DMD Diamond Core

SMART CONTRACT AUDIT

21.11.2025

Made in Germany by Softstack.io

Table of contents

1. Disclaimer

The audit makes no statements or warrantees about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of DMD DIAMOND ASSOCIATION. If you are not the intended receptor of this document, remember that any disclosure, copying or dissemination of it is forbidden.

Major Versions / Date	Description
0.1 (19.12.2024)	Layout
0.4 (20.12.2024)	Automated Security Testing Manual Security Testing
0.5 (26.12.2024)	Verify Claims
0.9 (09.01.2025)	Summary and Recommendation
1.0 (10.01.2025)	Submission of findings
1.1 (07.10.2025)	Re-check
1.2 (21.11.2025)	Final document

2. About the Project and Company

Company address:

DMD DIAMOND ASSOCIATION
Krachelberg 15
8301 Laßnitzhöhe
Austria

Website: https://bit.diamonds

Twitter (X): https://twitter.com/dmdcoin

Telegram: https://t.me/DMDcoin

GitHub: https://github.com/DMDcoin/Diamond

Facebook: https://www.facebook.com/dmdcoin

2.1 Project Overview

DMD Diamond is a fully launched, enterprise-grade Proof-of-Stake blockchain platform designed for high performance, decentralization, and security. Built on the HBBFT-POSDAO consensus mechanism, it delivers instant transaction finality through leaderless Byzantine Fault Tolerance and energy-efficient validation. DMD Diamond combines the robustness of Honeybadger BFT with a Decentralized Autonomous Organization (DAO) model, enabling validators and delegators to govern the network transparently through on-chain proposals and votes.

The network features a sophisticated staking system, supporting both validators and delegators, with dynamic pool selection and epoch-based automated reward cycles. With validator management and consensus-critical operations executed gas-free, the infrastructure is optimized for real-world usage. Governance is active on-chain, allowing participants to propose and vote on network upgrades, ecosystem changes, and treasury spending, all enforced within strict cryptographic parameter bounds.

Token migration from legacy DMD v3 is live through a secure, trustless process using ECDSA signature verification and Bitcoin-compatible address formats. This process includes a dilution-based incentive for early adoption, ensuring sustainable network economics.

The mainnet is operational with real validator pools, delegated staking rewards, treasury governance, and protocol upgrade proposals. Participants can operate nodes, stake or delegate tokens, vote on proposals, and claim legacy tokens directly on-chain. Future updates will focus on expanding the ecosystem with cross-chain bridges, Layer 2 scaling, confidential transactions, native DeFi tools, and specialized infrastructure for IoT and physical network use cases. DMD Diamond is now live and positioned as a secure, scalable foundation for decentralized innovation.

3. Vulnerability & Risk Level

Risk represents the probability that a certain source-threat will exploit vulnerability, and the impact of that event on the organization or system. Risk Level is computed based on CVSS version 3.0.

Level	Value	Vulnerability	Risk (Required Action)
Critical	9 – 10	A vulnerability that can disrupt the contract functioning in a number of scenarios, or creates a risk that the contract may be broken.	Immediate action to reduce risk level.
High	7 – 8.9	A vulnerability that affects the desired outcome when using a contract, or provides the opportunity to use a contract in an unintended way.	Implementation of corrective actions as soon as possible.
Medium	4 – 6.9	A vulnerability that could affect the desired outcome of executing the contract in a specific scenario.	Implementation of corrective actions in a certain period.
Low	2 – 3.9	A vulnerability that does not have a significant impact on possible scenarios for the use of the contract and is probably subjective.	Implementation of certain corrective actions or accepting the risk.
Informational	0 – 1.9	A vulnerability that have informational character but is not effecting any of the code.	An observation that does not determine a level of risk

4. Auditing Strategy and Techniques Applied

Throughout the review process, care was taken to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices. To do so, reviewed line-by-line by our team of expert auditors and smart contract developers, documenting any issues as there were discovered.

4.1 Methodology

The auditing process follows a routine series of steps:

1. Code review that includes the following:
   i. Review of the specifications, sources, and instructions provided to softstack to make sure we understand the size, scope, and functionality of the smart contract.
   ii. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
   iii. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to softstack describe.
2. Testing and automated analysis that includes the following:
   i. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
   ii. Symbolic execution, which is analysing a program to determine what inputs causes each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarify, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, actionable recommendations to help you take steps to secure your smart contracts.

5. Metrics

The metrics section should give the reader an overview on the size, quality, flows and capabilities of the codebase, without the knowledge to understand the actual code.

5.1 Tested Contract Files

The following are the MD5 hashes of the reviewed files. A file with a different MD5 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different MD5 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review.

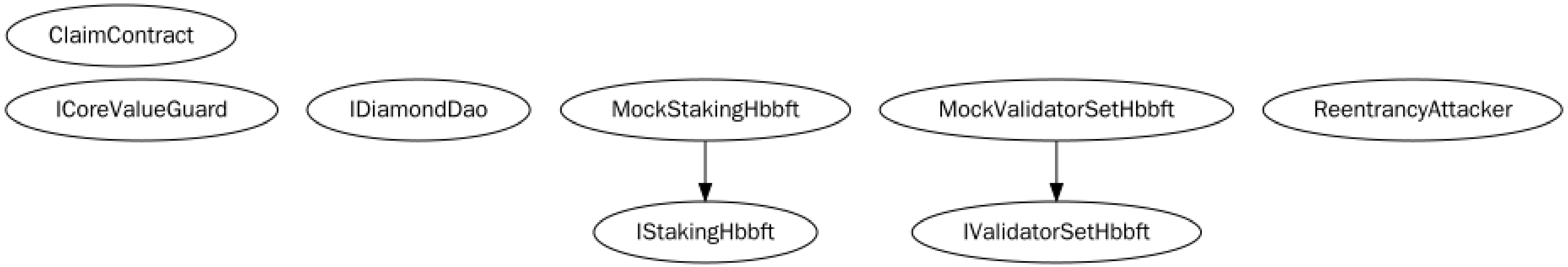
Source:
https://github.com/DMDcoin/diamond-contracts-core/commit/a121c2f26403134960095bc60a661e18cc3a0a5c
https://github.com/DMDcoin/diamond-contracts-claiming/commit/f6fc20fb8f3d5c7f2a6415d50f4cb49c7eec8ab6
https://github.com/DMDcoin/diamond-contracts-dao/commit/14c3154715bb0be223f1b0747706f875e2d27ea8

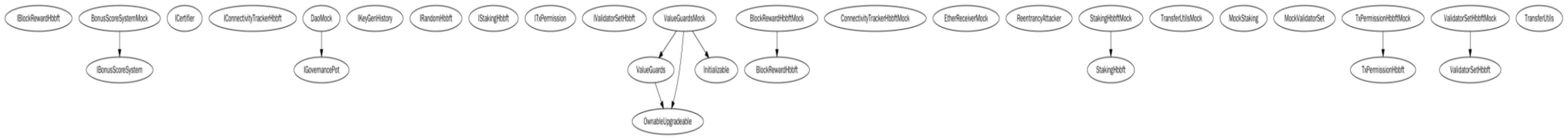
File	Fingerprint (MD5)
./contracts/BlockRewardHbbft.sol	67c702866867b406111022a0928d9da7
./contracts/BonusScoreSystem.sol	59cf66c602453617c143526e6be3ea90
./contracts/CertifierHbbft.sol	bc82bcd679f3529eb76c0c508fe8d4ab
./contracts/ConnectivityTrackerHbbft.sol	0fa2c40a112abafa87eb8c723dbae8ed
./contracts/KeyGenHistory.sol	9345c9ab4eb5daeb3848a02ba2257c43
./contracts/RandomHbbft.sol	6f73bf2356ed28084a5e6726c9082680
./contracts/StakingHbbft.sol	856841d6856e2f38d6191556b96a395d

./contracts/TxPermissionHbbft.sol	ce5660f938d2367d0abac1569e5a59eb
./contracts/ValidatorSetHbbft.sol	e70eb68fe22b15f012d7a157bddbb219
./contracts/interfaces/IBlockRewardHbbft.sol	d23724f84e8350aeadf5fb73db8faf01
./contracts/interfaces/IBonusScoreSystem.sol	cba38ffc2a2905e22362a26f1bdefafe
./contracts/interfaces/ICertifier.sol	d86c38587f798dd29a55c450644a6488
./contracts/interfaces/IConnectivityTrackerHbbft.sol	8a399022dc53172e9aa08892043434e7
./contracts/interfaces/IGovernancePot.sol	6fee261872f088e1c1c16f9d593d9bd5
./contracts/interfaces/IKeyGenHistory.sol	1d0508a7b6809460c23c9efc4c305e49
./contracts/interfaces/IRandomHbbft.sol	afe68de0816c80268b047c43a14f536a
./contracts/interfaces/IStakingHbbft.sol	d9add2b33464bb816220de3b05b5401b
./contracts/interfaces/ITxPermission.sol	e2ee44eab61bcf76ebc368bd99f6ed46
./contracts/interfaces/IValidatorSetHbbft.sol	795830ba494e96cec25f4297f73918eb
./contracts/lib/Constants.sol	a596f19276a6a5367dc3d76c0b940f2a
./contracts/lib/Errors.sol	5ffb2082af3c636e53e2bf2cae8f27ad
./contracts/lib/ValueGuards.sol	790e4d6edc2080792d84d39c41e9148f
./contracts/upgradeability/Import.sol	84d1bfe209b9def77ba521cf2501e5d7
./contracts/utils/TransferUtils.sol	02e23fea3fa802c2461852984615d3d4
./contracts/DiamondDao.sol	fb11c5030e14e0ea5a0e857dd6e50d24
./contracts/import/Import.sol	e133994fffa8c633ca8ceeaf95caa97a

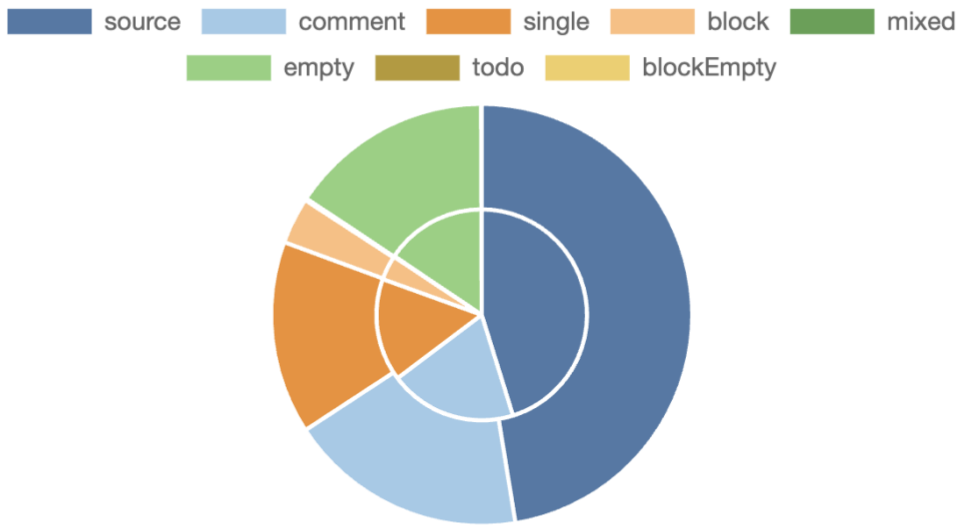
./contracts/interfaces/ICoreValueGuard.sol	149733b47c4cab546c4170e5949d7e8d
./contracts/interfaces/IDiamondDao.sol	0bf6bbade8cb870ea6396a86ee1ba9cb
./contracts/interfaces/IStakingHbbft.sol	899d9151a3da360b4d8f6ca252910cd2
./contracts/interfaces/IValidatorSetHbbft.sol	b7d73414e6361090179a1a365da31e25
./contracts/library/DaoStructs.sol	35f142ec3bf020fda9ca4f25ab8363e7

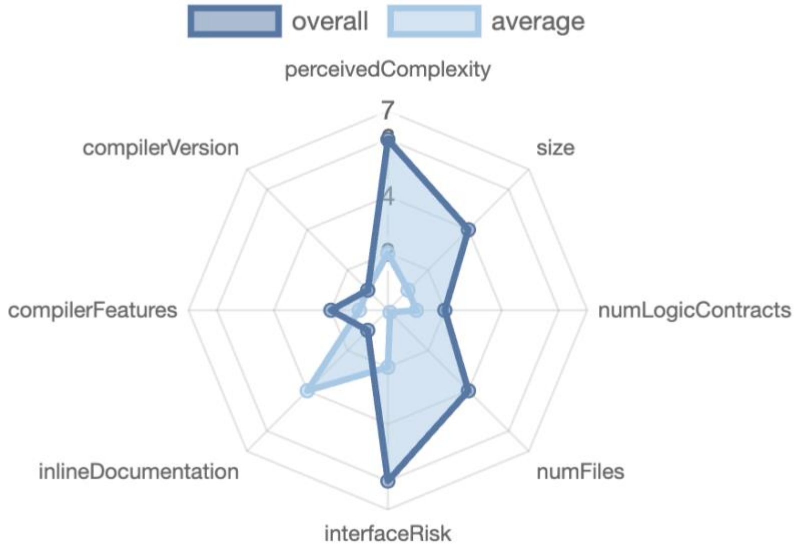
5.2 Inheritance Graph





5.3 Source Lines & Risk





5.4 Capabilities

🌐Public	💰Payable
293	12

External	Internal	Private	Pure	View
239	200	55	29	170

Total	🌐Public
172	129

Solidity Versions observed	🧪 Experimental Features	💰 Can Receive Funds	🖥 Uses Assembly	💣 Has Destroyable Contracts
=0.8.25 >=0.8.1 <0.9.0		yes	yes (4 asm blocks)	

📤 Transfers ETH	⚡ Low-Level Calls	👥 DelegateCall	🧮 Uses Hash Functions	🔖 ECRecover	🌀 New/Create/Create2
			yes	yes	

♻️ TryCatch	Σ Unchecked
yes	

5.5 Dependencies / External Imports

Dependency / Import Path	Source
@openzeppelin/contracts/utils/structs/EnumerableSet.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v5.0.2/contracts/utils/structs/EnumerableSet.sol
@openzeppelin/contracts/utils/structs/BitMaps.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v5.0.2/contracts/utils/structs/BitMaps.sol
@openzeppelin/contracts/utils/math/Math.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v5.0.2/contracts/utils/math/Math.sol
@openzeppelin/contracts/utils/Address.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v5.0.2/contracts/utils/Address.sol
@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v5.0.2/contracts/proxy/ERC1967/ERC1967Proxy.sol
@openzeppelin/contracts/proxy/transparent/TransparentUpgradeableProxy.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v5.0.2/contracts/proxy/transparent/TransparentUpgradeableProxy.sol
@openzeppelin/contracts/proxy/transparent/ProxyAdmin.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v5.0.2/contracts/proxy/transparent/ProxyAdmin.sol
@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol	https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/v5.0.2/contracts/proxy/utils/Initializable.sol
@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol	https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/v5.0.2/contracts/access/OwnableUpgradeable.sol

@openzeppelin/contracts-upgradeable/utils/ReentrancyGuardUpgradeable.sol	https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/v5.0.2/contracts/utils/ReentrancyGuardUpgradeable.sol
diamond-contracts-core/contracts/lib/ValueGuards.sol	https://github.com/DMDcoin/diamond-contracts-core/blob/master/contracts/lib/ValueGuards.sol

5.6 Source Unites in Scope

File	Logic Contracts	Interfaces	Lines	nLines	nSLOC	Comment Lines
diamond-contracts-core-master/upgradeability/Import.sol			10	10	4	3
diamond-contracts-core-master/interfaces/IBlockRewardHbbft.sol		1	9	6	3	1
diamond-contracts-core-master/interfaces/IStakingHbbft.sol		1	112	23	19	2
diamond-contracts-core-master/interfaces/IConnectivityTrackerHbbft.sol		1	34	5	3	1
diamond-contracts-core-master/interfaces/IRandomHbbft.sol		1	28	5	3	1
diamond-contracts-core-master/interfaces/ITxPermission.sol		1	12	5	3	2

File	Logic Contracts	Interfaces	Lines	nLines	nSLOC	Comment Lines
diamond-contracts-core-master/interfaces/IValidatorSetHbbft.sol		1	90	25	20	2
diamond-contracts-core-master/interfaces/IGovernancePot.sol		1	6	5	3	1
diamond-contracts-core-master/interfaces/ICertifier.sol		1	6	5	3	1
diamond-contracts-core-master/interfaces/IBonusScoreSystem.sol		1	21	12	9	1
diamond-contracts-core-master/interfaces/IKeyGenHistory.sol		1	21	5	3	1
diamond-contracts-core-master/RandomHbbft.sol	1		134	134	73	34
diamond-contracts-core-master/BonusScoreSystem.sol	1		251	246	136	62
diamond-contracts-core-master/StakingHbbft.sol	1		1599	1565	862	423
diamond-contracts-core-master/lib/ValueGuards.sol	1		174	170	74	62
diamond-contracts-core-master/lib/Constants.sol			7	7	4	3
diamond-contracts-core-master/lib/Errors.sol			7	7	5	1
diamond-contracts-core-master/ConnectivityTrackerHbbft.sol	1		472	455	220	148
diamond-contracts-core-master/TxPermissionHbbft.sol	1		517	500	282	126
diamond-contracts-core-master/utils/TransferUtils.sol	1		19	19	14	2

File	Logic Contracts	Interfaces	Lines	nLines	nSLOC	Comment Lines
diamond-contracts-core-master/utils/Secp256k1Utils.sol	1		44	44	31	3
diamond-contracts-core-master/KeyGenHistory.sol	1		192	178	114	28
diamond-contracts-core-master/BlockRewardHbbft.sol	1		507	493	264	126
diamond-contracts-core-master/CertifierHbbft.sol	1		124	120	52	42
diamond-contracts-core-master/ValidatorSetHbbft.sol	1		840	812	430	248
diamond-contracts-dao-main/interfaces/IDiamondDaoLowMajority.sol		1	18	7	4	1
diamond-contracts-dao-main/interfaces/IStakingHbbft.sol		1	8	5	3	1
diamond-contracts-dao-main/interfaces/IDiamondDao.sol		1	98	68	53	1
diamond-contracts-dao-main/interfaces/IValidatorSetHbbft.sol		1	12	5	3	1
diamond-contracts-dao-main/interfaces/ICoreValueGuard.sol		1	12	10	7	1
diamond-contracts-dao-main/import/Import.sol			4	4	2	1
diamond-contracts-dao-main/DiamondDaoLowMajority.sol	1		89	81	57	4
diamond-contracts-dao-main/library/QuorumCalculator.sol	1		28	22	12	7

File	Logic Contracts	Interfaces	Lines	nLines	nSLOC	Comment Lines
diamond-contracts-dao-main/library/DaoStructs.sol			76	76	65	1
diamond-contracts-dao-main/library/Errors.sol			5	5	3	1
diamond-contracts-dao-main/DiamondDao.sol	1		749	692	484	56
diamond-contracts-claiming-main/ClaimContract.sol	1		547	509	247	156
Totals	**16**	**15**	**6882**	**6340**	**3574**	**1555**

Legend:

- **Lines**: total lines of the source unit
- **nLines**: normalized lines of the source unit (e.g. normalizes functions spanning multiple lines)
- **nSLOC**: normalized source lines of code (only source-code lines; no comments, no blank lines)
- **Comment Lines**: lines containing single or block comments

6. Scope of Work

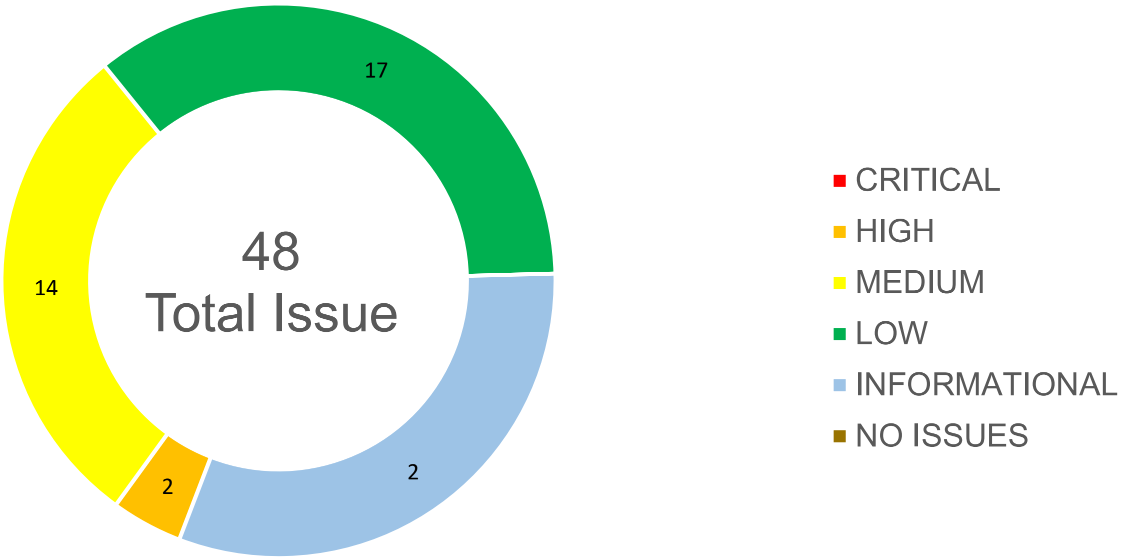
The DMD Diamonds Team provided us with the files that needs to be tested. The scope of the audit is the core, claiming and DAO contracts.

The team has outlined the following critical security and functionality assumptions for the DMD Diamond contracts:

1. **Epoch and Validator Rotation Logic**
   The audit must ensure safe and consistent epoch transitions across validator sets. Logic governing early epoch termination and disconnected validator detection should function reliably under edge conditions. Malicious or inactive validators must be penalized without risk of slashing honest participants or causing liveness faults.
2. **Reward Allocation and Pot Distribution**
   The reward distribution mechanism must correctly allocate minted rewards to the designated smart contract pots (deltaPot, reinsertPot, governancePot). Validation includes overflow protection, access restrictions for withdrawal, and behavior across epochs with varying validator activity or node connectivity.
3. **Bonus Score Incentivization Model**
   The bonus score system should fairly reflect validator performance without allowing manipulation or unintended inflation. The audit evaluates score calculations, weight impacts on reward scaling, and resistance against Sybil strategies or downtime masking.
4. **Staking and Pool Reward Safety**
   All staking operations (deposit, withdrawal, delegation) must be validated for correctness and security. Epoch-bound logic must prevent reentrancy, race conditions, or data loss during stake transitions. Pool-level reward distribution should reflect accurate stake proportions and prevent gaming through timing or fragmentation.
5. **Governance Controls and Role Permissions**
   Governance contracts must enforce strict access control over upgrade paths, treasury allocations, and protocol parameters. Only authorized roles (e.g., governance owner or DAO actors) may perform critical operations, and proposals must be validated with quorum enforcement and stake snapshot mechanisms.

The primary goal of this audit is to validate these assumptions and confirm the system's resilience, correctness, and upgrade readiness. Upon request, the audit team will also advise on performance bottlenecks, DAO treasury mechanisms, and potential L2/bridge interactions for future protocol extensions.

6.1 Findings Overview



No	Title	Severity	Status
6.2.1	Denial of Service (DoS) Attack on ClaimContract's Fill Function	HIGH	FIXED
6.2.2	Missing Access Control in AtomicSequencerImplementation	HIGH	ACKNOWLEDGED
6.2.3	Over-permissive Certification Logic in 'certified' Function	MEDIUM	ACKNOWLEDGED
6.2.4	Uninitialized Counters in KeyGenHistory Contract	MEDIUM	FIXED
6.2.5	Inconsistent Health Tracking in RandomHbbft Contract	MEDIUM	FIXED
6.2.6	Array Out of Bounds Vulnerability in getSliceUInt256 Function	MEDIUM	FIXED

6.2.7	Staking and Withdrawal Time Restrictions Disabled	MEDIUM	FIXED
6.2.8	Missing Input Validation in setPoolInfo Function	MEDIUM	FIXED
6.2.9	Potential Signature Malleability and Replay Attack Vulnerability in Claim Verification	MEDIUM	ACKNOWLEDGED
6.2.10	Incomplete Bitcoin-Style Address Generation in publicKeyToDMDAddress Function	MEDIUM	ACKNOWLEDGED
6.2.11	Insufficient Timestamp Validation in Dilution Events	MEDIUM	FIXED
6.2.12	Lack of Public Key Validation in claimMessageMatchesSignature Function	MEDIUM	FIXED
6.2.13	ECDSA Signature Malleability in claimMessageMatchesSignature Function	MEDIUM	FIXED
6.2.14	Phase Duration Will Be Shortened Each Time switchPhase Is Delayed	MEDIUM	FIXED
6.2.15	Finalization Uses Vote Snapshot But Dynamically Recalculates Quorum with Current Total Stake	MEDIUM	FIXED
6.2.16	Proposal Type Determination Ignores Intermediate Calldatas	MEDIUM	FIXED
6.2.17	Misspelled Parameter Name in Claim Function	LOW	FIXED
6.2.18	Inaccurate Comment for Public Key Y Coordinate	LOW	FIXED
6.2.19	Typographical Error in Dilution Error Message	LOW	FIXED
6.2.20	Mismatch Between Documented and Actual Return Type in createClaimMessage Function	LOW	FIXED
6.2.21	Use of RIPEMD-160 Precompile in publicKeyToDMDAddress Function May Not Be Universally Compatible	LOW	ACKNOWLEDGED
6.2.22	Misspelling in Variable Name for Reinsert Pot Transfer Amount	LOW	FIXED
6.2.23	Potential Reentrancy Vulnerability in _distributeRewards	LOW	FIXED
6.2.24	Redundant Check in 'certified' Function	LOW	FIXED
6.2.25	Revocation State Inconsistency After Key Generation	LOW	FIXED

6.2.26	Insufficient Validation in Pool Removal Functions	LOW	FIXED
6.2.27	Missing Reentrancy Protection in Critical Functions	LOW	FIXED
6.2.28	Lack of Bounds Checking in _getCurrentValWithSelector Function	LOW	FIXED
6.2.29	Implicit Double-Voting Prevention in DAO Voting System	LOW	FIXED
6.2.30	Owner Initialization Failure Making onlyOwner Functions Unusable	LOW	FIXED
6.2.31	Zero or Negligible Total Stake Allows Proposals to Always Pass	LOW	FIXED
6.2.32	ProxyAdmin Ownership Not Transferred to DAO, Blocking Proposal Type Upgrades	LOW	FIXED
6.2.33	Proposals Can Be Finalized at Any Time But Execution Is Restricted to Specific Window	LOW	FIXED
6.2.34	Inefficient Gas Usage Due to Non-Immutable Critical Variables	INFORMATIONAL	FIXED
6.2.35	Unit Test Gaps	INFORMATIONAL	FIXED
6.2.36	Limited Flexibility in Proposal Fee Configuration	INFORMATIONAL	FIXED
6.2.37	Potential for Proposal ID Collisions in hashProposal Function	INFORMATIONAL	ACKNOWLEDGED
6.2.38	Mismatch in Proposal Fee and Mention of Nonexistent Network Fees	INFORMATIONAL	ACKNOWLEDGED
6.2.39	Race Condition in _executeOperations Due to Limited governancePot	INFORMATIONAL	FIXED
6.2.40	Abstain Votes Are Ineffective	INFORMATIONAL	FIXED
6.2.41	Define and Use Constant Variables Instead of Using Literals	INFORMATIONAL	FIXED
6.2.42	Event Is Missing Indexed Fields	INFORMATIONAL	FIXED
6.2.43	Unused Custom Error	INFORMATIONAL	FIXED
6.2.44	Repeated Access to currentPhaseProposals.length in Loop Condition	INFORMATIONAL	FIXED

6.2.45	Hardcoded Governance Pot Address in Contract Logic	INFORMATIONAL	ACKNOWLEDGED
6.2.46	Commented Out Code in 'certified' Function	INFORMATIONAL	FIXED
6.2.47	Typo in Parameter Name '_upcommingEpoch'	INFORMATIONAL	FIXED
6.2.48	Unused '_txPermission' Parameter in Initialize Function	INFORMATIONAL	FIXED

6.2 Manual and Automated Vulnerability Test

CRITICAL ISSUES

During the audit, softstack‘s experts found **no Critical issues** in the code of the smart contract.

HIGH ISSUES

During the audit, softstack‘s experts found **Two High issues** in the code of the smart contract.

6.2.1 Denial of Service (DoS) Attack on ClaimContract's Fill Function

Severity: HIGH
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-claiming/issues/48
File(s) affected: ClaimContract.sol

Attack / Description	The ClaimContract contains a critical vulnerability in its fill function that allows a malicious actor to permanently prevent the contract from being initialized. This vulnerability stems from an incorrect balance check that can be exploited by sending a small amount of Ether to the contract before the fill function is called. The fill function contains the following check: `if (address(this).balance != msg.value) revert FillErrorBalanceDoubleFill();` This check is intended to ensure that the contract is being filled for the first time. However, it fails to account for the possibility of the contract receiving Ether through other means. **Impact:**

Denial of Service: An attacker can permanently prevent the contract from being initialized by sending a small amount of Ether (e.g., 1 wei) to the contract address. This will cause the balance check to always fail, making it impossible to call the fill function successfully.

Contract Rendered Unusable: Once this attack is executed, the entire contract becomes unusable, as the fill function is crucial for initializing the contract with the necessary balances.

Proof of Concept:

Deploy the ClaimContract.

Attacker sends 1 wei to the contract address.

Any attempt to call fill will now fail due to the balance check.

Code

Lines 175-195 (ClaimContract.sol):

```
function fill(bytes20[] memory _accounts, uint256[] memory _balances) external
payable {

    //for simplification we only support a one-shot initialisation.

    if (filled) revert FillErrorBalanceDoubleFill();

    if (msg.value == 0) revert FillErrorValueRequired();

    if (_accounts.length != _balances.length) revert
FillErrorNumberOfAccountsMissmatch();
```

```
    // we verify if the transfered amount that get added to the sum up to the total amount added.

    uint256 totalBalanceAdded = 0;


    for (uint256 i = 0; i < _accounts.length; ++i) {

        if (_accounts[i] == bytes20(address(0))) revert FillErrorAccountZero();

        if (_balances[i] == 0) revert FillErrorBalanceZero();

        if (balances[_accounts[i]] != 0) revert FillErrorAccountAlreadyDefined();

        totalBalanceAdded += _balances[i];

        balances[_accounts[i]] = _balances[i];

    }


    if (msg.value != totalBalanceAdded) revert FillErrorBalanceSumError();


    filled = true;

}
```

Result/Recommendation

Replace the current balance check with a mechanism that doesn't rely on the contract's balance. Some options include:

1. Use a boolean flag to track if the contract has been filled:

```
bool private filled;

function fill(bytes20[] memory _accounts, uint256[] memory _balances) external
payable {

    if (filled) revert FillErrorBalanceDoubleFill();

    filled = true;

    // Rest of the function...

}
```

2. Check the total balance of recorded accounts instead of the contract's balance:

```
uint256 public totalRecordedBalance;



function fill(bytes20[] memory _accounts, uint256[] memory _balances) external
payable {

    if (totalRecordedBalance > 0) revert FillErrorBalanceDoubleFill();

    // Rest of the function...
```

	`    totalRecordedBalance = msg.value;`
	
	`}`

6.2.2 Signature Replay Attack Vulnerability in ClaimContract

Severity: HIGH
Status: ACKNOWLEDGED
Github: https://github.com/DMDcoin/diamond-contracts-claiming/issues/49
File(s) affected: ClaimContract.sol

Attack / Description	The claim function in the ClaimContract is vulnerable to signature replay attacks. The function uses a _postfix parameter in the signature verification process, but there's no mechanism to ensure this postfix is unique for each claim. This oversight allows an attacker to potentially reuse a valid signature to claim funds multiple times, especially if the contract is redeployed or used on different chains. **Impact:** 1. **Multiple Claims**: An attacker could use a single valid signature to claim funds multiple times, potentially draining the contract of more funds than intended. 2. **Cross-Chain Vulnerability**: If the same contract is deployed on multiple chains, a signature used on one chain could be replayed on another. 3. **Redeployment Risk**: If the contract is ever redeployed (e.g., for upgrades), old signatures could become valid again, allowing for duplicate claims. **Proof of Concept:** 1. Alice generates a valid signature for claiming her funds, using a generic postfix. 2. Alice successfully claims her funds. 3. The contract is redeployed due to an upgrade.

	4. An attacker obtains Alice's old signature and uses it to claim funds again, as the new contract has no record of the previous claim.
Code	Lines 207-241 (ClaimContract.sol)

```
function claim(
    address payable _targetAddress,
    bytes memory _postfix,
    bytes32 _pubKeyX,
    bytes32 _pubKeyY,
    uint8 _v,
    bytes32 _r,
    bytes32 _s
) external {
    //retrieve the oldAddress out of public key.
    bytes20 oldAddress = publicKeyToDMDAddress(_pubKeyX, _pubKeyY);

    //if already claimed, it just returns.
    uint256 currentBalance = balances[oldAddress];
    if (currentBalance == 0) revert ClaimErrorNoBalance();



    // verify if the signature matches to the provided pubKey here.

    if (!claimMessageMatchesSignature(_targetAddress, _postfix, _pubKeyX, _pubKeyY,
_v, _r, _s))

        revert ClaimErrorSignatureMissmatch();



    (uint256 nominator, uint256 denominator) = getCurrentDilutedClaimFactor();

    uint256 claimBalance = (currentBalance * nominator) / denominator;



    // remember that the funds are going to get claimed, hard protection about
reentrancy attacks.

    balances[oldAddress] = 0;



    _transferNative(_targetAddress, claimBalance);



    emit Claim(oldAddress, _targetAddress, claimBalance, nominator, denominator);

}
```

Result/Recommendation

To mitigate this vulnerability, implement one or more of the following measures:

1. **Nonce System:**

```
mapping(bytes20 => uint256) public nonces;


function claim(..., uint256 nonce, ...) external {

    require(nonce == nonces[oldAddress], "Invalid nonce");

    // ... (existing claim logic)

    nonces[oldAddress]++;

}
```

2. **Single-Use Signatures:**

```
mapping(bytes32 => bool) public usedSignatures;


function claim(...) external {

    bytes32 sigHash = keccak256(abi.encodePacked(_v, _r, _s));

    require(!usedSignatures[sigHash], "Signature already used");
    usedSignatures[sigHash] = true;

    // ... (existing claim logic)

}
```

3. **Chain ID Inclusion:** Include the chain ID in the signed message to prevent cross-chain replay attacks.

MEDIUM ISSUES

During the audit, softstack's experts found **Fourteen Medium issues** in the code of the smart contract.

6.2.3 Over-permissive Certification Logic in 'certified' Function

Severity: MEDIUM
Status: ACKNOWLEDGED
Github: https://github.com/DMDcoin/diamond-contracts-core/issues/284
File(s) affected: CertifierHbbft.sol

Attack / Description	The 'certified' function in the CertifierHbbft contract contains a vulnerability in its logic for determining whether an address is certified. The function returns true for any address that has an associated staking address, without verifying if that staking address belongs to an active validator. This over-permissive validation could potentially allow unauthorized addresses to perform zero-gas transactions, compromising the intended access control of the system. This function returns true if either:

1. The address is explicitly certified (_certified[_who] is true)
2. The address has any non-zero staking address associated with it

The second condition is too permissive, as it doesn't verify if the staking address belongs to an active validator.

Impact:

- **Unauthorized Access**: Non-validator addresses with non-zero staking addresses could be incorrectly certified
- **Zero-gas Transaction Abuse**: Could allow unauthorized addresses to perform zero-gas transactions
- **Access Control Bypass**: Compromises the intended security model of the certification system

Proof of Concept:

```
it("should incorrectly certify an address with a non-zero staking address that is
not an active validator", async function () {

    const { certifier, validatorSetHbbft } = await loadFixture(deployContracts);

    const nonValidatorAddress = ethers.Wallet.createRandom().address;

    const fakeStakingAddress = ethers.Wallet.createRandom().address;



    await validatorSetHbbft.setStakingAddressTest(nonValidatorAddress,
fakeStakingAddress);
```

	`    expect(await certifier.certifiedExplicitly(nonValidatorAddress)).to.be.false;`
	`    expect(await certifier.certified(nonValidatorAddress)).to.be.true;`
	`    expect(await validatorSetHbbft.isValidator(fakeStakingAddress)).to.be.false;`
	`});`

Code	Lines 94-101 (CertifierHbbft.sol)
	`function certified(address _who) external view returns (bool) {`
	`    if (_certified[_who]) {`
	`        return true;`
	`    }`
	`    address stakingAddress = validatorSetContract.stakingByMiningAddress(_who);`
	`    return stakingAddress != address(0);`
	`}`

Result/Recommendation	1. Modify the 'certified' function to implement a multi-layered check:
	`function certified(address _who) external view returns (bool) {`
	`    if (_certified[_who]) {`

```
        return true;
    }
    address stakingAddress = validatorSetContract.stakingByMiningAddress(_who);
    if (stakingAddress == address(0)) {
        return false;
    }
    return validatorSetContract.isValidator(stakingAddress) &&
           validatorSetContract.isValidatorActive(stakingAddress) &&
           !validatorSetContract.isValidatorBanned(stakingAddress);
}
```

2. Implement a time-based recertification mechanism:
    - Add an expiration timestamp to each certification
    - Automatically revoke certifications that have expired
    - Require validators to recertify periodically
3. Implement an event-driven certification update system:
    - Emit events when validator status changes
    - Create a mechanism to revoke certifications based on these events

6.2.4 Uninitialized Counters in KeyGenHistory Contract

Severity: MEDIUM
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-core/issues/285
File(s) affected : KeyGenHistory.sol

Attack / Description

The KeyGenHistory contract has an initialization vulnerability where the counters numberOfPartsWritten and numberOfAcksWritten are not properly initialized during the contract's initialization process. This leads to an inconsistent state where these counters remain at zero even though parts and acknowledgments (acks) are stored for the initial set of validators.

The initialize function stores parts and acks for each validator but does not update the corresponding counters. This discrepancy between the actual stored data and the counter values can lead to incorrect behavior in functions that rely on these counters.

Impact:

- **State Inconsistency**: Counters don't reflect actual stored data
- **Incorrect Key Generation Progress Tracking**: System may make incorrect decisions based on inaccurate counter values
- **Validator Management Issues**: Could affect the key generation process and validator onboarding

Proof of Concept:

```
it('should demonstrate inconsistent state after initialization', async () => {

    const KeyGenFactory = await ethers.getContractFactory("KeyGenHistory");
    const contract = await upgrades.deployProxy(
        KeyGenFactory,
        [owner.address, stubAddress, initializingMiningAddresses, parts, acks],
        { initializer: 'initialize' }
    );
    const [numberOfPartsWritten, numberOfAcksWritten] = await
contract.getNumberOfKeyFragmentsWritten();
    const validatorsCount = initializingMiningAddresses.length;
    // Counters are zero
    expect(numberOfPartsWritten).to.equal(0);
    expect(numberOfAcksWritten).to.equal(0);
    // But data is actually stored
    let actualPartsCount = 0;
    let actualAcksCount = 0;
    for (let i = 0; i < validatorsCount; i++) {
        const storedPart = await contract.getPart(initializingMiningAddresses[i]);
        const storedAcksLength = await
contract.getAcksLength(initializingMiningAddresses[i]);
```

```
        if (storedPart.length > 0) actualPartsCount++;

        if (storedAcksLength > 0) actualAcksCount++;

    }

    expect(actualPartsCount).to.equal(validatorsCount);

    expect(actualAcksCount).to.equal(validatorsCount);

});
```

Code

Lines 82-117 (KeyGenHistory.sol)

```
function initialize(

    address _contractOwner,

    address _validatorSetContract,

    address[] memory _validators,

    bytes[] memory _parts,

    bytes[][] memory _acks

) external initializer {

    if (_contractOwner == address(0) || _validatorSetContract == address(0)) {

        revert ZeroAddress();

    }
```

```
if (_validators.length == 0) {
    revert ValidatorsListEmpty();
}
if (_validators.length != _parts.length) {
    revert WrongPartsNumber();
}

if (_validators.length != _acks.length) {
    revert WrongAcksNumber();
}
__Ownable_init(_contractOwner);
validatorSetContract = IValidatorSetHbbft(_validatorSetContract);
for (uint256 i = 0; i < _validators.length; i++) {
    parts[_validators[i]] = _parts[i];
    acks[_validators[i]] = _acks[i];
}
```

	`    currentKeyGenRound = 1;`
	`    numberOfPartsWritten = uint128(_validators.length);`
	`    numberOfAcksWritten = uint128(_validators.length);`
	`}`

Result/Recommendation	Modify the initialize function to properly set the counter values:
	`function initialize(`
	`    address _contractOwner,`
	`    address _validatorSetContract,`
	`    address[] memory _validators,`
	`    bytes[] memory _parts,`
	`    bytes[][] memory _acks`
	`) external initializer {`
	`    // ... existing initialization code ...`
	
	`    for (uint256 i = 0; i < _validators.length; i++) {`
	`        parts[_validators[i]] = _parts[i];`

```
        acks[_validators[i]] = _acks[i];

    }

    numberOfPartsWritten = uint128(_validators.length);

    numberOfAcksWritten = uint128(_validators.length);

    currentKeyGenRound = 1;

}
```

6.2.5 Inconsistent Health Tracking in RandomHbbft Contract

Severity: MEDIUM
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-core/issues/286
File(s) affected: RandomHbbft.sol

Attack / Description	The RandomHbbft contract has an inconsistency in how it tracks the health status of the validator set. The contract sets an unhealthiness flag for a block when setCurrentSeed is called and the validator set is not at full health. However, there is no mechanism to reset this flag when the validator set returns to full health. This can lead to inaccurate historical health data. The issue occurs in the setCurrentSeed function:

```
function setCurrentSeed(uint256 _currentSeed) external onlySystem {

    randomHistory[block.number] = _currentSeed;

    if (!validatorSetContract.isFullHealth()) {
        unhealthiness.set(block.number);

    }

}
```

Once a block is marked as unhealthy, it remains so permanently in the historical record, even if the validator set recovers in subsequent blocks.

Impact:

- **Inaccurate Historical Data**: Blocks remain marked as unhealthy even after validator set recovers
- **Misleading Health Metrics**: Cannot accurately track when the system returns to full health
- **Incorrect Decision Making**: Systems relying on health history may make suboptimal decisions

Proof of Concept:

```
it('should demonstrate inconsistent health tracking', async () => {

    const { randomHbbft, validatorSetHbbft } = await
helpers.loadFixture(deployContracts);

    const systemSigner = await impersonateSystemAcc();

    const validators = await validatorSetHbbft.getValidators();

    // Initial state: full health

    expect(await randomHbbft.isFullHealth()).to.be.true;
    // Block 1: Set to unhealthy by kicking a validator

    await validatorSetHbbft.kickValidator(validators[0]);

    await randomHbbft.connect(systemSigner).setCurrentSeed(random(0,
Number.MAX_SAFE_INTEGER));

    expect(await randomHbbft.isFullHealthHistoric(await
helpers.time.latestBlock())).to.be.false;



    // Block 2: Simulate returning to healthy state

    await randomHbbft.connect(systemSigner).setCurrentSeed(random(0,
Number.MAX_SAFE_INTEGER));

    // All blocks show unhealthy even though state may have changed

    expect(await randomHbbft.isFullHealth()).to.be.false;

    expect(await randomHbbft.isFullHealthHistoric(await helpers.time.latestBlock()
- 1)).to.be.false;

});
```

Code

Lines 81-90 (RandomHbbft.sol)

```
function setCurrentSeed(uint256 _currentSeed) external onlySystem {

    randomHistory[block.number] = _currentSeed;
    bool healthy = validatorSetContract.isFullHealth();

    if (!healthy) {

        unhealthiness.set(block.number);

    }

    emit SetCurrentSeed(block.number, _currentSeed, healthy);

}
```

Result/Recommendation

1. Modify the setCurrentSeed function to always update the health status:

```
function setCurrentSeed(uint256 _currentSeed) external onlySystem {

    randomHistory[block.number] = _currentSeed;

    if (validatorSetContract.isFullHealth()) {

        unhealthiness.unset(block.number);

    } else {

        unhealthiness.set(block.number);

    }

}
```

2. Alternatively, use two separate BitMaps to track both history and current state:

```
BitMaps.BitMap private hasBeenUnhealthy;
BitMaps.BitMap private currentHealth;



function setCurrentSeed(uint256 _currentSeed) external onlySystem {

    randomHistory[block.number] = _currentSeed;

    if (!validatorSetContract.isFullHealth()) {

        hasBeenUnhealthy.set(block.number);

        currentHealth.unset(block.number);

    } else {

        currentHealth.set(block.number);

    }

}
```

6.2.6 Array Out of Bounds Vulnerability in getSliceUInt256 Function

Severity: MEDIUM
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-core/issues/287
File(s) affected: TxPermissionHbbft.sol

Attack / Description	The getSliceUInt256 function in the TxPermissionHbbft contract contains a critical vulnerability that allows for out-of-bounds array access. This function attempts to read 32 bytes from a given offset within a byte array without properly checking the array's bounds. If the offset plus 32 exceeds the

length of the input array, it will read beyond the array's limits, potentially accessing unintended memory locations.

Impact:

- **Out-of-Bounds Memory Access**: Reading beyond array boundaries
- **Unpredictable Behavior**: Could lead to unexpected results or contract failures
- **Potential Security Risk**: Memory corruption could be exploited in certain scenarios

Proof of Concept:

```
it('should revert when accessing out of bounds', async function () {

    const { txPermission } = await helpers.loadFixture(deployContractsFixture);

    const smallData = ethers.hexlify(new Uint8Array(10)); // 10 bytes

    await expect(

        txPermission.testGetSliceUInt256(30, smallData) // 30 + 32 > 10

    ).to.be.reverted;

});
```

Code

Lines 455-466 (TxPermissionHbbft.sol)

```
function _getSliceUInt256(uint256 _begin, bytes memory _data) internal pure returns
(uint256) {
    if (_begin + 32 > _data.length) {
        revert ReadOutOfBounds();
    }

    uint256 a = 0;
    for (uint256 i = 0; i < 32; i++) {
        a = a + (((uint256)((uint8)(_data[_begin + i]))) * ((uint256)(2 ** ((31 -
i) * 8))));
    }
    return a;
}
```

Result/Recommendation

Implement proper bounds checking before accessing the array elements:

```
function _getSliceUInt256(uint256 _begin, bytes memory _data) internal pure returns
(uint256) {

    require(_begin + 32 <= _data.length, "Reading out of bounds");



    uint256 a = 0;

    for (uint256 i = 0; i < 32; i++) {

        a = a + (((uint256)((uint8)(_data[_begin + i]))) * ((uint256)(2 ** ((31 -
i) * 8))));

    }

    return a;

}
```

6.2.7 Staking and Withdrawal Time Restrictions Disabled

Severity: MEDIUM
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-core/issues/288
File(s) affected: StakingHbbft.sol

Attack / Description	The areStakeAndWithdrawAllowed() function in the StakingHbbft contract is currently hardcoded to always return true, effectively disabling any time-based restrictions on staking and withdrawal operations. This deviates from the original design described in the POSDAO Whitepaper, which specified a specific window within each staking epoch for these operations. The commented-out code suggests that there should be time-based restrictions, but these are not currently implemented. **Impact:** Allows Unrestricted Staking/Withdrawals: Permits operations at any time during the epoch Removes Protection Mechanism: Could allow rapid stake movements to manipulate validator selection Deviates from Design: Does not match the behavior described in the whitepaper Potential for Gaming: Users could exploit timing to their advantage
Result/Recommendation	1. Re-evaluate the need for time-based restrictions on staking and withdrawals. If they are deemed necessary: o Implement the time-based restrictions as originally designed o Update the areStakeAndWithdrawAllowed() function to properly check the current time against the allowed window 2. If time-based restrictions are not needed: o Update the contract documentation and whitepaper to reflect this change

	○ Consider implementing alternative mechanisms to prevent potential abuse, such as a delay between unstaking and the ability to withdraw funds

6.2.8 Missing Input Validation in setPoolInfo Function

Severity: MEDIUM
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-core/issues/289
File(s) affected: StakingHbbft.sol

| **Attack / Description** | The setPoolInfo function in the StakingHbbft contract lacks crucial input validation for its parameters. This function allows users to set their pool's public key, IP address, and port number. However, it does not perform any checks on the validity or format of these inputs. The specific issues are:
1. **PublicKey Validation**: No check on the length or format of the publicKey parameter
2. **IP Address Validation**: The ip parameter is not validated for proper IPv4 format
3. **Port Number Validation**: No check to ensure port is within valid range (0-65535)
Impact:
• **Storage of Invalid Data**: Nonsensical or malformed data can be stored
• **System Vulnerabilities**: Other parts assuming valid data could malfunction
• **Difficulty in Data Interpretation**: Invalid data makes it hard to use pool information correctly
• **Potential DOS Vectors**: Invalid data could cause issues in dependent functions
Proof of Concept: |

```
it('should accept any public key length', async () => {
    const shortPublicKey = ethers.zeroPadBytes("0xdead", 4);
    const longPublicKey = ethers.zeroPadBytes("0xbeef", 128);
    const ipAddress = ethers.zeroPadBytes("0xfe", 16);
    const port = '0x6987';
    await expect(stakingHbbft.connect(validator).setPoolInfo(shortPublicKey,
ipAddress, port))
        .to.not.be.reverted;
    await expect(stakingHbbft.connect(validator).setPoolInfo(longPublicKey,
ipAddress, port))
        .to.not.be.reverted;
});
it('should accept invalid IP addresses', async () => {
    const publicKey = ethers.zeroPadBytes("0xdeadbeef", 64);
    const invalidIpAddress = ethers.zeroPadBytes("0xffffffff", 16);
    const port = '0x6987';
    await expect(stakingHbbft.connect(validator).setPoolInfo(publicKey,
invalidIpAddress, port))
        .to.not.be.reverted;
});
```

Code

Lines 562-566 (StakingHbbft.sol)

```
function setPoolInfo(bytes calldata _publicKey, bytes16 _ip, bytes2 _port) external
{
    poolInfo[msg.sender].publicKey = _publicKey;
    poolInfo[msg.sender].internetAddress = _ip;
    poolInfo[msg.sender].port = _port;
}
```

Result/Recommendation

1. Implement strict input validation for all parameters:

```
function setPoolInfo(bytes calldata _publicKey, bytes16 _ip, bytes2 _port) external
{

    require(_publicKey.length == EXPECTED_PUBLIC_KEY_LENGTH, "Invalid public key
length");

    require(isValidIPv4(_ip), "Invalid IP address");



    uint16 portNumber = uint16(bytes2(_port));

    require(portNumber <= 65535, "Invalid port number");



    poolInfo[msg.sender].publicKey = _publicKey;

    poolInfo[msg.sender].internetAddress = _ip;

    poolInfo[msg.sender].port = _port;
```

	`}`
	`function isValidIPv4(bytes16 _ip) internal pure returns (bool) {`
	`    // Implement IPv4 validation logic here`
	`    return true;`
	`}`
	2. If possible, implement a mechanism to verify the ownership of the IP address and control over the port
	3. Consider adding events to log when pool info is updated

6.2.9 Potential Signature Malleability and Replay Attack Vulnerability in Claim Verification

Severity: MEDIUM
Status: ACKNOWLEDGED
Github: https://github.com/DMDcoin/diamond-contracts-claiming/issues/50
File(s) affected: ClaimContract.sol

Attack / Description	The claimMessageMatchesSignature function in the ClaimContract uses the Solidity ecrecover function directly to verify signatures. This implementation is vulnerable to signature malleability, which could potentially lead to replay attacks or other security issues. **Impact:** 1. **Signature Malleability**: The ecrecover EVM opcode allows for malleable (non-unique) signatures. For every valid signature (r,s,v), there exists another valid signature (r,-s mod N,v') for the same message and signer.

2. **Potential for Replay Attacks**: While the current implementation may not be directly vulnerable to replay attacks due to other safeguards, the use of malleable signatures is considered a security risk.
3. **Inconsistent Signature Verification**: Malicious actors could potentially create multiple valid signatures for the same message.

Proof of Concept:

1. A valid signature (r, s, v) is created for a claim.
2. An attacker could create a new signature (r, -s mod N, v') which would also be valid for the same claim.
3. Both signatures would pass the verification in claimMessageMatchesSignature.

Code

Lines 367-396 (ClaimContract.sol)

```
function claimMessageMatchesSignature(
    address _claimToAddr,
    bytes memory _postFix,
    bytes32 _pubKeyX,
    bytes32 _pubKeyY,
    uint8 _v,
    bytes32 _r,
    bytes32 _s
) public view returns (bool) {
    if (_v < 27 || _v > 30) revert CryptoInvalidV();

    if (!isValidPublicKey(_pubKeyX, _pubKeyY)) {
        revert InvalidPublicKey();
    }

    if (uint256(_s) > SECP256K1_N / 2) revert SignatureMalleabilityError();

    /*
```

```
      ecrecover() returns an Eth address rather than a public key, so
      we must do the same to compare.
    */
    address pubKeyEthAddr = pubKeyToEthAddress(_pubKeyX, _pubKeyY);

    //we need to check if X and Y corresponds to R and S.

    /* Create and hash the claim message text */
    bytes32 messageHash = getHashForClaimMessage(_claimToAddr, _postFix);

    /* Verify the public key */
    return ecrecover(messageHash, _v, _r, _s) == pubKeyEthAddr;
}
```

Result/Recommendation

1. Use OpenZeppelin's ECDSA library for signature verification:

```
import "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";



function claimMessageMatchesSignature(

    address _claimToAddr,

    bytes memory _postFix,

    bytes32 _pubKeyX,

    bytes32 _pubKeyY,

    uint8 _v,
    bytes32 _r,

    bytes32 _s

) public view returns (bool) {

    bytes32 messageHash = getHashForClaimMessage(_claimToAddr, _postFix);

    address signer = ECDSA.recover(messageHash, _v, _r, _s);

    return signer != address(0) && signer == pubKeyToEthAddress(_pubKeyX,
_pubKeyY);

}
```

2. Implement additional checks to ensure signature components are within valid ranges:

```
require(_s <= 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0,
"Invalid signature 's' value");
```

3. Consider implementing a nonce system for each claim.

6.2.10 Incomplete Bitcoin-Style Address Generation in publicKeyToDMDAddress Function

Severity: MEDIUM
Status: ACKNOWLEDGED
Github: https://github.com/DMDcoin/diamond-contracts-claiming/issues/51
File(s) affected: ClaimContract.sol

Attack / Description	The publicKeyToDMDAddress function in the ClaimContract attempts to generate a Bitcoin-style address from ECDSA public key coordinates. However, the implementation is incomplete as it

omits the crucial step of Base58Check encoding, which is standard for Bitcoin addresses. Instead of returning a properly formatted Bitcoin address string, the function returns a raw 20-byte value.

Impact:

1. **Incompatibility**: The generated addresses are not compatible with standard Bitcoin address formats
2. **Misinterpretation**: Users or external systems may mishandle the raw byte output
3. **Additional Processing Required**: Systems need to implement Base58Check encoding separately
4. **Potential Loss of Funds**: If addresses are used directly without proper encoding

Proof of Concept:

1. Generate a valid ECDSA key pair.
2. Call publicKeyToDMDAddress with the public key coordinates.
3. The resulting 20-byte output will not be a valid Bitcoin address and cannot be used directly in Bitcoin-compatible systems.

Code

Lines 429-441 (ClaimContract.sol)

```
function publicKeyToDMDAddress(
    bytes32 _publicKeyX,
    bytes32 _publicKeyY
) public pure returns (bytes20 rawBtcAddress) {
    return
        ripemd160(
            abi.encodePacked(
                sha256(
                    abi.encodePacked((uint256(_publicKeyY) & 1) == 0 ? 0x02 : 0x03,
_publicKeyX)
                )
            )
```

	`        );`
	`}`

Result/Recommendation	Implement the full Bitcoin address generation process, including Base58Check encoding:
	`function publicKeyToDMDAddress(bytes32 _publicKeyX, bytes32 _publicKeyY)`
	`    public pure returns (string memory) {`
	`    bytes memory compressedPubKey = compressPublicKey(_publicKeyX, _publicKeyY);`
	`    bytes20 pubKeyHash = ripemd160(abi.encodePacked(sha256(compressedPubKey)));`
	`    bytes21 versionedPayload = abi.encodePacked(byte(0x00), pubKeyHash);`
	`    bytes32 checksum = sha256(abi.encodePacked(sha256(versionedPayload)));`
	`    bytes25 addressBytes = abi.encodePacked(versionedPayload, checksum[0:4]);`
	`    return toBase58Check(addressBytes);`
	`}`
	Note: This corrected implementation requires additional helper functions like compressPublicKey and toBase58Check.

6.2.11 Insufficient Timestamp Validation in Dilution Events

Severity: MEDIUM
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-claiming/issues/52
File(s) affected: ClaimContract.sol

Attack / Description

The ClaimContract contains two significant issues related to the validation of dilution event timestamps:

1. **Lack of Minimum Time Gap**: While the contract ensures that each dilution timestamp is greater than the previous one, it does not enforce a minimum time gap between dilution events. This could allow dilution events to occur in rapid succession.
2. **Missing Upper Bound Check**: The contract does not implement an upper bound check for dilution timestamps. This allows for setting extremely large timestamp values.

Impact:

1. **Rapid Dilution**: Contract owner could set dilution timestamps very close to each other, diluting user balances more quickly than intended
2. **Permanent Token Lock**: Setting extremely high timestamp values could effectively lock tokens indefinitely
3. **Potential Overflow**: Large timestamp values might cause overflow issues

Proof of Concept:

```
// Rapid dilution scenario

ClaimContract contract = new ClaimContract(

    beneficorAddress1,

    beneficorAddress2,

    "prefix",

    1000000001, // 1 second after deployment
    1000000002, // 2 seconds after deployment

    1000000003  // 3 seconds after deployment

);



// Permanent lock scenario

ClaimContract contract = new ClaimContract(

    beneficorAddress1,

    beneficorAddress2,

    "prefix",

    type(uint256).max, // max uint256 value

    type(uint256).max,

    type(uint256).max

);
```

Code

Lines 142-169 (ClaimContract.sol)

```
constructor(
    address payable _lateClaimBeneficorAddressReinsertPot,
    address payable _lateClaimBeneficorAddressDAO,
    bytes memory _prefixStr,
    uint256 _dilute_s1_75_timestamp,
    uint256 _dilute_s2_50_timestamp,
    uint256 _dilute_s3_0_timestamp
) {
    if (_lateClaimBeneficorAddressReinsertPot == address(0))
        revert InitializationErrorReinsertPotAddressNull();
    if (_lateClaimBeneficorAddressDAO == address(0)) revert
InitializationErrorDaoAddressNull();
    if (_dilute_s1_75_timestamp <= block.timestamp)
        revert InitializationErrorDiluteTimestamp1();
    if (_dilute_s2_50_timestamp <= _dilute_s1_75_timestamp)
        revert InitializationErrorDiluteTimestamp2();
    if (_dilute_s3_0_timestamp <= _dilute_s2_50_timestamp)
        revert InitializationErrorDiluteTimestamp3();

    lateClaimBeneficorAddressReinsertPot = _lateClaimBeneficorAddressReinsertPot;
    lateClaimBeneficorAddressDAO = _lateClaimBeneficorAddressDAO;

    prefixStr = _prefixStr;

    dilute_s1_75_timestamp = _dilute_s1_75_timestamp;
    dilute_s2_50_timestamp = _dilute_s2_50_timestamp;
    dilute_s3_0_timestamp = _dilute_s3_0_timestamp;
    filled = false;
}
```

Result/Recommendation

1. Enforce minimum time gaps:

```
uint256 constant MIN_DILUTION_GAP = 30 days;



constructor(...) {
    if (_dilute_s2_50_timestamp <= _dilute_s1_75_timestamp + MIN_DILUTION_GAP)
        revert InsufficientDilutionGap();
    if (_dilute_s3_0_timestamp <= _dilute_s2_50_timestamp + MIN_DILUTION_GAP)
        revert InsufficientDilutionGap();
    // ...
}
```

2. Add a reasonable upper limit:

```
uint256 constant MAX_TIMESTAMP = 2524608000; // January 1, 2050


constructor(...) {
    if (_dilute_s1_75_timestamp > MAX_TIMESTAMP) revert TimestampTooHigh();
    if (_dilute_s2_50_timestamp > MAX_TIMESTAMP) revert TimestampTooHigh();
    if (_dilute_s3_0_timestamp > MAX_TIMESTAMP) revert TimestampTooHigh();
    // ...
}
```

6.2.12 Lack of Public Key Validation in claimMessageMatchesSignature Function

Severity: MEDIUM
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-claiming/issues/53
File(s) affected: ClaimContract.sol

Attack / Description	The claimMessageMatchesSignature function in the ClaimContract lacks crucial validation for the ECDSA public key coordinates (_pubKeyX and _pubKeyY) provided as input parameters. This omission could potentially allow the processing of invalid or maliciously crafted public keys. **Impact:** 1. **Security Vulnerability**: Contract may process claims using invalid public keys 2. **Potential for Attacks**: Malicious actors could exploit invalid public keys that pass other checks 3. **Integrity Compromise**: Absence of validation compromises the claiming mechanism 4. **Potential for Signature Forgery**: In extreme cases, could allow forged signatures **Proof of Concept:** An attacker could potentially pass invalid public key coordinates that don't lie on the secp256k1 curve: 1. Generate an invalid public key pair (X, Y) that doesn't lie on the secp256k1 curve 2. Use this invalid key pair in a claim transaction 3. The contract would process this claim without detecting the invalid key

Code	Lines 331-342 (ClaimContract.sol)
	`function isValidPublicKey(bytes32 _pubKeyX, bytes32 _pubKeyY) public pure returns (bool) {`
	`    uint256 p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F;`

```
    uint256 x = uint256(_pubKeyX);
    uint256 y = uint256(_pubKeyY);
    if (x == 0 || x >= p || y == 0 || y >= p) {
        return false;
    }
    // Check if the point is on the curve: y^2 = x^3 + 7 (mod p)
    uint256 lhs = mulmod(y, y, p);
    uint256 rhs = addmod(mulmod(mulmod(x, x, p), x, p), 7, p);
    return lhs == rhs;
}
```

Result/Recommendation

1. Implement a validation function for the public key coordinates:

```
error InvalidPublicKey();



function isValidPublicKey(bytes32 _pubKeyX, bytes32 _pubKeyY) internal pure returns
(bool) {

    uint256 p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F;

    uint256 x = uint256(_pubKeyX);

    uint256 y = uint256(_pubKeyY);



    if (x == 0 || x >= p || y == 0 || y >= p) {

        return false;

    }
    // Check if the point is on the curve: y^2 = x^3 + 7 (mod p)

    uint256 lhs = mulmod(y, y, p);

    uint256 rhs = addmod(mulmod(mulmod(x, x, p), x, p), 7, p);

    return lhs == rhs;

}
```

2. Modify the claimMessageMatchesSignature function:

```
function claimMessageMatchesSignature(...) public view returns (bool) {

    if (!isValidPublicKey(_pubKeyX, _pubKeyY)) {

        revert InvalidPublicKey();

    }

    // Rest of the function...

}
```

6.2.13 ECDSA Signature Malleability in claimMessageMatchesSignature Function

Severity: MEDIUM
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-core/issues/286
File(s) affected: ClaimContract.sol

<table>
<tr>
<td>

Attack / Description

</td>
<td>

The claimMessageMatchesSignature function in the ClaimContract is vulnerable to signature malleability attacks. The function verifies ECDSA signatures without implementing protection against the malleability property of ECDSA signatures. In ECDSA, for every valid signature (r, s), the signature (r, -s mod n) is also valid for the same message and public key.

Impact:

1. **Transaction Replay**: An attacker could potentially modify a valid signature to create a new valid signature for the same message
2. **Double Spending**: In systems where signatures authorize transfers, signature malleability could be exploited for double spending
3. **Contract State Manipulation**: Could lead to unexpected state changes or unauthorized actions

Proof of Concept:

1. A valid signature (r, s) is created for a claim.
2. An attacker could create a new signature (r, -s mod n) which would also be valid for the same claim.
3. Both signatures would pass the verification in claimMessageMatchesSignature.

</td>
</tr>
<tr>
<td>

Code

</td>
<td>

Lines 367-396 (ClaimContract.sol) - specifically line 384

```
function claimMessageMatchesSignature(
    address _claimToAddr,
    bytes memory _postFix,
    bytes32 _pubKeyX,
    bytes32 _pubKeyY,
    uint8 _v,
```

</td>
</tr>
</table>

```
    bytes32 _r,
    bytes32 _s
) public view returns (bool) {
    if (_v < 27 || _v > 30) revert CryptoInvalidV();

    if (!isValidPublicKey(_pubKeyX, _pubKeyY)) {
        revert InvalidPublicKey();
    }

    if (uint256(_s) > SECP256K1_N / 2) revert SignatureMalleabilityError();
    // ^^ This line (384) protects against malleability

    address pubKeyEthAddr = pubKeyToEthAddress(_pubKeyX, _pubKeyY);
    bytes32 messageHash = getHashForClaimMessage(_claimToAddr, _postFix);
    return ecrecover(messageHash, _v, _r, _s) == pubKeyEthAddr;
}
```

Result/Recommendation

Implement additional checks to ensure only one of the two possible signatures is considered valid:

1. Add a constant for the secp256k1 curve order:

```
uint256 private constant SECP256K1_N =

    0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141;
```

2. Modify the claimMessageMatchesSignature function:

```
function claimMessageMatchesSignature(...) public view returns (bool) {

    if (_v < 27 || _v > 30) revert CryptoInvalidV();
    // Ensure s is in the lower half of its range
    if (uint256(_s) > SECP256K1_N / 2) revert SignatureMalleabilityError();


    address pubKeyEthAddr = pubKeyToEthAddress(_pubKeyX, _pubKeyY);
    bytes32 messageHash = getHashForClaimMessage(_claimToAddr, _postFix);
    return ecrecover(messageHash, _v, _r, _s) == pubKeyEthAddr;
}
```

3. Add a new error:

```
error SignatureMalleabilityError();
```

6.2.14 Phase Duration Will Be Shortened Each Time switchPhase Is Delayed

Severity: MEDIUM
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-dao/issues/48
File(s) affected: DiamondDao.sol

Attack / 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 last 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

Proof of Concept:

```
function testSwitchPhaseDurationReducedDueToDelay() public {

    uint64 DAO_PHASE_DURATION = dao.DAO_PHASE_DURATION();

    (, uint64 currentEnd,,) = dao.daoPhase();


    // Simulate delay: Advance time by 1 hour after phase end

    uint64 timeAfterEnd = currentEnd + 1 hours;
    vm.warp(timeAfterEnd);


    dao.switchPhase();

    (uint64 newStart, uint64 newEnd,,) = dao.daoPhase();


    uint64 remainingDuration = newEnd > uint64(block.timestamp)

        ? newEnd - uint64(block.timestamp) : 0;

    // Remaining duration is less than DAO_PHASE_DURATION

    assertTrue(remainingDuration < DAO_PHASE_DURATION);

}
```

Code

Lines 219-231 (DiamondDao.sol)

```
function switchPhase() external nonReentrant {
    uint64 currentTimestamp = uint64(block.timestamp);

    if (currentTimestamp < daoPhase.end) {
        return;
    }

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

    daoPhase.start = currentTimestamp;
    daoPhase.end = currentTimestamp + DAO_PHASE_DURATION;
    daoPhase.phase = newPhase;
```

Result/Recommendation

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;

}
```

6.2.15 Finalization Uses Vote Snapshot But Dynamically Recalculates Quorum with Current Total Stake

Severity: MEDIUM
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-dao/issues/49
File(s) affected: DiamondDao.sol

Attack / 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;

    }



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

}
```

This creates a discrepancy: even though votes are locked-in at the Voting phase end, the effective quorum requirement can shift due to total stake amount changes happening after 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
- 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 encourages strategic stake manipulation.

Proof of Concept:

```
function testQuorumStakeManipulationBetweenVotingAndSnapshot() public {

    // Set initial stake

    mockStaking.setStake(voter, 10 ether);

    assertEq(mockStaking.totalStakedAmount(), 10 ether);



    // Create and vote on proposal

    uint256 proposalId = createProposal(/*...*/);

    switchPhase(); // to Voting

    vm.prank(voter);

    dao.vote(proposalId, Vote.Yes);

    switchPhase(); // end Voting
    // Verify proposal is passing

    VotingResult memory res = dao.countVotes(proposalId);

    uint256 requiredExceeding = mockStaking.totalStakedAmount() * (33 * 100) / 10000;

    assertGt(res.stakeYes, res.stakeNo + requiredExceeding);



    // Another user massively increases stake

    mockStaking.setStake(users[4], 1000 ether);



    // Now proposal fails

    result = dao.countVotes(proposalId);

    requiredExceeding = mockStaking.totalStakedAmount() * (33 * 100) / 10000;

    assertLt(result.stakeYes, result.stakeNo + requiredExceeding);

}
```

Code	Lines 542-562 (DiamondDao.sol)

```
function quorumReached(
    uint256 proposalId,
    ProposalType _type,
    VotingResult memory result
) public view returns (bool) {
    uint256 totalVotes = _proposalVoters[proposalId].length();

    uint256 daoEpoch = proposals[proposalId].votingDaoEpoch;
    uint256 totalStakedAmount = daoEpochTotalStakeSnapshot[daoEpoch];
    bool isQuorumReached;

    if (_type == ProposalType.ContractUpgrade || _type ==
ProposalType.OpenHighMajority) {
        isQuorumReached = result.highMajorityQuorum(totalStakedAmount);
    } else if (_type == ProposalType.OpenLowMajority) {
        isQuorumReached = lowMajorityDao.quorumReached(result, totalStakedAmount);
    } else {
        isQuorumReached = result.lowMajorityQuorum(totalStakedAmount);
    }

    return totalVotes > 0 && isQuorumReached;
}
```

Result/Recommendation	The total stake amount must be snapshotted in _snapshotStakes during the phase switch. This will ensure that the quorum calculation is fair and coherent. Follow OpenZeppelin's implementation.

6.2.16 Proposal Type Determination Ignores Intermediate Calldatas

Severity: MEDIUM
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-dao/issues/50
File(s) affected: DiamondDao.sol

<table>
<tr>
<td>

Attack / Description

</td>
<td>

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;
```

</td>
</tr>
</table>

```
    }

} else {

    return ProposalType.ContractUpgrade;

}
```

This creates issues 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 the type of the whole proposal is set as ContractUpgrade or EcosystemParameterChange, the execValue variable will be set to 0, preventing value transfer.

Impact:

- A proposal with multiple calldatas including empty calldatas with corresponding positive values will not transfer governance funds during execution
- Arbitrary ecosystem parameters can be set for functions that do not enforce the isWithinAllowedRange checks directly in their logic
- The proposal will not be executed as expected

Code

Lines 704-747 (DiamondDao.sol)

```
function _checkProposalType(
    address[] memory targets,
    bytes[] memory calldatas
) private view returns (ProposalType _type) {
    _type = ProposalType.OpenLowMajority;

    for (uint256 i = 0; i < calldatas.length; i++) {
        if (calldatas[i].length == 0) continue;
        (bytes4 setFuncSelector, uint256 newVal) = _extractCallData(calldatas[i]);

        (bool success, bytes memory result) = targets[i].staticcall(
            abi.encodeWithSelector(

ICoreValueGuard(targets[i]).getAllowedParamsRangeWithSelector.selector,
                setFuncSelector
            )
        );

        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;
        }
    }
```

	`}`
Result/Recommendation	• Each calldata should have a corresponding ProposalType - 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 ISSUES

During the audit, softstack‘s experts found **Seventeen Low issues** in the code of the smart contract

6.2.17 Misspelled Parameter Name in Claim Function

Severity: LOW
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-claiming/issues/56
File(s) affected: ClaimContract.sol

Attack / Description	In the claim function of the ClaimContract, there is a misspelling in the parameter name _targetAdress. The correct spelling should be _targetAddress. This inconsistency in naming could lead to confusion for developers working with or maintaining the contract. **Impact:** • **Code Readability**: Minor impact on code readability and maintainability • **Developer Confusion**: Could confuse developers integrating with the contract

Code	Lines 207-216 (ClaimContract.sol)
	`function claim(`
	`    address payable _targetAddress,`
	`    bytes memory _postfix,`

	`    bytes32 _pubKeyX,`
	`    bytes32 _pubKeyY,`
	`    uint8 _v,`
	`    bytes32 _r,`
	`    bytes32 _s`
	`) external {`
	`    //retrieve the oldAddress out of public key.`

Result/Recommendation	Correct the spelling of the parameter name throughout the function:
	`function claim(`
	`    address payable _targetAddress,  // Corrected spelling`
	`    // ... other parameters`
	`) external {`
	`    // Update all references within the function body`
	`    _transferNative(_targetAddress, claimBalance);`
	`    emit Claim(oldAddress, _targetAddress, claimBalance, nominator, denominator);`
	`}`

6.2.18 Inaccurate Comment for Public Key Y Coordinate

Severity: LOW
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-claiming/issues/57

File(s) affected: ClaimContract.sol

Attack / Description	In the claim function of the ClaimContract, there is an incorrect comment for the _pubKeyY parameter. The comment incorrectly states "ECDSA public key X coordinate" instead of "Y coordinate". **Impact:** - **Documentation Error**: Misleading documentation for developers - **No Functional Impact**: Does not affect contract execution

Code	Lines 200-207 (ClaimContract.sol)
	`/// @param _postfix an optional string postfix that can be added to the message.`
	`/// Useful to work around the limitation that only 32 byte R and S values can be processed.`
	`/// @param _pubKeyX ECDSA public key X coordinate`
	`/// @param _pubKeyY ECDSA public key Y coordinate`
	`/// @param _v ECDSA V`
	`/// @param _r ECDSA R (32 byte)`
	`/// @param _s ECDSA S (32 byte)`
	`function claim(`

Result/Recommendation	Correct the comment for the _pubKeyY parameter:
	`/// @param _pubKeyX ECDSA public key X coordinate`
	`/// @param _pubKeyY ECDSA public key Y coordinate  // Corrected comment`
	`function claim(`
	`    // ... other parameters`

```
    bytes32 _pubKeyX,

    bytes32 _pubKeyY,

    // ... remaining parameters

) external {

    // Function body

}
```

6.2.19 Typographical Error in Dilution Error Message

Severity: LOW
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-claiming/issues/58
File(s) affected: ClaimContract.sol

Attack / Description	In the ClaimContract, specifically within the dilution functions (dilute1, dilute2, and dilute3), there is a typographical error in the error message used when reverting due to an already executed dilution. The error message "DiluteAllreadyHappened" is misspelled, with an extra 'l' in "Allready". **Impact:** • **Code Quality**: Minor impact on code quality and professionalism • **Error Message Clarity**: Could confuse users reading error messages

Code	Lines 61-62 (ClaimContract.sol)
	`/// @notice dilute event did already happen.`
	`error DiluteAlreadyHappened();`
	`And usage in Lines 243-245:`
	`if (dilution_s1_75_executed) {`
	`    revert DiluteAlreadyHappened();`
	`}`

Result/Recommendation	1. Correct the spelling of the error message throughout the contract:
	`error DiluteAlreadyHappened();`
	2. Update all references to this error in the contract:
	`if (dilution_s1_75_executed) revert DiluteAlreadyHappened();`
	`if (dilution_s2_50_executed) revert DiluteAlreadyHappened();`
	`if (dilution_s3_0_executed) revert DiluteAlreadyHappened();`

6.2.20 Mismatch Between Documented and Actual Return Type in createClaimMessage Function

Severity: LOW
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-claiming/issues/59
File(s) affected: ClaimContract.sol

<table>
<tr>
<td><strong>Attack / Description</strong></td>
<td>

In the ClaimContract, the createClaimMessage function has a discrepancy between its documented return type and its actual return type. The function comment states that it returns a bytes32, but the function signature and implementation actually return bytes memory.

Impact:

- **Documentation Inconsistency**: Could confuse developers using or maintaining the contract
- **No Functional Impact**: Does not affect contract execution

</td>
</tr>
<tr>
<td><strong>Code</strong></td>
<td>

Lines 300-313 (ClaimContract.sol)

```
/**
 * @notice returns the claim message for the provided claim target address.
 * @param _claimToAddr address target address for the claim.
 * @return bytes memory Encoded claim message.
 */
function createClaimMessage(
    address _claimToAddr,
    bytes memory _postfix
) public view returns (bytes memory) {
    //TODO: pass this as an argument. evaluate in JS before includeAddrChecksum is
used or not.
    //now for testing, we assume Yes.

    bytes memory addrStr = calculateAddressString(_claimToAddr);
```

</td>
</tr>
<tr>
<td><strong>Result/Recommendation</strong></td>
<td>

Update the function comment to accurately reflect the actual return type:

```
/**

 * @notice returns the claim message for the provided claim target address.
```

</td>
</tr>
</table>

```
 * @param _claimToAddr address target address for the claim.

 * @return bytes memory Encoded claim message.

 */

function createClaimMessage(

    address _claimToAddr,

    bytes memory _postfix

) public view returns (bytes memory) {

    // Function implementation

}
```

6.2.21 Use of RIPEMD-160 Precompile in publicKeyToDMDAddress Function May Not Be Universally Compatible

Severity: LOW
Status: ACKNOWLEDGED
Github: https://github.com/DMDcoin/diamond-contracts-claiming/issues/60
File(s) affected: ClaimContract.sol

Attack / Description	The publicKeyToDMDAddress function in the ClaimContract uses the ripemd160 hash function, which is implemented as a precompile contract in Ethereum at address(2). This function is used to generate a Bitcoin-style address from ECDSA public key coordinates. However, the availability and behavior of this precompile may not be consistent across all Ethereum-compatible chains or Layer 2 solutions.

	Impact: If deployed on a chain where the RIPEMD-160 precompile is not available or functions differently: 1. **Execution Failure**: Transactions may revert 2. **Incorrect Addresses**: Could produce incorrect Bitcoin-style addresses 3. **Increased Gas Costs**: If precompile is implemented differently

Code	Lines 429-441 (ClaimContract.sol)
	`function publicKeyToDMDAddress(`
	`    bytes32 _publicKeyX,`
	`    bytes32 _publicKeyY`
	`) public pure returns (bytes20 rawBtcAddress) {`
	`    return`
	`        ripemd160(`
	`            abi.encodePacked(`
	`                sha256(`
	`                    abi.encodePacked((uint256(_publicKeyY) & 1) == 0 ? 0x02 : 0x03, _publicKeyX)`
	`                )`
	`            )`
	`        );`
	`}`

Result/Recommendation	1. Verify the availability and correctness of the RIPEMD-160 precompile on all target chains before deployment 2. Consider implementing a fallback pure Solidity version of RIPEMD-160 3. Add a configuration option to switch between precompile and pure Solidity implementation 4. Document clearly which chains have been verified to support the required precompiles 5. If possible, consider using alternative hash functions that are more universally supported Example of a potential fallback approach:

```
function publicKeyToDMDAddress(bytes32 _publicKeyX, bytes32 _publicKeyY)
    public view returns (bytes20 rawBtcAddress) {
    bytes memory input = abi.encodePacked(
        sha256(abi.encodePacked((uint256(_publicKeyY) & 1) == 0 ? 0x02 : 0x03,
_publicKeyX))
    );
    if (isPrecompileAvailable(address(2))) {
        return ripemd160(input);
    } else {
        return fallbackRIPEMD160(input);
    }
}
function isPrecompileAvailable(address precompile) internal view returns (bool) {
    uint256 codeSize;
    assembly {
        codeSize := extcodesize(precompile)
    }
```

	`    return codeSize > 0;`
	
	`}`

6.2.22 Misspelling in Variable Name for Reinsert Pot Transfer Amount

Severity: LOW
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-claiming/issues/61
File(s) affected: ClaimContract.sol

Attack / Description	In the _sendDilutedAmounts function of the ClaimContract, there is a typographical error in the variable name used for calculating the transfer amount to the Reinsert Pot. The variable is named transferForResinsertPot instead of the correct spelling transferForReinsertPot. **Impact:** - **Code Quality**: Minor impact on code quality - **No Functional Impact**: Does not affect contract execution

Code	Lines 498-505 (ClaimContract.sol)
	`function _sendDilutedAmounts(uint256 amount) internal {`
	`    //diluted amounts are split 50/50 to DAO and ReinsertPot.`
	`    uint256 transferForReinsertPot = amount / 2;`
	`    uint256 transferForDAO = amount - transferForReinsertPot;`
	
	`    _transferNative(lateClaimBeneficorAddressReinsertPot, transferForReinsertPot);`
	`    _transferNative(lateClaimBeneficorAddressDAO, transferForDAO);`
	`}`

Result/Recommendation	Correct the spelling of the variable name throughout the function:
	`function _sendDilutedAmounts(uint256 amount) internal {`
	`    //diluted amounts are split 50/50 to DAO and ReinsertPot.`
	`    uint256 transferForReinsertPot = amount / 2;  // Corrected spelling`
	`    uint256 transferForDAO = amount - transferForReinsertPot;`
	`    _transferNative(lateClaimBeneficorAddressReinsertPot, transferForReinsertPot);`
	`    _transferNative(lateClaimBeneficorAddressDAO, transferForDAO);`
	`}`

6.2.23 Potential Reentrancy Vulnerability in _distributeRewards

Severity: LOW
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-core/issues/290
File(s) affected: BlockRewardHbbft.sol

Attack / Description	The _distributeRewards function in the BlockRewardHbbft contract performs external calls before completing all state updates. Although it includes a reentrancy guard and follows the Checks-Effects-Interactions pattern, there is still a potential vulnerability related to state updates after the external call to _mintNativeCoins.

The vulnerable pattern is:

_mintNativeCoins(payable(reinsertPot), mintedForReinsertPot);

potPayoutAmount[0] = mintedForReinsertPot;

State is updated (potPayoutAmount[0]) after an external call, which could be exploited if the reinsert pot contract is malicious or compromised.

Impact:

- Potential reentrancy attacks if _mintNativeCoins allows callback execution
- Temporary state inconsistency between minting and state recording
- Risk of double accounting if the function is reentered

Proof of Concept: Attack scenario:

1. Malicious reinsert pot contract receives minted tokens via _mintNativeCoins
2. Callback triggers a reentrant call to _distributeRewards
3. Since potPayoutAmount[0] hasn't been updated yet, second call operates on stale state
4. Potential double distribution or accounting mismatch

Code

Lines 307-315 (BlockRewardHbbft.sol)

```
stakingContract.restake{ value: poolReward }(poolStakingAddress,
minValidatorRewardPercent);
                }
            }

            nativeRewardUndistributed = shares.totalRewards - distributedAmount;
```

	TransferUtils.transferNative(governancePotAddress, shares.governancePotAmount); // slither-disable-end reentrancy-eth

Result/Recommendation	1. Update all state variables before making external calls:
	// Update state first
	potPayoutAmount[0] = mintedForReinsertPot;
	// Then make external call
	_mintNativeCoins(payable(reinsertPot), mintedForReinsertPot);
	2. Add explicit reentrancy checks at function entry
	3. Consider implementing ReentrancyGuard pattern from OpenZeppelin
	4. Document all external call points and their reentrancy implications

6.2.24 Redundant Check in 'certified' Function

Severity: LOW
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-core/issues/291
File(s) affected: CertifierHbbft.sol

Attack / Description	The certified function in the CertifierHbbft contract performs a redundant check against the validator set. The function checks if the address is in the pending validators list but also retrieves

and checks the miningByStakingAddress mapping, which is unnecessary given that getValidators already returns the comprehensive list.

The redundant code:

```
address miningAddress = IValidatorSetHbbft(contractAddress)
    .miningByStakingAddress(_who);
if (miningAddress != address(0)) {
    return miningAddress;
}
```

This check is redundant because the pending validators check already covers all active validators.

Impact:

- Increased gas costs due to unnecessary external calls
- Added complexity and reduced code maintainability
- Potential confusion about which check is authoritative
- No security impact but affects efficiency

Code

Lines 94-101 (CertifierHbbft.sol)

```
function certified(address _who) external view returns (bool) {
    if (_certified[_who]) {
        return true;
    }

    address stakingAddress = validatorSetContract.stakingByMiningAddress(_who);
    return stakingAddress != address(0);
}
```

Result/Recommendation

Remove the redundant check and simplify the function logic:

```
function certified(address _who) external view returns (address) {
    address contractAddress = address(validatorSetContract);
    if (contractAddress == address(0)) {
        return address(0);
    }

    address[] memory validators =
IValidatorSetHbbft(contractAddress).getValidators();
    for (uint256 i = 0; i < validators.length; i++) {
        if (validators[i] == _who) {
            return _who;
        }
    }
    return address(0);
}
```

6.2.25 Revocation State Inconsistency After Key Generation

Severity: LOW
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-core/issues/292
File(s) affected: KeyGenHistory.sol

Attack / Description	In the KeyGenHistory contract, when the clearPrevKeyGenState function is called at the start of a new key generation round, it clears all pending validators' states and revocations from the previous epoch. However, it doesn't properly handle validators whose isPendingValidator status has changed between epochs.
	The issue occurs in this sequence:
	address[] memory validators = IValidatorSetHbbft(contractAddress).getPendingValidators();
	for (uint256 i = 0; i < validators.length; i++) {
	    // Clear states for current pending validators only
	    delete parts[validators[i]][_prevKeyGenRound];
	    delete acks[validators[i]][_prevKeyGenRound];
	}
	If a validator was pending in the previous round but is no longer pending in the current round, their key generation data from the previous round persists.

Impact:

- Stale validator data remains in the contract
- Increased storage costs from orphaned data
- Potential confusion when querying historical key generation data
- May interfere with proper epoch transitions

Code

Lines 121-129 (KeyGenHistory.sol)

```
function clearPrevKeyGenState(address[] calldata _prevValidators) external
onlyValidatorSet {
    for (uint256 i = 0; i < _prevValidators.length; ++i) {
        delete parts[_prevValidators[i]];
        delete acks[_prevValidators[i]];
    }

    numberOfPartsWritten = 0;
    numberOfAcksWritten = 0;
}
```

Result/Recommendation

1. Store validators list at the start of each key generation round
2. Use the stored list for cleanup:

```
// At start of key generation

validatorsForRound[round] = getPendingValidators();



// During cleanup

for (uint256 i = 0; i < validatorsForRound[_prevKeyGenRound].length; i++) {
    address validator = validatorsForRound[_prevKeyGenRound][i];

    delete parts[validator][_prevKeyGenRound];

    delete acks[validator][_prevKeyGenRound];

}
```

6.2.26 Insufficient Validation in Pool Removal Functions

Severity: LOW
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-core/issues/293
File(s) affected: ValidatorSetHbbft.sol

Attack / Description	The ValidatorSetHbbft contract's pool removal functions (_removePool and _removePoolUnordered) lack comprehensive validation checks before removing a pool from the active validator set. The functions assume the pool exists and is in a removable state without verifying critical preconditions. Missing validations: 1. No check if pool has pending stakes that should be processed first 2. No validation of pool's current epoch participation 3. No check if pool is currently part of consensus 4. No verification of dependent state in related contracts **Impact:** • Premature pool removal could orphan stakes

<table>
<tr><td></td><td>

- Potential loss of staking rewards for pool participants
- Disruption to consensus if active validator is removed
- State inconsistency across interconnected contracts

</td></tr>
<tr><td><b>Result/Recommendation</b></td><td>Add comprehensive validation before pool removal:
<pre>
function _removePool(address _stakingAddress) internal {

    require(_stakingAddress != address(0), "Invalid address");

    require(isPoolActive(_stakingAddress), "Pool not active");

    require(!isPending(_stakingAddress), "Cannot remove pending validator");

    require(stakeAmount[_stakingAddress] == 0, "Pool has active stakes");

    require(!isValidatorOnline(_stakingAddress), "Validator currently online");

    // Proceed with removal

    _removePoolUnordered(_poolIndex(_stakingAddress));

}
</pre></td></tr>
</table>

6.2.27 Missing Reentrancy Protection in Critical Functions

Severity: LOW
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-dao/issues/51
File(s) affected: DiamondDao.sol

<table>
<tr>
<td><strong>Attack / Description</strong></td>
<td>

The DiamondDao contract initializes the ReentrancyGuard in its initialize function but fails to apply the nonReentrant modifier to several critical functions that handle funds or make significant state changes. This omission could potentially leave these functions vulnerable to reentrancy attacks.

Functions missing the nonReentrant modifier:

1. propose: Handles incoming funds (proposal fee) and updates contract state
2. finalize: Transfers funds and updates critical contract state
3. _transfer: A private function that handles fund transfers
4. switchPhase: Makes significant state changes to the DAO's phase and proposals

Impact: Potential reentrancy vulnerabilities in functions that handle funds or make critical state changes.

</td>
</tr>
<tr>
<td><strong>Result/Recommendation</strong></td>
<td>

Add the nonReentrant modifier to the following functions:

```
function propose(...) external payable nonReentrant

    onlyPhase(Phase.Proposal) noUnfinalizedProposals { ... }



function finalize(uint256 proposalId) external nonReentrant

    exists(proposalId) { ... }



function switchPhase() external nonReentrant { ... }
```

For the private _transfer function, ensure it's only called within functions that have the nonReentrant modifier.

</td>
</tr>
</table>

6.2.28 Lack of Bounds Checking in _getCurrentValWithSelector Function

Severity: LOW
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-dao/issues/52
File(s) affected: DiamondDao.sol

Attack / Description	The _getCurrentValWithSelector function in the DiamondDao contract performs a staticcall to an external contract and attempts to read a 32-byte value from the returned data without verifying the data length. This can lead to reading beyond the bounds of the returned data array, potentially causing unexpected behaviour or security vulnerabilities. **Impact:** 1. Reading uninitialized or garbage data if the returned data is less than 32 bytes 2. Potential exposure of sensitive information from adjacent memory 3. Possible runtime errors or unexpected contract behaviour The function assumes that the staticcall always returns at least 32 bytes of data, which may not be true for all external function calls.

Result/Recommendation	Add a bounds check and use safer decoding:
	`function _getCurrentValWithSelector(address contractAddress, string memory funcSelector)`
	`    private view returns(uint256) {`
	`    bytes memory selectorBytes = abi.encodeWithSelector(bytes4(keccak256(bytes(funcSelector))));`

```
    (bool success, bytes memory data) = contractAddress.staticcall(selectorBytes);

    if (!success) revert ContractCallFailed(selectorBytes, contractAddress);

    require(data.length >= 32, "Insufficient return data");


    uint256 result;

    assembly {

        result := mload(add(data, 0x20))

    }

    return result;

}
```

6.2.29 Implicit Double-Voting Prevention in DAO Voting System

Severity: LOW
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-dao/issues/53
File(s) affected: DiamondDao.sol

Attack / Description	The DiamondDao contract implements a voting system for proposals using an EnumerableSet to track voters and a mapping to store vote records. While the current implementation implicitly

prevents double-voting by overwriting previous votes, it lacks explicit checks and clear feedback mechanisms to handle attempted double-voting scenarios.

Impact: Lack of explicit double-voting prevention and user feedback.

Result/Recommendation

1. Implement an explicit check for existing votes:

```
function _submitVote(address voter, uint256 proposalId, Vote _vote, string memory
reason) private {

    _requireState(proposalId, ProposalState.Active);

    require(!_proposalVoters[proposalId].contains(voter), "Already voted");

    require(_proposalVoters[proposalId].add(voter), "Failed to record voter");


    votes[proposalId][voter] = VoteRecord({

        timestamp: uint64(block.timestamp),

        vote: _vote,

        reason: reason

    });

}
```

2. Consider adding a separate function for vote changes if that's an intended feature

6.2.30 Owner Initialization Failure Making onlyOwner Functions Unusable

Severity: LOW
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-dao/issues/54
File(s) affected: DiamondDao.sol

Attack / 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.
Result/Recommendation	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

6.2.31 Zero or Negligible Total Stake Allows Proposals to Always Pass

Severity: LOW
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-dao/issues/55
File(s) affected: DiamondDao.sol

Attack / 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.

```
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).

<table>
<tr>
<td></td>
<td>

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

</td>
</tr>
<tr>
<td>

Code

</td>
<td>

Lines 542-562 (DiamondDao.sol)

```
function quorumReached(
    uint256 proposalId
) public view exists(proposalId) returns (bool) {
    ProposalType _type = proposals[proposalId]._type;
    VotingResult memory result = results[proposalId];

    uint256 totalVotes = _proposalVoters[proposalId].length();

    uint256 daoEpoch = proposals[proposalId].votingDaoEpoch;
    uint256 totalStakedAmount = daoEpochTotalStakeSnapshot[daoEpoch];
    bool isQuorumReached;

    if (_type == ProposalType.ContractUpgrade || _type ==
ProposalType.OpenHighMajority) {
        isQuorumReached = result.highMajorityQuorum(totalStakedAmount);
    } else if (_type == ProposalType.OpenLowMajority) {
        isQuorumReached = lowMajorityDao.quorumReached(result, totalStakedAmount);
    } else {
        isQuorumReached = result.lowMajorityQuorum(totalStakedAmount);
    }

    return totalVotes > 0 && isQuorumReached;
}
```

</td>
</tr>
<tr>
<td>

Result/Recommendation

</td>
<td>

- Ensure that proposals cannot pass with 0 votes
- Consider implementing a base quorum that is independent of the total staked amount

</td>
</tr>
</table>

6.2.32 ProxyAdmin Ownership Not Transferred to DAO, Blocking Proposal Type Upgrades

Severity: LOW
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-dao/issues/56
File(s) affected: DiamondDao.sol

Attack / Description	The DAO uses a transparent proxy pattern controlled by a ProxyAdmin contract. For on-chain governance proposals to perform contract upgrades, the DAO must own the ProxyAdmin. However, the current deployment setup 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. **Impact:** This lack of proper ownership assignment prevents the DAO from freely executing upgrade proposals.
Result/Recommendation	During deployment or initialization, explicitly transfer the ProxyAdmin ownership to the DAO contract.

6.2.33 Proposals Can Be Finalized at Any Time But Execution Is Restricted to Specific Window

Severity: LOW
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-dao/issues/57
File(s) affected: DiamondDao.sol

Attack / Description	According to the DiamondDAO documentation, historical proposals can be finalized and executed at any time. However, execution of a finalized proposal is only possible within a certain window:

	`function _requireIsExecutable(uint256 _proposalId) private view {`
	
	`    Proposal memory proposal = getProposal(_proposalId);`
	
	`    if (proposal.daoPhaseCount + 1 != daoPhaseCount) {`
	
	`        revert OutsideExecutionWindow(_proposalId);`
	
	`    }`
	
	`}`
	
	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.

Code	Lines 660-665 (DiamondDao.sol)
	`function _requireIsExecutable(uint256 _proposalId) private view {`
	`    Proposal memory proposal = getProposal(_proposalId);`
	
	`    if (proposal.daoPhaseCount + 1 != daoPhaseCount) {`
	`        revert OutsideExecutionWindow(_proposalId);`
	`    }`

Result/Recommendation	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 ISSUES

During the audit, softstack‘s experts found **Fifteen Informational issues** in the code of the smart contract

6.2.34 Inefficient Gas Usage Due to Non-Immutable Critical Variables

Severity: INFORMATIONAL
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-claiming/issues/62
File(s) affected: ClaimContract.sol

Attack / Description	The ClaimContract contains several critical state variables that are set in the constructor and are not intended to be changed after contract deployment. These variables include beneficiary addresses and dilution timestamps. However, these variables are not declared as immutable, leading to unnecessary gas costs for reading these values.
	Impact:
	1. **Gas Inefficiency**: Every read operation on these variables consumes more gas than necessary
	2. **Cumulative Cost**: In a frequently accessed contract, this can lead to significantly higher cumulative gas costs for users
	3. **Potential Security Risk**: Without the immutable keyword, there's a theoretical risk that these critical values could be changed if the contract were to include functions to modify them in the future
	Proof of Concept: Consider the following snippet from the constructor:
	`constructor(`
	`    address payable _lateClaimBeneficorAddressReinsertPot,`
	`    address payable _lateClaimBeneficorAddressDAO,`

	`    bytes memory _prefixStr,`
	`    uint256 _dilute_s1_75_timestamp,`
	`    uint256 _dilute_s2_50_timestamp,`
	`    uint256 _dilute_s3_0_timestamp`
	`){`
	`    lateClaimBeneficorAddressReinsertPot = _lateClaimBeneficorAddressReinsertPot;`
	`    lateClaimBeneficorAddressDAO = _lateClaimBeneficorAddressDAO;`
	`    dilute_s1_75_timestamp = _dilute_s1_75_timestamp;`
	`    dilute_s2_50_timestamp = _dilute_s2_50_timestamp;`
	`    dilute_s3_0_timestamp = _dilute_s3_0_timestamp;`
	`}`
	These assignments could be more gas-efficient if the variables were declared as immutable.

Code	Lines 18-47 (ClaimContract.sol)
	`/// @notice timestamp in UNIX Epoch timestep when the first dilution can happen.`
	`/// Claimers will only receive 75% of their balance.`
	`uint256 public immutable dilute_s1_75_timestamp;`
	
	`/// @notice timestamp in UNIX Epoch timestep when the second dilution can happen.`
	`/// Claimers will only receive 50% of their balance.`
	`uint256 public immutable dilute_s2_50_timestamp;`

	/// @notice timestamp in UNIX Epoch timestep when the third and final dilution can happen. /// All remaining unclaimed balances will be sent to the DAO and ReinsertPot. uint256 public immutable dilute_s3_0_timestamp; // ... /// @notice address of the reinsert pot that will receive half the diluted funds. address payable public immutable lateClaimBeneficorAddressReinsertPot; /// @notice address of the DAO address that will receive the other half of the diluted funds. address payable public immutable lateClaimBeneficorAddressDAO;

Result/Recommendation	Declare the beneficiary addresses and dilution timestamps as immutable to improve gas efficiency and enhance security:
	
	address payable public immutable lateClaimBeneficorAddressReinsertPot;
	
	address payable public immutable lateClaimBeneficorAddressDAO;
	
	uint256 public immutable dilute_s1_75_timestamp;
	
	uint256 public immutable dilute_s2_50_timestamp;
	
	uint256 public immutable dilute_s3_0_timestamp;

6.2.35 Unit Test Gaps

Severity: INFORMATIONAL

Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-core/issues/283

Attack / Description	The diamond-contracts-core repository has achieved 95% test coverage, but there are specific test gaps that should be addressed to ensure comprehensive coverage of all contract functionalities and edge cases. **Impact:** - **Incomplete Coverage**: Some code branches and scenarios remain untested - **Potential Hidden Bugs**: Untested code paths may contain undiscovered issues - **Documentation Gap**: Missing tests serve as missing documentation for expected behavior **Identified Test Gaps:** 1. nonReentrant Modifier branch not covered in reward function of BlockRewardHbbft contract 2. Validator score increase from MIN_SCORE after receiving frequent penalties 3. Test the withinAllowedRange modifier in the setReportDisallowPeriod function 4. onlyBlockRewardContract modifier of function handleFailedKeyGeneration 5. EpochNotYetFinished revert not triggering in handleFailedKeyGeneration function 6. Revert StakingPoolNotExist in setValidatorInternetAddress 7. getPreviousValidators function 8. Revert InvalidPossibleValidatorCount 9. Test for the public keys count check initialize in StakingHbbft 10. Test for the IP addresses count check: initialize in StakingHbbft 11. Test the withinAllowedRange modifier for the setDelegatorMinStake function 12. Test the stakingFixedEpochEndTime function

Result/Recommendation	Implement the missing test cases to achieve comprehensive coverage. The issue contains detailed test examples for each gap that should be added to the test suite.

6.2.36 Limited Flexibility in Proposal Fee Configuration

Severity: INFORMATIONAL
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-dao/issues/58
File(s) affected: DiamondDao.sol

Attack / Description	The DiamondDAO contract's initialize function sets up a fixed array of allowed proposal fee values (createProposalFeeAllowedParams). This array is hardcoded with nine specific values ranging from 10 to 90 ether. While this approach provides a clear set of options for proposal fees, it lacks flexibility for future adjustments or adaptations to changing economic conditions. **Impact:** • Inability to set fees outside the predefined range (10-90 ether) • Lack of granularity for fee adjustments (only 10 ether increments allowed) • Difficulty adapting to significant changes in the native token's value • Potential need for contract upgrade to modify fee options

Code	Lines 188-196 (DiamondDao.sol)
	`uint256[] memory createProposalFeeAllowedParams = new uint256[](9);`
	`for (uint256 i = 0; i < 9; ++i) {`
	`    createProposalFeeAllowedParams[i] = (i + 1) * 10 ether;`
	`}`
	
	`__initAllowedChangeableParameter(`

```
    this.setCreateProposalFee.selector,
    this.createProposalFee.selector,
    createProposalFeeAllowedParams
);
```

Result/Recommendation

Consider implementing a more dynamic fee structure using min-max range with step value:

```
uint256 public minProposalFee;

uint256 public maxProposalFee;

uint256 public proposalFeeStep;



function initializeProposalFeeParams(uint256 _min, uint256 _max, uint256 _step)

    external onlyGovernance {

    require(_min < _max && _step > 0, "Invalid fee parameters");

    minProposalFee = _min;

    maxProposalFee = _max;

    proposalFeeStep = _step;

}
```

6.2.37 Potential for Proposal ID Collisions in hashProposal Function

Severity: INFORMATIONAL
Status: ACKNOWLEDGED
Github: https://github.com/DMDcoin/diamond-contracts-dao/issues/59
File(s) affected: DiamondDao.sol

<table>
<tr>
<td><b>Attack / Description</b></td>
<td>

The DiamondDao contract uses a hashProposal function to generate unique identifiers for proposals. While the current implementation uses cryptographically secure methods (Keccak256), there remains a theoretical possibility of hash collisions, albeit extremely unlikely.

Impact: In the extremely unlikely event of a collision:

- Two different proposals could have the same ID
- This could lead to confusion in proposal tracking and voting
- It might potentially allow the overwriting of an existing proposal

</td>
</tr>
<tr>
<td><b>Code</b></td>
<td>

Lines 564-573 (DiamondDao.sol)

```
function hashProposal(
    address[] memory targets,
    uint256[] memory values,
    bytes[] memory calldatas,
    string memory description
) public pure virtual returns (uint256) {
    bytes32 descriptionHash = keccak256(bytes(description));

    return uint256(keccak256(abi.encode(targets, values, calldatas,
descriptionHash)));
}
```

</td>
</tr>
</table>

Result/Recommendation	Add additional unique elements to the hash input and implement collision checking:

```
function hashProposal(address[] memory targets, uint256[] memory values,
    bytes[] memory calldatas, string memory description) public view
returns(uint256) {
    bytes32 descriptionHash = keccak256(bytes(description));
    return uint256(keccak256(abi.encode(targets, values, calldatas,
        descriptionHash, block.timestamp, msg.sender)));
}


function propose(...) external {
    uint256 proposalId = hashProposal(...);
    require(!proposalExists(proposalId), "Proposal ID collision");
    // Rest of the function
}
```

6.2.38 Mismatch in Proposal Fee and Mention of Nonexistent Network Fees

Severity: INFORMATIONAL
Status: ACKNOWLEDGED

Github: https://github.com/DMDcoin/diamond-contracts-dao/issues/61
File(s) affected: DiamondDao.sol

Attack / 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. **Impact:** • Documentation mismatch with implementation • Confusion for users and developers • References to non-existent features
Result/Recommendation	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

6.2.39 Race Condition in _executeOperations Due to Limited governancePot

Severity: INFORMATIONAL
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-dao/issues/62
File(s) affected: DiamondDao.sol

Attack / 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.

Impact:

- Unfair execution order dependency
- Later proposals may fail due to insufficient funds
- No guarantee of execution for accepted proposals

Code

Lines 625-642 (DiamondDao.sol)

```
function _executeOperations(
    address[] memory targets,
    uint256[] memory values,
    bytes[] memory calldatas
) private {
    for (uint256 i = 0; i < targets.length; ++i) {
        uint256 execValue = calldatas[i].length == 0 ? values[i] : 0;
        (bool success, bytes memory returndata) = targets[i].call{ value: execValue
}(
            calldatas[i]
        );
```

```
            Address.verifyCallResult(success, returndata);

            if (execValue != 0) {
                governancePot -= execValue;
            }
        }
    }
```

Result/Recommendation	Consider adding logic to: • Partially fulfill transfers or fail gracefully • Implement a roll-over mechanism that allows accepted Open proposals that do not get executed because of lack of funds to be executed even after the 28 days window

6.2.40 Abstain Votes Are Ineffective

Severity: INFORMATIONAL
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-dao/issues/63
File(s) affected: DiamondDao.sol

Attack / 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 in the quorum determination, the current logic, which ignores abstain votes and does not rigorously enforce participation checks, will become insufficient.

Impact:

- Abstain votes don't affect proposal outcomes
- Participation thresholds not properly enforced
- Future upgrades may require redesign

Code

Lines 501-534 (DiamondDao.sol)

```
function _countVotes(
    uint256 proposalId,
    bool useSnapshot
) private view returns (VotingResult memory) {
    uint64 daoEpoch = proposals[proposalId].votingDaoEpoch;

    VotingResult memory result;

    address[] memory voters = getProposalVoters(proposalId);

    for (uint256 i = 0; i < voters.length; ++i) {
        address voter = voters[i];

        uint256 stakeAmount = 0;

        if (useSnapshot) {
            stakeAmount = daoEpochStakeSnapshot[daoEpoch][voter];
        } else {
            stakeAmount = stakingHbbft.stakeAmountTotal(voter);
        }

        Vote _vote = votes[proposalId][voter].vote;

        if (_vote == Vote.Yes) {
            result.countYes += 1;
            result.stakeYes += stakeAmount;
        } else if (_vote == Vote.No) {
            result.countNo += 1;
            result.stakeNo += stakeAmount;
        }
    }

    return result;
}
```

Note: Abstain votes are not counted in the result, only Yes and No votes affect the outcome.

Result/Recommendation	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.

6.2.41 Define and Use Constant Variables Instead of Using Literals

Severity: INFORMATIONAL
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-dao/issues/64
File(s) affected: QuorumCalculator.sol

Attack / Description	Rather than using constant literal values, create a constant state variable and reference it throughout the contract. Found 6 instances: • Line 169: createProposalFeeAllowedParams[4] = 50 ether; • Line 460: requiredExceeding = totalStakedAmount * (50 * 100) / 10000; • Line 462: requiredExceeding = totalStakedAmount * (33 * 100) / 10000; **Impact:** • Reduced code maintainability • Magic numbers without clear meaning

- Harder to update values consistently

Code

Lines 11-28 (QuorumCalculator.sol)

```
function lowMajorityQuorum(
    VotingResult memory vs,
    uint256 totalStakedAmount
) internal pure returns (bool) {
    uint256 requiredExceeding = totalStakedAmount * 2; // 2/6 = 1/3

    return vs.stakeYes * 6 >= vs.stakeNo * 6 + requiredExceeding;
}

function highMajorityQuorum(
    VotingResult memory vs,
    uint256 totalStakedAmount
) internal pure returns (bool) {
    uint256 requiredExceeding = totalStakedAmount * 3; // 3/6 = 1/2

    return vs.stakeYes * 6 >= vs.stakeNo * 6 + requiredExceeding;
}
```

Result/Recommendation

Define constants for commonly used literal values:

```
uint256 constant DEFAULT_PROPOSAL_FEE = 50 ether;

uint256 constant UPGRADE_THRESHOLD_PERCENT = 50;

uint256 constant STANDARD_THRESHOLD_PERCENT = 33;

uint256 constant BASIS_POINTS = 10000;
```

6.2.42 Event Is Missing Indexed Fields

Severity: INFORMATIONAL
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-dao/issues/65
File(s) affected: IDiamondDao.sol

Attack / Description	Index event fields make the field more quickly accessible to off-chain tools that parse events. 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. Found 12 instances in: • IDiamondDao.sol: ProposalCreated, ProposalCanceled, VotingFinalized, SubmitVote, SubmitVoteWithReason, SwitchDaoPhase, SetCreateProposalFee, SetIsCoreContract, SetChangeAbleParameters • MockStakingHbbft.sol: SetDelegatorMinStake, SetChangeAbleParameter, RemoveChangeAbleParameter **Impact:** • Less efficient off-chain event parsing • Increased query costs for indexed data • Reduced performance of event filtering
Result/Recommendation	Add indexed fields to events where appropriate:

```
event ProposalCreated(
    address indexed proposer,
    uint256 indexed proposalId,
    address[] indexed targets,
    // ... other fields
);
```

6.2.43 Unused Custom Error

Severity: INFORMATIONAL
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-dao/issues/66
File(s) affected: DiamondDao.sol

Attack / Description	It is recommended that the definition be removed when custom error is unused. Found 2 instances: • Line 55: error FunctionUpgradeNotAllowed(bytes4 funcSelector, address targetContract); • Line 56: error InvalidUpgradeValue(uint256 currentVal, uint256 newVal); **Impact:** • Code bloat • Confusion about intended functionality

	• Wasted gas for deployment
Result/Recommendation	Remove unused custom error definitions if they are not planned for future use, or implement the logic that would use these errors.

6.2.44 Repeated Access to currentPhaseProposals.length in Loop Condition

Severity: INFORMATIONAL
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-dao/issues/67
File(s) affected: DiamondDao.sol

Attack / 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. **Impact:** • Unnecessary gas costs • Inefficient storage reads • Performance degradation with larger arrays
Result/Recommendation	Cache the array length in a local variable before the loop: `uint256 proposalsLength = currentPhaseProposals.length;`

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

6.2.45 Hardcoded Governance Pot Address in Contract Logic

Severity: INFORMATIONAL
Status: ACKNOWLEDGED
Github: https://github.com/DMDcoin/diamond-contracts-core/issues/294
File(s) affected: BlockRewardHbbft.sol

Attack / Description	The BlockRewardHbbft contract uses a hardcoded governance pot address (0x2000000000000000000000000000000000000002 or similar) throughout its distribution logic. This hardcoded value reduces the flexibility and upgradeability of the contract, making it difficult to adapt to future governance system changes or different network deployments. The address is used in multiple critical operations: • Reward distribution calculations • Balance queries for governance pot • Transfer operations to the governance pot • Epoch reward allocations Without the ability to update this address through governance or administrative functions, the contract is locked into a specific governance infrastructure that cannot be changed without a full contract upgrade.

Impact:

1. **Reduced Flexibility**: Cannot adapt to new governance systems without contract upgrades
2. **Deployment Limitations**: Different networks require code modifications rather than configuration
3. **Migration Challenges**: Moving to a new governance pot address requires full contract redeployment
4. **Risk of Fund Lock**: If the governance pot address becomes compromised or needs to change, funds cannot be redirected

Proof of Concept:

```
// Current implementation with hardcoded address

address payable public constant GOVERNANCE_POT_ADDRESS =

    payable(0x2000000000000000000000000000000000000002);



function _distributeRewards() internal {

    // Funds are always sent to hardcoded address

    TransferUtils.transferNative(GOVERNANCE_POT_ADDRESS,
shares.governancePotAmount);

}



// If governance needs to move to a new address, there's no way to update it
// The contract would need to be redeployed entirely
```

Code

Lines 65-73 (BlockRewardHbbft.sol)

```
address payable public governancePotAddress;

/// @dev nominator of the epoch reward that get's forwarded to the
/// `governancePotAddress`. See also `governancePotShareDenominator`
uint256 public governancePotShareNominator;

/// @dev denominator of the epoch reward that get's forwarded to the
/// `governancePotAddress`. See also `governancePotShareNominator`
uint256 public governancePotShareDenominator;
```

And initialization at Line 147:

```
governancePotAddress = payable(0xDA0da0da0Da0Da0Da0DA00DA0da0da0DA0DA0dA0);
```

Result/Recommendation

1. Make the governance pot address configurable through initialization and governance:

```
address payable public governancePotAddress;



function initialize(

    address payable _governancePot,

    // other params

) external initializer {
    require(_governancePot != address(0), "Invalid governance pot");

    governancePotAddress = _governancePot;

    // rest of initialization

}



function setGovernancePot(address payable _newPot) external onlyGovernance {

    require(_newPot != address(0), "Invalid address");

    address oldPot = governancePotAddress;

    governancePotAddress = _newPot;

    emit GovernancePotUpdated(oldPot, _newPot);

}
```

2. Add events to track governance pot updates:

```
event GovernancePotUpdated(address indexed oldPot, address indexed newPot);
```

3. Implement access controls:
   - Only allow governance or authorized admin to update
   - Add time delays for critical address changes
   - Implement multi-sig requirements for governance pot changes

	4. Document the governance pot change process: ○ Create clear documentation on how to propose governance pot changes ○ Define the approval process and timeline ○ Outline emergency procedures if the governance pot is compromised

6.2.46 Commented Out Code in 'certified' Function

Severity: INFORMATIONAL
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-core/issues/295
File(s) affected: CertifierHbbft.sol

Attack / Description	The certified function in the CertifierHbbft contract contains commented-out code that references a staking contract check. The commented section suggests incomplete or planned functionality that was never fully implemented or was disabled for some reason.
	
	The commented code appears to be:
	`// IStakingHbbft stakingContract = IStakingHbbft(`
	`//     validatorSetContract.getStakingContract()`
	`// );`
	`// Additional checks may have been commented out here`

This indicates that there may have been plans to:

1. Verify certification status through the staking contract
2. Add additional validation layers beyond the current implementation
3. Cross-reference validator status with staking information

The presence of commented code in production contracts raises several concerns:

- **Unclear Intent**: It's not clear why the code was commented out rather than removed
- **Maintenance Burden**: Future developers may be confused about whether this code should be restored
- **Code Quality**: Commented code suggests incomplete refactoring or abandoned features

Impact:

1. **Code Clarity Issues**: Makes it difficult to understand the intended behavior
2. **Maintenance Confusion**: Future developers may not know if this code should be restored
3. **Incomplete Features**: May indicate unfinished functionality that could be important
4. **Documentation Gap**: Lacks explanation for why this code exists in commented form
5. **Version Control Confusion**: Unclear if this is temporary debugging code or a permanent change

Proof of Concept:

```
function certified(address _who) external view returns (bool) {

    if (_certified[_who]) {

        return true;

    }
    // This commented code suggests additional validation was planned
    // IStakingHbbft stakingContract = IStakingHbbft(
    //     validatorSetContract.getStakingContract()
    // );
    // if (stakingContract.isActive(_who)) {
    //     return true;
    // }
    // Current implementation only checks this
    address stakingAddress = validatorSetContract.stakingByMiningAddress(_who);
    return stakingAddress != address(0);
}
```

Result/Recommendation	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.

6.2.47 Typo in Parameter Name '_upcommingEpoch'

Severity: INFORMATIONAL

Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-core/issues/296
File(s) affected: KeyGenHistory.sol

Attack / Description

The KeyGenHistory contract contains a consistent spelling error in multiple locations where the parameter name _upcommingEpoch is used instead of the correct spelling _upcomingEpoch (with one 'm' instead of two). This typo appears in:

1. Function parameters in writePart function
2. Function parameters in writeAcks function
3. Modifier definition onlyUpcomingEpoch
4. All references to this parameter throughout the contract

The typo appears in multiple places:

```
function writePart(uint256 _upcommingEpoch, ...) external { }

function writeAcks(uint256 _upcommingEpoch, ...) external { }

modifier onlyUpcomingEpoch(uint256 _upcommingEpoch) { }
```

While this doesn't affect the contract's functionality, it impacts:

- **Code Professionalism**: Spelling errors reduce code quality perception
- **Developer Experience**: Can be confusing or frustrating for developers integrating with the contract
- **Code Searchability**: Makes it harder to search for related code using correct spelling
- **Documentation Consistency**: If documentation uses correct spelling, it won't match the code

Impact:

1. **Reduced Code Readability**: Spelling errors make the code appear less professional
2. **Developer Confusion**: Developers may question if this is intentional or an error
3. **Integration Challenges**: External systems using correct spelling will need to adjust
4. **Documentation Mismatch**: Creates inconsistency between documentation and implementation
5. **Search Difficulties**: Hard to find related code when searching with correct spelling
6. **Code Review Overhead**: Reviewers waste time identifying if this is intentional

Proof of Concept:

```
// Current implementation with typo

function writePart(

    uint256 _upcommingEpoch,  // Two 'm's - incorrect spelling

    uint256 _roundCounter,

    bytes memory _part

) external onlyUpcomingEpoch(_upcommingEpoch) onlyCorrectRound(_roundCounter) {

    // Function implementation

}

// Developer trying to integrate

// Searches documentation which uses "upcoming"

// Gets confused when they see "upcomming" in actual code

// May think they're looking at wrong version or documentation is outdated
```

Code	Lines 139-143 (KeyGenHistory.sol)
	`function writePart(`
	`    uint256 _upcomingEpoch,`
	`    uint256 _roundCounter,`
	`    bytes memory _part`
	`) external onlyUpcomingEpoch(_upcomingEpoch) onlyCorrectRound(_roundCounter) {`
	`And Lines 159-163:`
	`function writeAcks(`
	`    uint256 _upcomingEpoch,`
	`    uint256 _roundCounter,`
	`    bytes[] memory _acks`
	`) external onlyUpcomingEpoch(_upcomingEpoch) onlyCorrectRound(_roundCounter) {`

Result/Recommendation	1. **Correct the spelling throughout the contract:**
	`// Corrected function signatures`
	`function writePart(`
	`    uint256 _upcomingEpoch,  // Corrected spelling`
	`    uint256 _roundCounter,`
	`    bytes memory _part`
	`) external onlyUpcomingEpoch(_upcomingEpoch) onlyCorrectRound(_roundCounter) {`
	`    // Function implementation`

```
}


function writeAcks(

    uint256 _upcomingEpoch,  // Corrected spelling

    uint256 _roundCounter,

    bytes[] memory _acks

) external onlyUpcomingEpoch(_upcomingEpoch) onlyCorrectRound(_roundCounter) {

    // Function implementation

}


// Corrected modifier

modifier onlyUpcomingEpoch(uint256 _upcomingEpoch) {

    require(_upcomingEpoch == validatorSetContract.getUpcomingEpoch(), "Invalid
epoch");

    _;

}
```

2. **Update all internal references:**

```
// Ensure all variable references use correct spelling

uint256 upcomingEpoch = _upcomingEpoch;

require(upcomingEpoch > currentEpoch, "Epoch must be upcoming");
```

3. **Implement code quality checks:**
   - Add spell-checking to CI/CD pipeline
   - Use linters that catch common spelling errors
   - Implement pre-commit hooks for code quality
   - Maintain a project glossary of technical terms
4. **Testing considerations:**
   - Update test files to use correct spelling
   - Verify external contracts calling these functions
   - Check for any ABI-dependent integrations
5. **Documentation updates:**
   - Ensure all documentation uses correct spelling
   - Update API references and integration guides
   - Create migration guide if changing public interfaces

	Breaking Change Note: If this contract has already been deployed and other contracts are calling these functions, fixing the typo would be a breaking change. In this case: • Consider deprecating old functions and adding new correctly-spelled versions • Provide a migration period where both versions are supported • Clearly communicate the change to all integrators

6.2.48 Unused '_txPermission' Parameter in Initialize Function

Severity: INFORMATIONAL
Status: FIXED
Github: https://github.com/DMDcoin/diamond-contracts-dao/issues/60
File(s) affected: DiamondDao.sol

Attack / Description	The DiamondDao contract's initialize function accepts an _txPermission parameter but never uses it anywhere in the contract. The parameter is passed during initialization but is not: • Stored in any state variable • Validated beyond the zero address check • Referenced in any function • Used for access control or permissions • Passed to any other contract This suggests one of several scenarios: 1. **Incomplete Implementation**: The feature was planned but never completed 2. **Removed Functionality**: The feature was removed but the parameter remained 3. **Future-Proofing**: The parameter is included for future use but not documented

4. **Copy-Paste Error**: The parameter was copied from another contract where it is used

The presence of unused parameters:

- Increases gas costs for contract deployment and initialization
- Creates confusion about the contract's intended functionality
- May mislead integrators about what the contract does
- Adds unnecessary complexity to the interface

Impact:

1. **Wasted Gas**: Passing unused parameters costs gas for no benefit
2. **Code Confusion**: Developers may wonder why this parameter exists
3. **Integration Errors**: External systems may try to use this parameter incorrectly
4. **Maintenance Overhead**: Future developers must investigate if it's safe to remove
5. **Interface Bloat**: Makes the function signature unnecessarily complex
6. **Documentation Burden**: Requires explanation of why parameter exists but isn't used

Proof of Concept:

```
// Current implementation

function initialize(

    address _contractOwner,

    address _validatorSet,

    address _stakingHbbft,

    address _reinsertPot,
    address _txPermission,  // Passed but never used!
    address _bonusScore,
    address _lowMajorityDao,
    uint256 _createProposalFee,
    uint64 _startTimestamp
) external initializer {
    // Validation checks _txPermission is not zero
    if (
        _contractOwner == address(0) ||
        _validatorSet == address(0) ||
        _reinsertPot == address(0) ||
        _stakingHbbft == address(0) ||
        _txPermission == address(0) ||  // Checked but never stored
        _bonusScore == address(0) ||
        _lowMajorityDao == address(0) ||
        _createProposalFee == 0
    ) {
        revert InvalidArgument();
    }

    // _txPermission is never referenced after this point
    __Ownable_init(_contractOwner);
    __ReentrancyGuard_init();

    validatorSetHbbft = IValidatorSetHbbft(_validatorSet);
    stakingHbbft = IStakingHbbft(_stakingHbbft);
    reinsertPot = _reinsertPot;
    bonusScoreContract = IBonusScoreSystem(_bonusScore);
    lowMajorityDao = ILowMajorityDao(_lowMajorityDao);

    // No line storing: txPermission = _txPermission;
}

// Deployment wastes gas
```

```
DiamondDao dao = new DiamondDao();

dao.initialize(

    owner,

    validatorSet,

    staking,

    reinsertPot,

    someAddress,  // This address is validated but never used - wasted gas

    bonusScore,

    lowMajority,

    fee,

    timestamp

);
```

Code

Lines 145-157 (DiamondDao.sol)

```
function initialize(
    address _contractOwner,
    address _validatorSet,
    address _stakingHbbft,
    address _reinsertPot,
    address _txPermission,
    address _bonusScore,
    address _lowMajorityDao,
    uint256 _createProposalFee,
    uint64 _startTimestamp
) external initializer {
    if (
        _contractOwner == address(0) ||
```

Result/Recommendation

Option 1: Remove the unused parameter (Preferred if no future use planned)

```
function initialize(

    address _contractOwner,

    address _validatorSet,

    address _stakingHbbft,

    address _reinsertPot,

    // Removed: address _txPermission,

    address _bonusScore,

    address _lowMajorityDao,

    uint256 _createProposalFee,

    uint64 _startTimestamp

) external initializer {

    if (
        _contractOwner == address(0) ||
        _validatorSet == address(0) ||
        _reinsertPot == address(0) ||
        _stakingHbbft == address(0) ||
        // Removed: _txPermission == address(0) ||
        _bonusScore == address(0) ||
        _lowMajorityDao == address(0) ||
        _createProposalFee == 0
    ) {
        revert InvalidArgument();
    }
    // Rest of initialization
}
```

Option 2: Implement the intended functionality

If the parameter was meant to be used, implement it:

```
ITxPermission public txPermission;
```

```
function initialize(
    address _contractOwner,
    address _validatorSet,
    address _stakingHbbft,
    address _reinsertPot,
    address _txPermission,
    address _bonusScore,
    address _lowMajorityDao,
    uint256 _createProposalFee,
    uint64 _startTimestamp
) external initializer {
    // ... validation ...
    txPermission = ITxPermission(_txPermission);

    // Use it in relevant functions
}
```

```
function propose(...) external {
    require(txPermission.allowed(msg.sender), "Not allowed");
    // ... rest of function
}
```

Option 3: Document for future use

If keeping for future implementation:

```
function initialize(
    address _contractOwner,
    address _validatorSet,
    address _stakingHbbft,
    address _reinsertPot,
    address _txPermission,  // Reserved for future transaction permission checks
    address _bonusScore,
    address _lowMajorityDao,
    uint256 _createProposalFee,
    uint64 _startTimestamp
) external initializer {
    // Note: _txPermission parameter is validated but not currently used

    // It is reserved for future implementation of transaction permission checks

    // See enhancement issue #XXX for planned implementation



    // ... rest of initialization

}
```

6.3 Verify Claims

6.3.1 Epoch and Validator Rotation Logic

The audit must ensure safe and consistent epoch transitions across validator sets. Logic governing early epoch termination and disconnected validator detection should function reliably under edge conditions. Malicious or inactive validators must be penalized without risk of slashing honest participants or causing liveness faults.
Status: tested and verified ✅

6.3.2 Reward Allocation and Pot Distribution

The reward distribution mechanism must correctly allocate minted rewards to the designated smart contract pots (deltaPot, reinsertPot, governancePot). Validation includes overflow protection, access restrictions for withdrawal, and behavior across epochs with varying validator activity or node connectivity.
Status: tested and verified ✅

6.3.3 Bonus Score Incentivization Model

The bonus score system should fairly reflect validator performance without allowing manipulation or unintended inflation. The audit evaluates score calculations, weight impacts on reward scaling, and resistance against Sybil strategies or downtime masking.
Status: tested and verified ✅

6.3.4 Staking and Pool Reward Safety

All staking operations (deposit, withdrawal, delegation) must be validated for correctness and security. Epoch-bound logic must prevent reentrancy, race conditions, or data loss during stake transitions. Pool-level reward distribution should reflect accurate stake proportions and prevent gaming through timing or fragmentation.
Status: tested and verified ✅

6.3.5 Governance Controls and Role Permissions

Governance contracts must enforce strict access control over upgrade paths, treasury allocations, and protocol parameters. Only authorized roles (e.g., governance owner or DAO actors) may perform critical operations, and proposals must be validated with quorum enforcement and stake snapshot mechanisms.
Status: tested and verified ✅

7. Executive Summary

Three independent softstack experts performed an unbiased and isolated audit of the smart contract provided by the DMD Diamond team. The main objective of the audit was to verify the security and functionality claims of the smart contract. The audit process involved a thorough manual code review and automated security testing.

Overall, the audit identified a total of 48 issue, classified as follows:
- zero critical issue were found.
- 2 high severity issues were found.
- 14 medium severity issues were found.
- 17 low severity issues were discovered
- 15 informational issues were discovered

The audit report provides detailed descriptions of each identified issue, including severity levels, proof of concepts and recommendations for mitigation. We recommend the DMD Diamonds team to review the suggestions.

Update (21.11.2025): All previously identified findings have been successfully mitigated. The codebase was re-checked twice by softstack auditors to confirm the fixes and ensure no regressions were introduced.

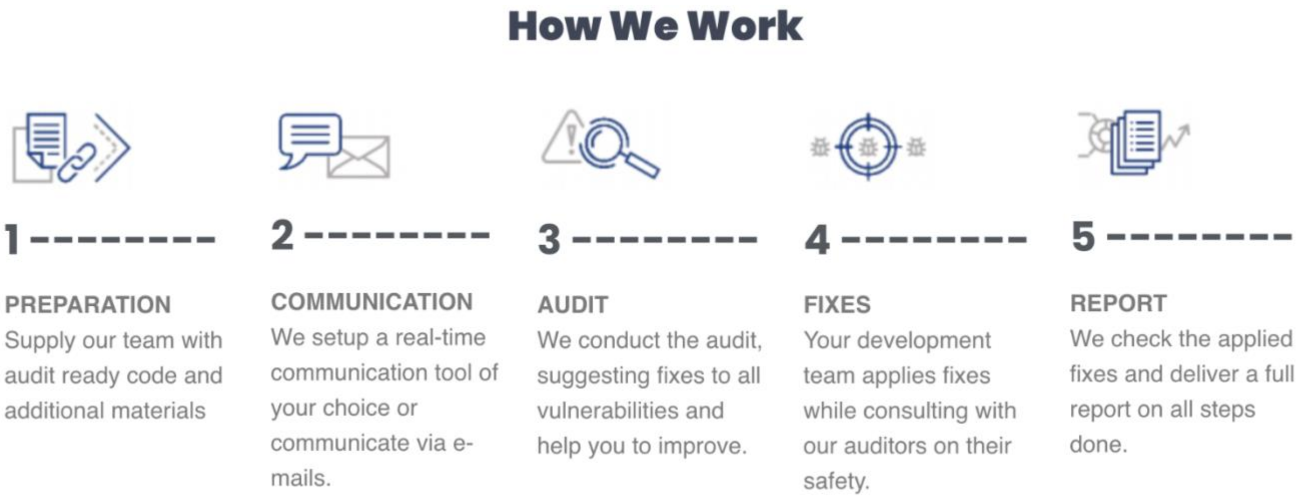
8. About the Auditor

Established in 2017 under the name Chainsulting, and rebranded as softstack GmbH in 2023, softstack has been a trusted name in Web3 Security space. Within the rapidly growing Web3 industry, softstack provides a comprehensive range of offerings that include software development, cybersecurity, and consulting services. Softstack's competency extends across the security landscape of prominent blockchains like Solana, Tezos, TON, Ethereum and Polygon. The company is widely recognized for conducting thorough code audits aimed at mitigating risk and promoting transparency.

The firm's proficiency lies particularly in assessing and fortifying smart contracts of leading DeFi projects, a testament to their commitment to maintaining the integrity of these innovative financial platforms. To date, softstack plays a crucial role in safeguarding over $100 billion worth of user funds in various DeFi protocols.

Underpinned by a team of industry veterans possessing robust technical knowledge in the Web3 domain, softstack offers industry-leading smart contract audit services. Committed to evolving with their clients' ever-changing business needs, softstack's approach is as dynamic and innovative as the industry it serves.

Check our website for further information: https://softstack.io



How We Work

1 PREPARATION
Supply our team with audit ready code and additional materials

2 COMMUNICATION
We setup a real-time communication tool of your choice or communicate via e-mails.

3 AUDIT
We conduct the audit, suggesting fixes to all vulnerabilities and help you to improve.

4 FIXES
Your development team applies fixes while consulting with our auditors on their safety.

5 REPORT
We check the applied fixes and deliver a full report on all steps done.