to Voting phase
756 +        switchPhase();
757 +
758 +        // Voters vote Yes
759 +        vote(proposalId, votersList, Vote.Yes);
760 +
761 +        // Switch phase to end Voting
762 +        switchPhase();
763 +
764 +        // Someone finalizes the proposal
765 +        dao.finalize(proposalId);
766 +
767 +        // Verify that the proposal type is EcosystemParameterChange
    which is determined by last calldata
768 +        Proposal memory proposal = dao.getProposal(proposalId);
769 +        assertEq(uint256(proposal.proposalType), uint256(ProposalType.
    ContractUpgrade));
770 +
771 +        // Now, attempt to execute the proposal as nonProposer
772 +        vm.prank(proposer);
773 +        dao.execute(proposalId);
774 +
775 +        // Verify that the parameters were updated
776 +        assertEq(dao.createProposalFee(), 40 ether);
777 +        assertEq(dao.isCoreContract(address(mockStaking)), false);
778 +        uint256 oldFee = 50 ether;
779 +        assertEq(proposer.balance - oldFee, 0); //@note Since, the
    proposal Type for everything is seen as Upgrade, the execValue of
    the first Open calldata was 0 instead of 20 ether as set in the
    value.
780 +    }
781 +
782 +    function
    testExecuteEcosystemParameterChangeProposalBypassesWithinRangeModifier
    () public {
783 +        address proposer = users[2];
784 +        uint256 newDelegatorMinStake = 500 ether;
785 +
```

```
786 +         //set setChangeAbleParameters in Mock contract
787 +         string memory setter = "setDelegatorMinStake(uint256)";
788 +         string memory getter = "delegatorMinStake()";
789 +         uint256[] memory params = new uint256[](5);
790 +         params[0] = 5000000000000000000;
791 +         params[1] = 10000000000000000000;
792 +         params[2] = 15000000000000000000;
793 +         params[3] = 20000000000000000000;
794 +         params[4] = 25000000000000000000;
795 +         mockStaking.setAllowedChangeableParameter(setter, getter,
    params);
796 +
797 +         // calldata: ContractUpgrade
798 +         bytes memory calldata1 = abi.encodeWithSignature(
799 +             "setIsCoreContract(address,bool)", address(mockStaking),
    false
800 +         );
801 +
802 +         // Prepare calldata to change delegatorMinStake
803 +         bytes memory calldata2 = abi.encodeWithSignature("
    setDelegatorMinStake(uint256)", newDelegatorMinStake);
804 +
805 +         address[] memory targets = new address[](2);
806 +         targets[0] = address(dao);
807 +         targets[1] = address(mockStaking);
808 +
809 +         uint256[] memory values = new uint256[](2);
810 +         values[0] = 0;
811 +         values[1] = 0;
812 +
813 +         bytes[] memory callDatas = new bytes[](2);
814 +         callDatas[0] = calldata1;
815 +         callDatas[1] = calldata2;
816 +
817 +         uint256 proposalId = createProposal(proposer, "Change
    delegatorMinStake", targets, values, callDatas);
818 +
819 +         // Add validators
820 +         address[] memory votersList = new address[](5);
821 +         for (uint256 i = 0; i < 5; i++) {
822 +             votersList[i] = users[i + 3];
823 +         }
824 +         addValidatorsStake(votersList, 10 ether);
825 +
826 +         // Switch to Voting phase
827 +         switchPhase();
828 +
829 +         // Voters vote Yes
830 +         vote(proposalId, votersList, Vote.Yes);
831 +
832 +         // Switch phase to end Voting
```

```
833 +         switchPhase();
834 +
835 +         // Finalize proposal
836 +         dao.finalize(proposalId);
837 +
838 +         // Verify that the proposal type is EcosystemParameterChange
839 +         Proposal memory proposal = dao.getProposal(proposalId);
840 +         assertEq(uint256(proposal.proposalType), uint256(ProposalType.
    ContractUpgrade));
841 +
842 +         // Execute proposal
843 +         vm.prank(proposer);
844 +         dao.execute(proposalId);
845 +
846 +         // Verify that the parameter was updated
847 +         assertEq(mockStaking.delegatorMinStake(), newDelegatorMinStake
    );
848 +     }
849 +
850 +     function testOwnerNotInitialized() public {
851 +         // Attempt to call a function that requires onlyOwner
852 +         string memory setter = "setDelegatorMinStake(uint256)";
853 +         string memory getter = "delegatorMinStake()";
854 +         uint256[] memory params = new uint256[](5);
855 +         params[0]= 50 ether;
856 +         params[1]= 100 ether;
857 +         params[2]= 150 ether;
858 +         params[3]= 200 ether;
859 +         params[4]= 250 ether;
860 +
861 +         vm.expectRevert(abi.encodeWithSignature("
    OwnableUnauthorizedAccount(address)", address(this)));
862 +         dao.setAllowedChangeableParameter(bytes4(abi.encode(setter)),
    bytes4(abi.encode(getter)), params);
863 +
864 +         assertEq(dao.owner(), address(0));
865 +     }
866 +
867 +     function testDaoCannotBeUpgradedBecauseProxyAdminIsCentralized()
    public {
868 +         address proposer = users[2];
869 +
870 +         // Deploy new implementation
871 +         DiamondDao newImplementation = new DiamondDao();
872 +         // Get proxyAdmin from storage
873 +         bytes32 proxyAdmin =
874 +             vm.load(address(dao), bytes32(uint256(0
    xb53127684a568b3173ae13b9f8a6016e243e63b6e8ee1178d6a717850b5d6103)))
    ;
875 +
876 +         assertEq(ProxyAdmin(address(uint160(uint256(proxyAdmin)))).
```

```
        owner(), owner);
877 +
878 +       // Prepare calldata to call upgrade on ProxyAdmin
879 +       bytes memory calldata_ =
880 +           abi.encodeWithSelector(ProxyAdmin.upgradeAndCall.selector,
         address(dao), address(newImplementation), "");
881 +
882 +       address[] memory targets = new address[](1);
883 +       targets[0] = address(uint160(uint256(proxyAdmin)));
884 +
885 +       uint256[] memory values = new uint256[](1);
886 +       values[0] = 0;
887 +
888 +       bytes[] memory callDatas = new bytes[](1);
889 +       callDatas[0] = calldata_;
890 +
891 +       uint256 proposalId = createProposal(proposer, "Upgrade DAO
        Contract", targets, values, callDatas);
892 +
893 +       // Add validators
894 +       address[] memory votersList = new address[](5);
895 +       for (uint256 i = 0; i < 5; i++) {
896 +           votersList[i] = users[i + 3];
897 +       }
898 +       addValidatorsStake(votersList, 10 ether);
899 +
900 +       // Switch to Voting phase
901 +       switchPhase();
902 +
903 +       // Voters vote Yes
904 +       vote(proposalId, votersList, Vote.Yes);
905 +
906 +       // Switch phase to end Voting
907 +       switchPhase();
908 +
909 +       // Finalize proposal
910 +       dao.finalize(proposalId);
911 +
912 +       // Verify that the proposal type is ContractUpgrade
913 +       Proposal memory proposal = dao.getProposal(proposalId);
914 +       assertEq(uint256(proposal.proposalType), uint256(ProposalType.
        ContractUpgrade));
915 +
916 +       // Execute proposal
917 +       vm.expectRevert();
918 +       vm.prank(proposer);
919 +       dao.execute(proposalId);
920 +    }
921 +}
```

Findings

Medium

[M-1] Phase duration will be shortened each time `switchPhase` is delayed

Description:

When `switchPhase` is called, the new phase's start and end timestamps are derived from the previous phase's `end` timestamp :

```
@>  uint64 newPhaseStart = daoPhase.end + 1;
    daoPhase.start = newPhaseStart;
@>  daoPhase.end = newPhaseStart + DAO_PHASE_DURATION;
    daoPhase.phase = newPhase;
```

If `block.timestamp` is already past the previous `end`, the new phase will in reality be shorter than the intended `DAO_PHASE_DURATION`. In the worst case, `block.timestamp` could surpass `newPhaseStart + DAO_PHASE_DURATION` immediately, allowing multiple rapid `switchPhase` calls and effectively skipping phases. This leads to irregular and potentially manipulated proposal cycles.

Impact:

- Phases lasts less than 2 weeks.
- In the worst case scenario, phases can become significantly shorter than intended, giving insufficient time for proposals to be created, voted on, or executed as expected.
- The governance process becomes unpredictable and possibly unfair, undermining trust in the DAO's decision-making mechanism.
- **Severity:** Medium
- **Likelihood:** High

Proof of Concept:

You can execute the following test by running `forge test --mt testSwitchPhaseDurationReducedDueT -vv`:

```
function testSwitchPhaseDurationReducedDueToDelay() public {
    uint64 DAO_PHASE_DURATION = dao.DAO_PHASE_DURATION();
```

```
3
4           // Get the current phase end time
5           (, uint64 currentEnd,,) = dao.daoPhase();
6
7           // Simulate a delay: Advance time by 1 hour after the current
                phase end
8           uint64 timeAfterEnd = currentEnd + 1 hours;
9           vm.warp(timeAfterEnd);
10
11          // Assume switchPhase is called after the delay
12          dao.switchPhase();
13
14          // Get the new phase start and end times
15          (uint64 newStart, uint64 newEnd,,) = dao.daoPhase();
16
17          // The new phase end is set to newStart + DAO_PHASE_DURATION
18          uint64 expectedEnd = newStart + DAO_PHASE_DURATION;
19
20          // However, since block.timestamp > newStart, the actual
                duration remaining is less than 2 weeks
21
22          // Calculate the remaining duration of the new phase
23          uint64 remainingDuration = newEnd > uint64(block.timestamp) ?
                newEnd - uint64(block.timestamp) : 0;
24
25          // The remaining duration should be less than
                DAO_PHASE_DURATION
26          assertTrue(remainingDuration < DAO_PHASE_DURATION);
27
28          // Verify that the new phase will last less than the expected
                DAO_PHASE_DURATION
29          uint64 expectedRemainingDuration = expectedEnd - uint64(block.
                timestamp);
30          assertEq(remainingDuration, expectedRemainingDuration);
31
32          // Logging for more clarity
33          emit log_named_uint("Current Timestamp:", uint64(block.
                timestamp));
34          emit log_named_uint("New Phase Start:", newStart);
35          emit log_named_uint("New Phase End:", newEnd);
36          emit log_named_uint("DAO_PHASE_DURATION(2 weeks):",
                DAO_PHASE_DURATION);
37          emit log_named_uint("Remaining Duration of New Phase:",
                remainingDuration);
38      }
```

Recommended Mitigation:

Ensure that the new phase duration is calculated from the current `block.timestamp` rather than from `daoPhase.end`.

```
    function switchPhase() external {
        if (block.timestamp < daoPhase.end) {
            return;
        }

        Phase newPhase = daoPhase.phase == Phase.Proposal ? Phase.
            Voting : Phase.Proposal;

-       uint64 newPhaseStart = daoPhase.end + 1;
-       daoPhase.start = newPhaseStart;
+       daoPhase.start = block.timestamp;
-       daoPhase.end = newPhaseStart + DAO_PHASE_DURATION;
+       daoPhase.end = block.timestamp + DAO_PHASE_DURATION;
        daoPhase.phase = newPhase;
```

[M-2] Finalization uses a vote snapshot but dynamically recalculates quorum with the current total stake

Description:

During the `finalize` step of a proposal, votes are counted from a fixed user `stakeAmount` snapshot taken at the end of the Voting phase.

```
    function _snapshotStakes(uint64 daoEpoch) private {
        address[] memory daoEpochVoters = _daoEpochVoters[daoEpoch].
            values();

        for (uint256 i = 0; i < daoEpochVoters.length; ++i) {
            address voter = daoEpochVoters[i];
@>          uint256 stakeAmount = stakingHbbft.stakeAmountTotal(voter);

            daoEpochStakeSnapshot[daoEpoch][voter] = stakeAmount;
        }
    }
```

However, the quorum and threshold checks are performed using the **current** `totalStakingAmount` rather than the snapshot value.

```
    function quorumReached(ProposalType _type, VotingResult memory
        result) public view returns (bool) {
        uint256 requiredExceeding;
@>      uint256 totalStakedAmount = _getTotalStakedAmount();

        if (_type == ProposalType.ContractUpgrade) {
            requiredExceeding = totalStakedAmount * (50 * 100) / 10000;
        } else {
            requiredExceeding = totalStakedAmount * (33 * 100) / 10000;
        }
```

```
10
11          return result.stakeYes >= result.stakeNo + requiredExceeding;
12      }
```

This creates a discrepancy: even though votes are locked-in at the Voting phase end, the effective quorum requirement can shift due to the total stake amount changes happening after the voting concludes but before the proposal is finalized.

Impact:

As a result:

- Validators who initially voted "Yes" could find their proposal failing if other participants increase their stake after voting ends, thereby raising the quorum threshold.
- Validators who initially voted "Yes" are discouraged from increasing their stake until `finalize()` is called. And this can happen each time they vote Yes for a proposal.
- Participants who voted "No" are incentivized to increase their stake post-vote to push the proposal below the required quorum.
- Even abstainers who increase their stake inadvertently raise the quorum target and may cause a proposal to fail.

This will encourage strategic stake manipulation.

- **Severity:** Medium
- **Likelihood:** Medium

Proof of Concept:

You can execute the following test by running `forge test --mt testQuorumStakeManipulationBetween -vv`:

```
1       function testQuorumStakeManipulationBetweenVotingAndSnapshot()
           public {
2           address proposer = users[2];
3           address voter = users[3];
4
5           // Set initial stake
6           mockValidatorSet.add(voter, voter, true);
7           mockValidatorSet.add(users[4], users[4], true);
8           mockStaking.setStake(voter, 10 ether);
9
10          assertEq(mockStaking.totalStakedAmount(), 10 ether);
11
12          // Create a proposal
13          address[] memory targets = new address[](1);
14          targets[0] = users[1];
15
```

```
16          uint256[] memory values = new uint256[](1);
17          values[0] = 100 ether;
18
19          bytes[] memory callDatas = new bytes[](1);
20          callDatas[0] = "";
21
22          uint256 proposalId = createProposal(proposer, "Test Proposal",
                targets, values, callDatas);
23
24          // Switch to Voting phase
25          switchPhase();
26
27          // Voter casts a vote
28          vm.prank(voter);
29          dao.vote(proposalId, Vote.Yes);
30
31          // Switch phase to end Voting
32          switchPhase(); //@note users stake snapshot is set here but not
                 the total staked amount
33
34          VotingResult memory res = dao.countVotes(proposalId);
35          uint256 requiredExceeding = mockStaking.totalStakedAmount() *
                (33 * 100) / 10000;
36          assertGt(res.stakeYes, res.stakeNo + requiredExceeding); //
                Proposal is passing at this point
37
38          // A user massively increases their stake before finalization
39          mockStaking.setStake(users[4], 1000 ether);
40
41          // Get the vote counts
42          VotingResult memory result = dao.countVotes(proposalId);
43          requiredExceeding = mockStaking.totalStakedAmount() * (33 *
                100) / 10000;
44          assertLt(result.stakeYes, result.stakeNo + requiredExceeding);
                // Now the proposal is not passing
45
46          // Finalize proposal
47          dao.finalize(proposalId);
48
49          // Check if the increased stake affected voting power
50          assertEq(uint256(dao.getProposal(proposalId).state), uint256(
                ProposalState.Declined)); // not passing
51      }
```

Recommended Mitigation:

The total stake amount has to be snapshotted in `_snapshotStakes` during the phase switch. This will make sure that the quorum calculation is fair and coherent. You can follow OZ implementation.

[M-3] Proposal type determination ignores intermediate calldatas which could bypass the `Open` or `EcosystemParameterChange` type calldatas value transfer

Description:

When a proposal contains multiple calldatas, the `_checkProposalType` function determines the proposal type based on the **last** processed calldata that successfully identifies a range or defaults to a `ContractUpgrade` if any of the calldatas is an Upgrade.

This logic means that if one or many of the intermediate `calldatas` of a single proposal were intended to be `Open` or `EcosystemParameterChange`, the final proposal type could be set for example to a `ContractUpgrade` for all `calldatas`.

```
            if (success && result.length > 0) {
                ICoreValueGuard.ParameterRange memory rangeData = abi.
                    decode(result, (ICoreValueGuard.ParameterRange));

                if (isCoreContract[targets[i]] && rangeData.range.
                    length > 0) {
                    _type = ProposalType.EcosystemParameterChange;

@>                  if (!ICoreValueGuard(targets[i]).
    isWithinAllowedRange(setFuncSelector, newVal)) {
                        revert NewValueOutOfRange(newVal);
                    }
                } else {
@>                  return ProposalType.ContractUpgrade;
                }
            } else {
                // If the call fails, treat it as a ContractUpgrade
                    proposal
@>              return ProposalType.ContractUpgrade;
            }
```

This will be an issue when stacking multiple calldatas where one or many of them are supposed to be of the `Open` type. Some `Open` type proposals are supposed to transfer value. However, because of the type of the whole proposal is set as `ContractUpgrade` or `EcosystemParameterChange` means that the `execValue` variable will be set to `0`. Thus, no value can be sent which will alter the execution logic of the proposal.

```
    function _executeOperations(
        address[] memory targets,
        uint256[] memory values,
        bytes[] memory calldatas,
        ProposalType proposalType
    ) private {
        for (uint256 i = 0; i < targets.length; ++i) {
```

```
8 @>          uint256 execValue = proposalType == ProposalType.Open ?
      values[i] : 0;
9             (bool success, bytes memory returndata) = targets[i].call{
                  value: execValue }
```

Also, according to the documentation on ecosystem parameter change proposals, parameters must remain within allowed ranges defined by the community. However, within the `_checkProposalType` function, if a preceding calldata is an upgrade, the proposal type defaults to `ContractUpgrade` and returns it. This classification occurs even if the proposal initially intended to perform an `EcosystemParameterChange`. As a result, it skips the `isWithinAllowedRange` checks enforced by `ICoreValueGuard` for the calldatas that would normally change some `EcosystemParameterChange`. This allows parameter changes that exceed defined thresholds to be applied without validation in the functions that do not enforce it directly in the logic.

For example, the `setCreateProposalFee` function of the `DiamondDao` enforces the `withinAllowedRange` check by default. So any fee change will still be checked against the range. However, the `setDelegatorMinStake` of the `MockStakingHbbft.sol` do not. Basically, a propose could choose any arbitrary `_minStake` and it will be set.

Impact:

- A proposal with multiple calldatas with one or multiple empty calldatas and with coresponding positive values, will not transfer governance funds during execution. The proposal will not be executed as expected.
- Arbitrary ecosystem parameters can be set for functions that do not enforce the `isWithinAllowedRange` checks directly in their logic. In this scope, only the contract `MockStakingHbbft.sol` is impacted, which makes this second impact low.
- **Severity:** Medium
- **Likelihood:** Medium

Proof of Concept:

For the first impact, you can execute the following test by running `forge test --mt testProposalTypeDeterminedByLastCalldataOrUpgrade -vv`:

```
1     function testProposalTypeDeterminedByLastCalldataOrUpgrade() public
          {
2         address proposer = makeAddr("Proposer");
3
4         // calldata: Open
5         bytes memory calldata1 = "";
6
7         // calldata: ContractUpgrade
8         bytes memory calldata2 = abi.encodeWithSignature(
```

```
            "setIsCoreContract(address,bool)", address(mockStaking),
                false
        );

        // calldata: EcosystemParameterChange
        bytes memory calldata3 = abi.encodeWithSignature(
            "setCreateProposalFee(uint256)", 40 ether
        );

        // Targets
        address[] memory targets = new address[](3);
        targets[0] = proposer;
        targets[1] = address(dao);
        targets[2] = address(dao);

        uint256[] memory values = new uint256[](3);
        values[0] = 20 ether;
        values[0] = 0;
        values[1] = 0;

        address(dao).call{value: 20 ether}("");

        bytes[] memory callDatas = new bytes[](3);
        callDatas[0] = calldata1;
        callDatas[1] = calldata2;
        callDatas[2] = calldata3;

        uint256 proposalId = createProposal(
            proposer,
            "Test multiple calldatas",
            targets,
            values,
            callDatas
        );

        // Add validators
        address[] memory votersList = new address[](5);
        for (uint256 i = 0; i < 5; i++) {
            votersList[i] = users[i + 4];
        }
        addValidatorsStake(votersList, 10 ether);

        // Switch to Voting phase
        switchPhase();

        // Voters vote Yes
        vote(proposalId, votersList, Vote.Yes);

        // Switch phase to end Voting
        switchPhase();

```

```
        // Someone finalizes the proposal
        dao.finalize(proposalId);

        // Verify that the proposal type is EcosystemParameterChange
           which is determined by last calldata
        Proposal memory proposal = dao.getProposal(proposalId);
        assertEq(uint256(proposal.proposalType), uint256(ProposalType.
           ContractUpgrade));

        // Now, attempt to execute the proposal as nonProposer
        vm.prank(proposer);
        dao.execute(proposalId);

        // Verify that the parameters were updated
        assertEq(dao.createProposalFee(), 40 ether);
        assertEq(dao.isCoreContract(address(mockStaking)), false);
        uint256 oldFee = 50 ether;
        assertEq(proposer.balance - oldFee, 0); //@note Since, the
           proposal Type for everything is seen as Upgrade, the
           execValue of the first Open calldata was 0 instead of 20
           ether as set in the value.
    }
```

For the second impact, you can execute the following test by running `forge test --mt testExecuteEcosystemParameterChangeProposalBypassesWithinRangeModifier -vv`:

```
    function
       testExecuteEcosystemParameterChangeProposalBypassesWithinRangeModifier
       () public {
        address proposer = users[2];
        uint256 newDelegatorMinStake = 500 ether;

        //set setChangeAbleParameters in Mock contract
        string memory setter = "setDelegatorMinStake(uint256)";
        string memory getter = "delegatorMinStake()";
        uint256[] memory params = new uint256[](5);
        params[0] = 500000000000000000;
        params[1] = 1000000000000000000;
        params[2] = 1500000000000000000;
        params[3] = 2000000000000000000;
        params[4] = 2500000000000000000;
        mockStaking.setAllowedChangeableParameter(setter, getter,
           params);

        // calldata: ContractUpgrade
        bytes memory calldata1 = abi.encodeWithSignature(
            "setIsCoreContract(address,bool)", address(mockStaking),
               false
        );
```

```
20
21          // Prepare calldata to change delegatorMinStake
22          bytes memory calldata2 = abi.encodeWithSignature("
                setDelegatorMinStake(uint256)", newDelegatorMinStake);
23
24          address[] memory targets = new address[](2);
25          targets[0] = address(dao);
26          targets[1] = address(mockStaking);
27
28          uint256[] memory values = new uint256[](2);
29          values[0] = 0;
30          values[1] = 0;
31
32          bytes[] memory callDatas = new bytes[](2);
33          callDatas[0] = calldata1;
34          callDatas[1] = calldata2;
35
36          uint256 proposalId = createProposal(proposer, "Change
                delegatorMinStake", targets, values, callDatas);
37
38          // Add validators
39          address[] memory votersList = new address[](5);
40          for (uint256 i = 0; i < 5; i++) {
41              votersList[i] = users[i + 3];
42          }
43          addValidatorsStake(votersList, 10 ether);
44
45          // Switch to Voting phase
46          switchPhase();
47
48          // Voters vote Yes
49          vote(proposalId, votersList, Vote.Yes);
50
51          // Switch phase to end Voting
52          switchPhase();
53
54          // Finalize proposal
55          dao.finalize(proposalId);
56
57          // Verify that the proposal type is EcosystemParameterChange
58          Proposal memory proposal = dao.getProposal(proposalId);
59          assertEq(uint256(proposal.proposalType), uint256(ProposalType.
                ContractUpgrade));
60
61          // Execute proposal
62          vm.prank(proposer);
63          dao.execute(proposalId);
64
65          // Verify that the parameter was updated
66          assertEq(mockStaking.delegatorMinStake(), newDelegatorMinStake)
                ;
```

```
67         }
```

Recommended Mitigation:

- Each calldata should have a corresponding `ProposalType`. Basically, treat each calldata individually.
- Use an array of proposal types `ProposalType[]` to store every calldata type for the proposal.
- Enforce `isWithinAllowedRange` modifier for every function that must accept parameters within a specific range.

Low

[L-1] The Dao's owwner not initialized, making `onlyOwner` functions unusable

Description:

The DAO contract inherits `Ownable` functionality but never invokes `__Ownable_init` or otherwise sets the owner during initialization. As a result, the `owner()` is permanently `address(0)`. This renders all `onlyOwner` restricted functions effectively inaccessible. In particular, functions like `setAllowedChangeableParameter` and `removeAllowedChangeableParameter` from the `ValueGuards` module cannot be called.

Impact:

If these functions are intended to be called by a legitimate owner (e.g., an admin or deployer with special authority), then this poses a significant governance limitation. To modify ecosystem parameters, one would need a workaround, such as passing a contract upgrade proposal to assign an owner first. This is likely not the original design intent. If the intention was for governance (rather than an owner) to control these parameters, the contract would have used `onlyGovernance` modifiers instead of `onlyOwner`.

I consider this issue to be of Low severity and medium Likelihood. However:

- If the `onlyOwner` functions are critical for ecosystem parameter management and intended for an authorized party: **Severity:** High.
  Likelihood: Medium.

Proof of Concept:

You can execute the following test by running `forge test --mt testOwnerNotInitialized -vv`:

```
    function testOwnerNotInitialized() public {
        // Attempt to call a function that requires onlyOwner
        string memory setter = "setDelegatorMinStake(uint256)";
        string memory getter = "delegatorMinStake()";
        uint256[] memory params = new uint256[](5);
        params[0]= 50 ether;
        params[1]= 100 ether;
        params[2]= 150 ether;
        params[3]= 200 ether;
        params[4]= 250 ether;

        vm.expectRevert(abi.encodeWithSignature("
            OwnableUnauthorizedAccount(address)", address(this)));
        dao.setAllowedChangeableParameter(bytes4(abi.encode(setter)),
            bytes4(abi.encode(getter)), params);

        assertEq(dao.owner(), address(0));
    }
```

Recommended Mitigation:

- Initialize ownership during the initialize function by calling `__Ownable_init()`
- If the intention is to delegate these changes to governance rather than a single owner, replace `onlyOwner` with `onlyGovernance`

[L-2] Zero or negligible total stake allows proposals to always pass

Description:

In extreme scenarios where the total staking amount is zero or extremely small, the quorum calculation will allow proposals to almost always pass. Because the quorum is calculated relative to total stake, if that stake is zero, any proposal effectively meets the quorum threshold. Consequently, proposals can be accepted without receiving any meaningful support.

```
    function quorumReached(ProposalType _type, VotingResult memory
        result) public view returns (bool) {
        uint256 requiredExceeding;
        uint256 totalStakedAmount = _getTotalStakedAmount();

        if (_type == ProposalType.ContractUpgrade) {
@>          requiredExceeding = totalStakedAmount * (50 * 100) / 10000;
        } else {
@>          requiredExceeding = totalStakedAmount * (33 * 100) / 10000;
        }

        return result.stakeYes >= result.stakeNo + requiredExceeding;
```

In this case, `requiredExceeding` will be 0 as long as `totalStakedAmount` is small enough (less than `5000`).

Impact:

With zero or negligible total stake,the quorum requirements and thresholds do not provide any meaningful resistance or require actual consensus.

- **Severity:** Medium.
- **Likelihood:** Very Low.

Proof of Concept:

You can execute the following test by running `forge test --mt testQuorumCalculationWithZeroOrSma -vv`:

```
    function testQuorumCalculationWithZeroOrSmallTotalStake() public {
        address proposer = users[2];

        // Create a proposal
        address[] memory targets = new address[](1);
        targets[0] = users[1];

        uint256[] memory values = new uint256[](1);
        values[0] = 100 ether;

        bytes[] memory callDatas = new bytes[](1);
        callDatas[0] = "";

        uint256 proposalId = createProposal(proposer, "Test Proposal",
            targets, values, callDatas);

        // No validators added, total stake is zero

        // Switch to Voting phase
        switchPhase();

        // No votes cast

        // Switch phase to end Voting
        switchPhase();

        // Finalize proposal
        dao.finalize(proposalId);

        // Check proposal state
        Proposal memory proposal = dao.getProposal(proposalId);

        // Proposal is accepted
        // @note Basically, when total stakes are 0 or very small at
```

```
                the time of quorum calculation, any proposal will get
                accepted (as long as Yes >= No)
34          assertEq(uint256(proposal.state), uint256(ProposalState.
                Accepted));
35      }
```

Recommended Mitigation:

- Ensure that proposals cannot pass without 0 votes.
- Consider implementing a base quorum that is independent of the total staked amount.

[L-3] In the scripts, when a DAO proxy is created, the `ProxyAdmin` ownership is not transferred to the DAO, blocking proposal type upgrades

Description:

The DAO uses a transparent proxy pattern controlled by a `ProxyAdmin` contract. For on-chain governance proposals to perform contract upgrades like deploying new implementations, the DAO must own the `ProxyAdmin`. However, the current deployment setup int he hardhat scripts repo never transfers the `ProxyAdmin` ownership to the DAO. As a result, the `ProxyAdmin` remains controlled by a privileged external address (e.g., the deployer), effectively preventing governance proposals from upgrading the DAO independently. The only time where the ownership transfer is made is in the ProposalExecution.spec.ts file :

```
1 await proxyAdmin.transferOwnership(daoAddress);
```

Unless the `ProxyAdmin` is manually transferred to the DAO, governance proposals attempting to upgrade the DAO contract will fail.

Impact:

This lack of proper ownership assignment prevents the DAO from freely executing upgrade proposals.

- **Severity:** Medium
- **Likelihood:** Low

Proof of Concept:

You can execute the following test by running `forge test --mt testQuorumCalculationWithZeroOrSma -vv`:

```
1     function testDaoCannotBeUpgradedBecauseProxyAdminIsCentralized()
          public {
2         address proposer = users[2];
3
4         // Deploy new implementation
```

```
5           DiamondDao newImplementation = new DiamondDao();
6           // Get proxyAdmin from storage
7           bytes32 proxyAdmin =
8               vm.load(address(dao), bytes32(uint256(0
                    xb53127684a568b3173ae13b9f8a6016e243e63b6e8ee1178d6a717850b5d6103
                    )));
9
10          assertEq(ProxyAdmin(address(uint160(uint256(proxyAdmin)))).
                owner(), owner);
11
12          // Prepare calldata to call upgrade on ProxyAdmin
13          bytes memory calldata_ =
14              abi.encodeWithSelector(ProxyAdmin.upgradeAndCall.selector,
                    address(dao), address(newImplementation), "");
15
16          address[] memory targets = new address[](1);
17          targets[0] = address(uint160(uint256(proxyAdmin)));
18
19          uint256[] memory values = new uint256[](1);
20          values[0] = 0;
21
22          bytes[] memory callDatas = new bytes[](1);
23          callDatas[0] = calldata_;
24
25          uint256 proposalId = createProposal(proposer, "Upgrade DAO
                Contract", targets, values, callDatas);
26
27          // Add validators
28          address[] memory votersList = new address[](5);
29          for (uint256 i = 0; i < 5; i++) {
30              votersList[i] = users[i + 3];
31          }
32          addValidatorsStake(votersList, 10 ether);
33
34          // Switch to Voting phase
35          switchPhase();
36
37          // Voters vote Yes
38          vote(proposalId, votersList, Vote.Yes);
39
40          // Switch phase to end Voting
41          switchPhase();
42
43          // Finalize proposal
44          dao.finalize(proposalId);
45
46          // Verify that the proposal type is ContractUpgrade
47          Proposal memory proposal = dao.getProposal(proposalId);
48          assertEq(uint256(proposal.proposalType), uint256(ProposalType.
                ContractUpgrade));
49
```

```
50          // Execute proposal
51          vm.expectRevert();
52          vm.prank(proposer);
53          dao.execute(proposalId);
54      }
```

Recommended Mitigation:

During deployment or initialization, explicitly transfer the ProxyAdmin ownership to the DAO contract.

[L-4] Proposals can be finalized at any time but execution is restricted to a specific window

Description:

According to the DiamondDAO documentation:

> The history of proposals is stored on the blockchain and is accessible in a separate tab of the DAO UI. Users can inspect historic proposals and finalize or execute those that require action.

This suggests that historical proposals can be finalized and executed at any time. However, execution of a finalized proposal is only possible within a certain window :

```
1       function _requireIsExecutable(uint256 _proposalId) private view {
2           Proposal memory proposal = getProposal(_proposalId);
3
4   @>      if (proposal.daoPhaseCount + 1 != daoPhaseCount) {
5               revert OutsideExecutionWindow(_proposalId);
6           }
```

If this execution window passes without action, proposals can no longer be executed. This mismatch in finalization and execution timing creates a scenario where proposals may become stuck in a finalized state without any practical way to enforce their intended actions.

Impact:

In practice, proposals that are not executed within the permissible timeframe become effectively worthless, even though they were successfully finalized. This deviates from the documentation.

- **Severity:** Medium
- **Likelihood:** Medium

Proof of Concept:

You can execute the following test by running `forge test --mt testPreventExecutionOutside28DayWi -vv`:

```
    function testPreventExecutionOutside28DayWindow() public {
        address proposer = users[2];

        // Create a proposal
        address[] memory targets = new address[](1);
        targets[0] = users[1];

        uint256[] memory values = new uint256[](1);
        values[0] = 100 ether;

        bytes[] memory callDatas = new bytes[](1);
        callDatas[0] = "";

        uint256 proposalId = createProposal(proposer, "Test Proposal",
            targets, values, callDatas);

        // Add validators
        address[] memory votersList = new address[](5);
        for (uint256 i = 0; i < 5; i++) {
            votersList[i] = users[i + 3];
        }
        addValidatorsStake(votersList, 10 ether);

        // Switch to Voting phase
        switchPhase(); // proposal --> Voting

        // Voters vote Yes
        vote(proposalId, votersList, Vote.Yes);
        address(dao).call{value: 100 ether}("");

        // Switch phase to end Voting/ next proposal
        switchPhase(); // proposal --> Voting --> next proposal

        // Switch phases
        switchPhase(); // proposal --> Voting --> next proposal -->
            next Voting
        switchPhase(); // proposal --> Voting --> next proposal -->
            next Voting --> next proposal(2)

        // Finalize proposal is possible
        dao.finalize(proposalId);

        // Attempt to execute the proposal is not possible
        vm.expectRevert(abi.encodeWithSelector(IDiamondDao.
            OutsideExecutionWindow.selector, proposalId));
        vm.prank(proposer);
        dao.execute(proposalId);

        // Switch again; but still not possible
        switchPhase(); // proposal --> Voting --> next proposal -->
```

```
                next Voting --> next proposal(2) --> next Voting (2)
47          vm.expectRevert(abi.encodeWithSelector(IDiamondDao.
                OutsideExecutionWindow.selector, proposalId));
48          vm.prank(proposer);
49          dao.execute(proposalId);
50      }
```

Recommended Mitigation:

Align the finalization and execution windows so that once a proposal is finalized, it can be executed at any point until some clearly defined, predictable deadline.

Informational

[I-1] Unused `_txPermission` parameter in `initialize` function

Description:

The `_txPermission` parameter, passed into the `initialize` function, is never referenced in the contract.

```
1       function initialize(
2           address _validatorSet,
3           address _stakingHbbft,
4           address _reinsertPot,
5           address _txPermission, //@note not used
6           uint256 _createProposalFee,
7           uint64 _startTimestamp
8       ) external initializer {
```

Recommended Mitigation:

Remove the `_txPermission` parameter if it's not intended for future functionality, or implement the logic associated with it to fulfill its intended role.

[I-2] Mismatch in proposal fee and mention of nonexistent network fees

Description:

The documentation states that the proposal fee is fixed at 20 DMD and can be changed via an ecosystem parameter proposal.

> The proposal fee is fixed at 20 DMD for all proposal types. This fee can be adjusted by submitting an ecosystem parameter change proposal.

However, the deployed contract and tests use a different fee (e.g., 50 DMD), and “network fees” referenced in the documentation do not appear in the code.

Recommended Mitigation: Synchronize the code and documentation by:

1. Defining the actual default proposal fee in code to match the documentation or updating the documentation to reflect the true default fee.
2. Removing references to network fees if not implemented, or implement them if they are intended to be part of the system.

[I-3] Race condition in `_executeOperations` due to limited `governancePot`

Description: When multiple `Open` proposals execute in quick succession, they may drain the `governancePot`. Subsequent proposals attempting to execute transfers could fail if insufficient funds remain. This creates a race condition where early proposal executors get fully funded, while later ones might not.

Recommended Mitigation: Consider adding logic to partially fulfill transfers or fail gracefully. For accepted `Open` proposal that do not get get executed because of lack of funds, you can consider a roll-over mechanism that allows them to be executed even after the 28 days window.

[I-4] Abstain votes are ineffective

Description: Currently, the contract calculates proposal acceptance based on exceeding yes/no percentages. Since participation thresholds is the same as the acceptance, abstain votes have no meaningful impact on the final outcome, despite being recorded. If threshold values are upgraded in the future, there could be a need to consider participation int he quorum determination, the current logic, which ignores abstain votes and does not rigorously enforce participation checks, will become insufficient.

Recommended Mitigation: Incorporate abstain votes into the participation calculations. You can even adjust acceptance logic to ensure that abstain votes influence whether a proposal meets minimum participation thresholds.

[I-5] Risk of malicious upgrades by a colluding validator majority

Description:

Even if highly unlikely, if a majority of validators collude to pass a malicious contract upgrade proposal, they could modify the system to their advantage, potentially draining funds or destroying the contract.

[I-6] Define and use `constant` variables instead of using literals

Rather than using constant literal values, create a constant state variable and reference it throughout the contract.

6 Found Instances

- Found in contracts/DiamondDao.sol Line: 169

```
createProposalFeeAllowedParams[4] = 50 ether;
```

- Found in contracts/DiamondDao.sol Line: 460

```
requiredExceeding = totalStakedAmount * (50 * 100) /
    10000;
```

- Found in contracts/DiamondDao.sol Line: 462

```
requiredExceeding = totalStakedAmount * (33 * 100) /
    10000;
```

[I-7] Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

12 Found Instances

- Found in contracts/interfaces/IDiamondDao.sol Line: 7

```
event ProposalCreated(
```

- Found in contracts/interfaces/IDiamondDao.sol Line: 19

```
event ProposalCanceled(address indexed proposer, uint256
    indexed proposalId, string reason);
```

- Found in contracts/interfaces/IDiamondDao.sol Line: 23

```
event VotingFinalized(address indexed caller, uint256 indexed
    proposalId, bool accepted);
```

- Found in contracts/interfaces/IDiamondDao.sol Line: 25

```
1     event SubmitVote(address indexed voter, uint256 indexed
          proposalId, Vote vote);
```

- Found in contracts/interfaces/IDiamondDao.sol Line: 27

```
1     event SubmitVoteWithReason(
```

- Found in contracts/interfaces/IDiamondDao.sol Line: 34

```
1     event SwitchDaoPhase(Phase phase, uint256 start, uint256 end);
```

- Found in contracts/interfaces/IDiamondDao.sol Line: 36

```
1     event SetCreateProposalFee(uint256 fee);
```

- Found in contracts/interfaces/IDiamondDao.sol Line: 38

```
1     event SetIsCoreContract(address contractAddress, bool isCore);
```

- Found in contracts/interfaces/IDiamondDao.sol Line: 40

```
1     event SetChangeAbleParameters(bool allowed, string setter,
          string getter, uint256[] params);
```

- Found in contracts/mocks/MockStakingHbbft.sol Line: 24

```
1     event SetDelegatorMinStake(uint256 minStake);
```

- Found in contracts/mocks/MockStakingHbbft.sol Line: 32

```
1     event SetChangeAbleParameter(
```

- Found in contracts/mocks/MockStakingHbbft.sol Line: 42

```
1     event RemoveChangeAbleParameter(string funcSelector);
```

[I-8] Modifiers invoked only once can be shoe-horned into the function to save gas

1 Found Instances

- Found in contracts/DiamondDao.sol Line: 111

```
1     modifier noUnfinalizedProposals() {
```

[I-9] Unused Custom Error

it is recommended that the definition be removed when custom error is unused

2 Found Instances

- Found in contracts/interfaces/IDiamondDao.sol Line: 55

```
error FunctionUpgradeNotAllowed(bytes4 funcSelector, address
    targetContract);
```

- Found in contracts/interfaces/IDiamondDao.sol Line: 56

```
error InvalidUpgradeValue(uint256 currentVal, uint256 newVal);
```

[I-10] Unused `_getCurrentValWithSelector` Function

Description: The `_getCurrentValWithSelector` function is defined but never called within the contract.

Recommended Mitigation: Remove the `_getCurrentValWithSelector` function if it is not intended for future use, or implement the logic that depends on its returned value to ensure it serves a meaningful purpose in the contract.

[I-11] Repeated Access to `currentPhaseProposals.length` in Loop Condition

Description: In the loop condition (`i < currentPhaseProposals.length`) of the `switchPhase` function, the contract repeatedly reads the `length` property of a storage array. Each read from storage is more expensive than reading from memory. Since the array length is constant during the loop's execution, accessing it once and storing it in a local variable before the loop runs can reduce gas consumption.

Recommended Mitigation: Cache the array length in a local variable before the loop:

```
uint256 proposalsLength = currentPhaseProposals.length;
for (uint256 i = 0; i < proposalsLength; ++i) {
    // loop body
}
```

Focus Area Results

This section provides a detailed analysis of each focus area with tests references.

Proposal Workflow

Proper Validation of Proposals:
The code ensures that each newly created proposal meets strict validation criteria before it is accepted. When a proposal is submitted, the contract checks for consistent array lengths across targets, values, and calldatas, ensuring that each proposed operation is well-defined. In addition, a nonzero proposal fee is required, and insufficient funds cause immediate reversion. By enforcing these requirements at creation time, the contract prevents malformed or incomplete proposals from entering the governance process.

- **Hardhat Tests:**
  - `propose` block:
    * *"should revert propose with empty targets array"*
    * *"should revert propose with targets.length != values.length"*
    * *"should revert propose with targets.length != calldatas.length"*
    * *"should revert propose without proposal fee payment"*

Prevention of Duplicate or Invalid Proposals:
The code hashes the proposal details (targets, values, calldatas, and description) to generate a unique identifier. Any attempt to recreate an identical proposal reverts, effectively preventing duplicates.

- **Hardhat Tests:**
  - `propose` block:
    * *"should revert propose if same proposal already exists"*

Safeguards Against Vote Manipulation or Unfinalized Proposals:
The contract enforces a strict lifecycle: a new proposal phase cannot begin if there are proposals from the previous phase that remain unfinalized. This prevents the buildup of unfinalized proposals that could be exploited or create governance deadlock. Additionally, the requirement to finalize proposals before moving forward ensures that the final outcome of each proposal is settled before new governance actions occur. While the code correctly enforces proposal finalization, it does not fully prevent certain stake manipulations after voting; an issue explored in the voting mechanisms section below.

- **Hardhat Tests:**
  - `propose` block:
    * *"should revert propose if there are unfinalized proposals in previous phases"*
- **Foundry Tests:**

- `testPreventFinalizationBeforeVotingPhaseEnds`: Ensures proposals aren't finalized prematurely.
- `testMissedFinalizeCanStillBeFinalizedDuringNextPhase`: Shows finalization is still possible much later, but at least the process is controlled to avoid lingering states.

Voting Mechanisms

Stake Snapshot Integrity:
The code takes a snapshot of votes at the end of the Voting phase to ensure that the final tally reflects the state of votes at a fixed moment in time. However, while votes themselves are snapshotted, the current total staked amount used for quorum calculations is not. This partial snapshotting leaves the door open for stake changes post-vote, allowing strategic manipulation of the effective quorum after all votes are cast but before finalization.

- **Hardhat Tests:**
  - `vote` countVotes:
    - *"should use stake amounts snapshot after voting finish"*
- **Foundry Tests:**
  - `testQuorumStakeManipulationBetweenVotingAndSnapshot`: Highlights that changing total stake after voting can shift quorum requirements, exposing a vulnerability in stake snapshot logic.

Accurate Vote Recording and Tallying:
The contract records each validator's vote and the associated staked amount at voting time, ensuring accurate aggregation of Yes/No votes. It distinguishes between proposal types and applies relevant thresholds for acceptance.

- **Hardhat Tests:**
  - `vote` block:
    - *"should submit vote by validator and emit event"*
    - *"should submit vote and save its data"*
  - `countVotes` block:
    - *"should use current stake amounts for active proposal"*
    - *"should use stake amounts snapshot after voting finish"*

  - `countVotes` block:
    * *"should use current stake amounts for active proposal"*
      Ensures vote tallying is accurate based on recorded votes.

Quorum Calculations, Including Edge Case Scenarios:
While quorum logic is implemented, the current approach calculates quorum based on the latest total stake rather than a fixed snapshot. This discrepancy allows validators or token holders to increase or decrease their stakes after voting ends to manipulate proposal outcomes. Also, in a scenario with negligible or zero total stake, the required thresholds become trivial and proposals may pass too easily.

- **Hardhat Tests:**
  - `Proposal acceptance threshold` block:
- **Foundry Tests:**
  - *`testQuorumStakeManipulationBetweenVotingAndSnapshot`*: Highlights that changing total stake after voting can shift quorum requirements, exposing a vulnerability in stake snapshot logic.
  - *`testQuorumCalculationWithZeroOrSmallTotalStake`*: Reveals that low-stake conditions let proposals pass without meaningful participation.

Upgradeable Contracts

Risks Associated With Core Contract Upgrades:
While extremely unlikely, there remains a non-zero risk that if a majority of validators collude, they could compromise the contract through malicious upgrades, potentially stealing governance funds or destroying the contract entirely.

The contract supports proposals that can perform contract upgrades via a `ProxyAdmin`. However, if the `ProxyAdmin` ownership is never transferred to the DAO contract, governance-driven upgrades are impossible without external intervention. This introduces a centralization point contrary to the intended decentralized model.

- **Hardhat Tests:**
  - `DAO proposal execution` block:
    * *"should perform DAO self upgrade"*
- **Foundry Tests:**

  - *testDaoCannotBeUpgradedBecauseProxyAdminIsCentralized*: Demonstrates that if ProxyAdmin isn't owned by the DAO, upgrade proposals cannot execute, contradicting the intended decentralized upgrade process.

Proper Implementation of OnlyGovernance Access Control:
Both setCreateProposalFee and setIsCoreContract functions can be executed using a proposal to change some DAO's parameters. The usage of the modifier OnlyGovernance allows only calls that go through validated proposals to execute these functions. However, some functions are guarded by onlyOwner rather than a governance-based modifier. Since the owner is never initialized, the DAO must rely on an upgrade to enable actual governance over these parameters.

- **Hardhat Tests:**
  - self function calls block:
    * *"should update createProposalFee using proposal"*
    * *"should update createProposalFee and refund original fee to proposers"*
- **Foundry Tests:**
  - *testOwnerNotInitialized*: Confirms that onlyOwner functions remain unreachable, preventing certain governance actions.

DAO Phases

Smooth Phase Transitions Without Data Inconsistencies:
The switchPhase function moves the DAO between Proposal and Voting phases at fixed intervals. In normal conditions, these transitions are smooth, and actions restricted to each phase (like creating proposals or casting votes) are appropriately enforced. However, if switchPhase is delayed and called long after the expected phase end, the subsequent phase might be shorter than intended.

- **Hardhat Tests:**
  - switchPhase block:
    * *"should not switch DAO phase before its end"*
    * *"should switch DAO phase and emit event"*
- **Foundry Tests:**
  - *testSwitchPhaseDurationReducedDueToDelay*: Highlights how delaying switchPhase calls can shorten subsequent phases.

Prevention of Stale or Dangling Proposal States:
The code ensures that proposals must be finalized before proceeding to new proposals, reducing the chance of stale states persisting indefinitely. While a proposal can be finalized well after its Voting phase has ended, the contract's enforcement of finalization before new proposals ensures that no permanently dangling proposals impede progress.

- **Hardhat Tests:**
  - `propose` block:
    * *"should revert propose if there are unfinalized proposals in previous phases"*
- **Foundry Tests:**
  - *`testMissedFinalizeCanStillBeFinalizedDuringNextPhase`*: Demonstrates that finalization can happen much later, though it at least ensures no permanent stale states linger indefinitely.

Fund Management

Secure Handling of governancePot and reinsertPot:
All proposal fee collections, refunds, and payouts are strictly tied to the proposal lifecycle. Accepted proposals trigger refunds to the proposer and, if relevant, distributions from the governancePot. Declined proposals route proposal fees to the reinsertPot, ensuring funds are properly accounted for. By segregating funds, the code prevents arbitrary fund manipulation.

- **Hardhat Tests:**
  - `funds transfer from governance pot` block:
    * *"should transfer funds from governance pot and confirm Open proposalType"*
- **Foundry Tests:**
  - *`testExecuteOpenProposal`*: Validates open proposals executing and distributing funds properly.

Prevention of Unauthorized Transfers or Misuse of DAO Funds:
Proposals must pass all checks before execution, and reentrancy guards prevent malicious call patterns. The governance funds will always be deducted from the storage variable `governancePot` after transfer. The contract never arbitrarily releases funds, adhering strictly to proposal execution logic.

- **Hardhat Tests:**
  - `funds transfer from governance pot` block:

    * *“should revert funding with insufficient governance pot balance”*
    * *“should transfer funds from governance pot and confirm Open proposalType”*
  - `reentrancy protection` block:
    * *“should revert reentrant calls to execute”*

External Interactions

Validation of Calls to External Contracts (e.g., stakingHbbft, validatorSet):
The DAO relies on external contracts for staking and validator info. The code ensures these integrations are correct, validating only active validators can vote and that staking data is read and applied properly.

Potential External Call Vulnerabilities in `_executeOperations`:
The `_executeOperations` function allows executing arbitrary calls defined by proposals. The code uses measures like verifying call results and reentrancy guards to limit vulnerabilities. The DAO collectively choses the proposals to execute and thus the risk of executing a malicious call is very low.