Security Review Report NM-0383
Vana Smart Contracts

NETHERMIND
SECURITY

(January 17, 2025)

Contents

1 Executive Summary

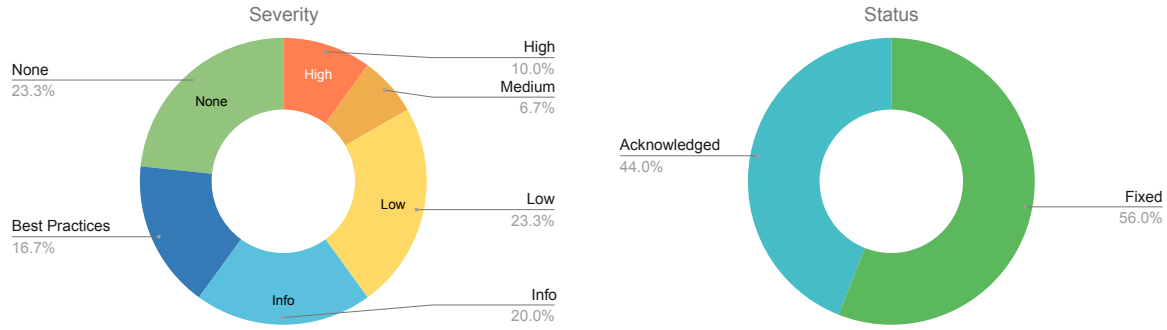
This document outlines the security review conducted by Nethermind Security for the Vana smart contracts. These contracts will be deployed on the Vana Network and allow users to directly provide their data to data consumers, in exchange for rewards.

User data is submitted to the Data Registry contract in the form of a URL, which is then associated to a unique file ID. This file ID can then be submitted to Data Liquidity Pools (DLPs), which is then validated by a Trusted Execution Environment (TEE) to ensure that the data is both valid and relevant. The TEE can then submit a proof on behalf of the user indicating that the data is valid, at which point the user will receive tokens in return. These Data Access Tokens (DAT) have voting and vote delegation capabilities allowing for governance on how data is shared.

The audited code comprises of 2060 lines of code written in the Solidity language, and the audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contract.

Along this document, we report fourteen points of attention, where three are classified as `High`, two are classified as `Medium`, sever are classified as `Low`, and eleven are classified as `Informational` or `Best Practices`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 discusses the risk rating methodology. Section 5 details the issues. Section 6 discusses the documentation provided by the client for this audit. Section 7 presents the compilation, tests, and automated tests. Section 8 concludes the document



(a) distribution of issues according to the severity

(b) distribution of issues according to the status

Fig 1: (a) Distribution of issues: Critical (0), **High** (3), **Medium** (2), **Low** (7), **Undetermined** (0), **Informational** (7), **Best Practices** (8). **(b) Distribution of status: Fixed** (15), **Acknowledged** (12), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Interim Report	December 02, 2024
Primary Report	December 19, 2024
Follow-up Report	January 17, 2025
Methods	Manual Review, Automated analysis
Repository	vana-com/vana-smart-contracts
Commit Hash	`dde493a2b13d1fc1bd4bc61afaf5e25915f67771`
Primary Commit Hash	`95cf2ec373ee5b2bd6f964017ec19e8b3d66fdb4`
Follow-up Commit Hash	`c472aecf485e6b829ca9c0bb5bde9fbd5b4af047`
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	contracts/root/DLPRootProxy.sol	5	1	20.0%	2	8
2	contracts/root/DLPRootImplementation.sol	915	165	18.0%	201	1281
3	contracts/root/interfaces/DLPRootStorageV1.sol	29	10	34.5%	10	49
4	contracts/root/interfaces/IDLPRoot.sol	189	10	5.3%	22	221
5	contracts/dlp/DataLiquidityPoolProxy.sol	5	1	20.0%	2	8
6	contracts/dlp/DataLiquidityPoolImplementation.sol	204	130	63.7%	59	393
7	contracts/dlp/interfaces/IDataLiquidityPool.sol	49	1	2.0%	8	58
8	contracts/dlp/interfaces/DataLiquidityPoolStorageV1.sol	18	6	33.3%	5	29
9	contracts/dataRegistry/DataRegistryImplementation.sol	123	107	87.0%	40	270
10	contracts/dataRegistry/DataRegistryProxy.sol	5	1	20.0%	2	8
11	contracts/dataRegistry/interfaces/DataRegistryStorageV1.sol	8	6	75.0%	2	16
12	contracts/dataRegistry/interfaces/IDataRegistry.sol	48	1	2.1%	6	55
13	contracts/teePool/TeePoolProxy.sol	5	1	20.0%	2	8
14	contracts/teePool/TeePoolImplementation.sol	254	164	64.6%	76	494
15	contracts/teePool/interfaces/ITeePool.sol	72	1	1.4%	7	80
16	contracts/teePool/interfaces/TeePoolStorageV1.sol	14	6	42.9%	6	26
17	contracts/token/DAT.sol	117	83	70.9%	35	235
	Total	**2060**	**694**	**33.7%**	**485**	**3239**

3 Summary of Issues

Primary Review

	Finding	Severity	Update
1	Anyone can steal contributor rewards from the DLP	High	Fixed
2	DLPRoot contract does not accept native tokens	High	Fixed
3	`_msgSender` should be used instead of `msg.sender`	High	Fixed
4	Removing a TEE can prevent a reward from being claimed	Medium	Acknowledged
5	TeePool's implementation for pausing and unpausing contract is not effective	Medium	Fixed
6	Blocking users in DAT token contract does not prevent from voting or delegation	Low	Fixed
7	Changing the ProofInstruction in DLP could prevent processing the reward request	Low	Acknowledged
8	DLP owner can withdraw their stake and continue running the DLP	Low	Acknowledged
9	DLP registration can be frontrun	Low	Acknowledged
10	Eligible DLP list should be rotated based on staked amounts on periodic basis	Low	Acknowledged
11	Incorrect rewards calculation in `estimatedDlpRewardPercentages`	Low	Fixed
12	Treasury's reward in an epoch can be lost	Low	Acknowledged
13	DLP eligibility and sub-eligibility threshold invariant is not enforced	Info	Fixed
14	Epoch storage field `isFinalised` inconsistently utilized	Info	Fixed
15	Epoch to claim should be bound in `claimStakeRewardUntilEpoch`	Info	Fixed
16	Fetching epochs for a DLP includes epoch0	Info	Fixed
17	TotalStakesScore can be updated for partial list of DLPs	Info	Acknowledged
18	Unnecessary use of checkpoints	Info	Acknowledged
19	Missing function to revoke permissions on a file in Data registry	Best Practices	Acknowledged
20	Reward calculation logic is duplicated in view and claiming flows	Best Practices	Fixed
21	Timestamp should indicate reward claim instead of reward amount	Best Practices	Acknowledged
22	UnauthorizedUserAction event log does not relay complete information	Best Practices	Fixed
23	Unnecessary payable operator in `addProof`	Best Practices	Fixed
24	Unused `MAINTAINER` role in `DataLiquidityPool`	Best Practices	Acknowledged
25	Unused imports	Best Practices	Fixed

Follow-up Metrics Review

	Finding	Severity	Update
1	`estimatedDlpRewardPercentages` does not consider non-top DLPs	Info	Acknowledged
2	Unnecessary `receive` function in `DLPRootImplementation`	Best Practices	Fixed

4 Protocol Overview

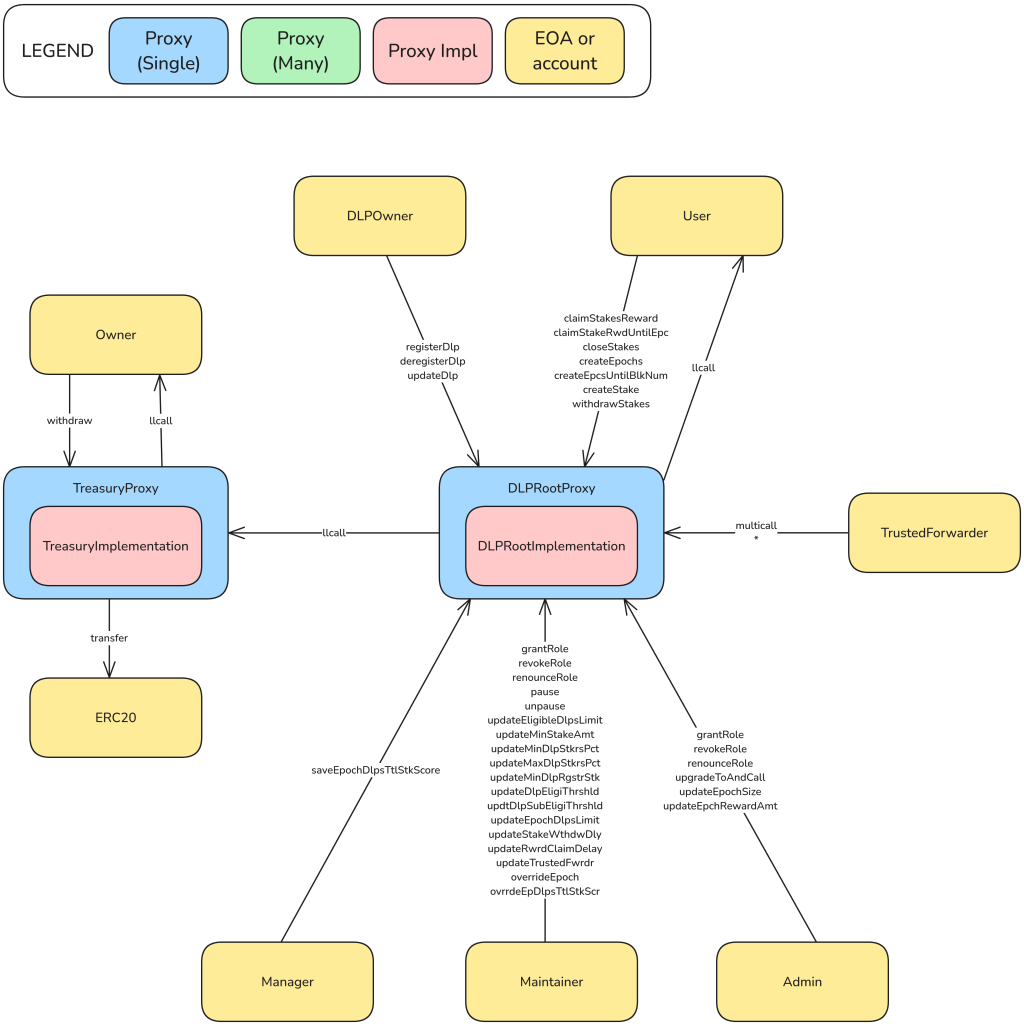
Root: The DLP Root contract manages the registration and reward distribution for Data Liquidity Pools (DLPs) in the Vana ecosystem. It operates on an epoch-based system, where the top 16 most staked DLPs and their stakers receive rewards at the end of each epoch. The contract allows users to stake VANA tokens as guarantors for DLPs, with rewards distributed based on the staking position at the beginning of each epoch.

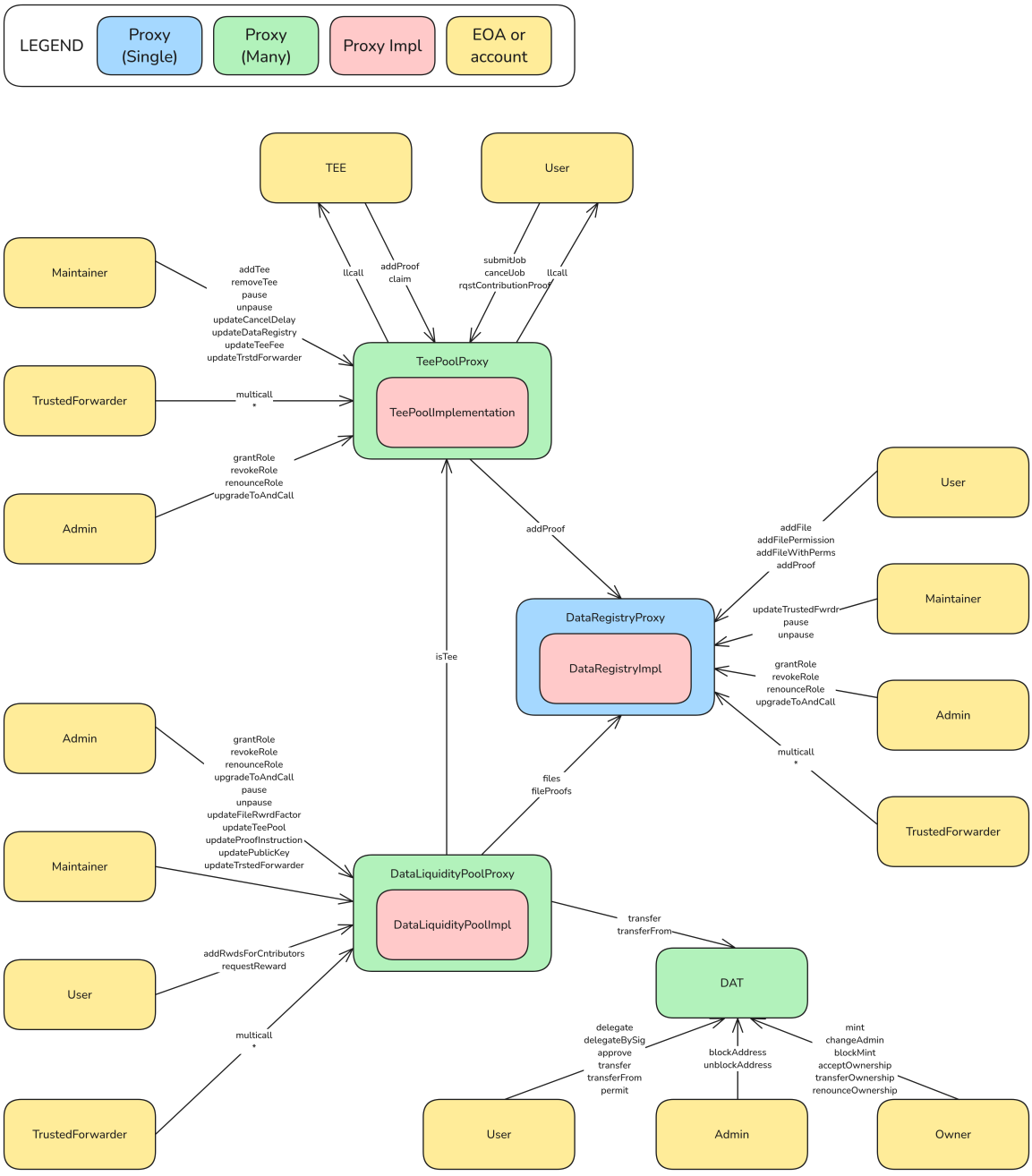
TeePool: The TEE Pool contract manages and coordinates the TEE Validators and serves as an escrow for holding fees associated with validation tasks. Users pay a fee to submit data for validation, and the contract ensures that the validators process the data and provide proof of validation.

DataRegistry: The data registry contract functions as a central repository for managing all data within the network, functioning as a comprehensive file catalog. It allows users to add new files to the system, with each file receiving a unique identifier for future reference. The contract manages access control for these files, enabling file owners to grant specific addresses permission to access their files. It also handles the storage of file metadata, including any offchain proofs or attestations related to file validation, which can include various metrics such as authenticity, ownership, and quality scores. Users can retrieve detailed information about any file in the registry using its unique identifier, including its permissions and associated proofs.

DataLiquidityPool: A Data Liquidity Pool (DLP) is a core component of the Vana ecosystem, designed to transform raw data into a liquid asset. It functions as a smart contract on the Vana blockchain that allows users to monetize, control, and govern their data in a decentralized manner. Each DLP can have its own token, providing contributors with ongoing rewards and governance rights.

DAT: The token that is given as a reward for providing data to a data liquidity pool, it has governance rights and can be used for voting.




LEGEND
Proxy (Single)
Proxy (Many)
Proxy Impl
EOA or account
TEE
User
Maintainer
addTee
removeTee
pause
unpause
updateCancelDelay
updateDataRegistry
updateTeeFee
updateTrstdForwarder
llcall
addProof
claim
submitJob
canceJob
rqstContributionProof
llcall
TeePoolProxy
TeePoolImplementation
TrustedForwarder
multicall
*
grantRole
revokeRole
renounceRole
upgradeToAndCall
Admin
addProof
User
addFile
addFilePermission
addFileWithPerms
addProof
Maintainer
updateTrustedFwrdr
pause
unpause
DataRegistryProxy
DataRegistryImpl
grantRole
revokeRole
renounceRole
upgradeToAndCall
Admin
multicall
*
TrustedForwarder
isTee
Admin
grantRole
revokeRole
renounceRole
upgradeToAndCall
pause
unpause
updateFileRwrdFactor
updateTeePool
updateProofInstruction
updatePublicKey
updateTrstedForwarder
files
fileProofs
Maintainer
DataLiquidityPoolProxy
DataLiquidityPoolImpl
transfer
transferFrom
addRwdsForCntributors
requestReward
User
DAT
multicall
*
delegate
delegateBySig
approve
transfer
transferFrom
permit
blockAddress
unblockAddress
mint
changeAdmin
blockMint
acceptOwnership
transferOwnership
renounceOwnership
TrustedForwarder
User
Admin
Owner

5 Risk Rating Methodology

The risk rating methodology used by Nethermind follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely an attacker will uncover and exploit the finding. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to Motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to formally pass to the client;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Findings (Primary Review)

6.1 [High] Anyone can steal contributor rewards from the DLP

File(s): DataRegistryImplementation.sol DataLiquidityPoolImplementation.sol

Description: function addProof is an external function that allows any one to submit a proof for a fileId. With a valid proof, the owner of the file can claim contributor rewards from the DataliquidityPool by calling function requestReward.

```
function addProof(uint256 fileId, Proof memory proof) external override whenNotPaused {
    uint256 cachedProofCount = ++_files[fileId].proofsCount;

    _files[fileId].proofs[cachedProofCount] = proof;

    emit ProofAdded(fileId, cachedProofCount);
}
```

An attacker can potentially drain the DataliquidityPool 's contributor rewards bucket by executing the below strategy.

As a first step of strategy, Attacker goes through the normal flow and adds a new file to the dataRegistry. He then requests for attestation of the file to the TeePool by paying the necessary fee. The proof will be generated by the TeePool which is submitted to the DataRegistry for the file. A good quality file will ensure that the reward size is large due to good score which increase the reward amount. Now, the attacker call the nominated DLP's requestReward() to claim the contributor reward.

```
function requestReward(uint256 fileId, uint256 proofIndex) external override whenNotPaused nonReentrant {
    IDataRegistry.Proof memory fileProof = dataRegistry.fileProofs(fileId, proofIndex);
    //...

    bytes32 _messageHash = keccak256(
        abi.encodePacked(
            registryFile.url,
            fileProof.data.score,
            fileProof.data.dlpId,
            fileProof.data.metadata,
            fileProof.data.proofUrl,
            fileProof.data.instruction
        )
    );

    address signer = _messageHash.toEthSignedMessageHash().recover(fileProof.signature);

    if (!teePool.isTee(signer)) {
        revert InvalidAttestator();
    }

    //..
}
```

After completing the normal flow, the attacker now has a valid proof from the previous transaction. The attacker can start adding the new files to the DataRegistry and directly submit the above proof to each of the files by calling addProof on the DataRegistry for each of the files added. Attacker can then start claiming rewards from the DLP using the unique fileId and the proof to claim the contributor rewards.

Doing this strategy, attacker can potentially steal all of the contributor rewards in the DLP.

NOTE: Because of the restriction where DLP smart logic can't really be changed our recommendation no longer is applicable, so your own fix approach can be used instead.

Recommendation(s): The _messageHash used for verifying if the signer is an activeTee does not include the fileId, and hence can be validated for all files. Adding fileId to _messageHash will prevent reusing of the proof against other files. Also, consider restricting the addProof to be called by valid tees only.

Status: Fixed

Update from the client: Client fixed the issue by making the file Url unique in the data registry. This will ensure the same proof cannot be used across other files as url is part of the hash message while verifying the signer.

6.2 [High] DLPRoot contract does not accept native tokens

File(s): DLPRootImplementation.sol

Description: The DLPRootImplementation contract allows any one to participate in the protocol ecosystem by staking vana tokens. The participants can be a simple staking user or can be a DLP. It rewards the participants with vana tokens using an epoch reward system. In order to participate as user or as a DLP, the contract has functions to accept vana tokens for staking and withdrawing the staked tokens.

```
function registerDlp(
    DlpRegistration calldata registrationInfo
) external payable override whenNotPaused nonReentrant {
    _createEpochsUntilBlockNumber(block.number);
    _registerDlp(registrationInfo);
}

function createStake(uint256 dlpId) external payable override nonReentrant whenNotPaused {
    _createEpochsUntilBlockNumber(block.number);
    _createStake(msg.sender, dlpId, msg.value);
}
```

To reward the participants with vana tokens, the DLPRootImplementation contract should have a mechanism for inflow of vana tokens. As vana tokens are the native tokens, there should be a receive/fallback function in the contract to support adding tokens for reward distribution. The receive function is missing from the DLPRootImplementation contract.

Recommendation(s): Add a receive function to the DLP root contract.

Status: Fixed

Update from the client:

6.3 [High] _msgSender should be used instead of msg.sender

File(s): DLPRootImplementation.sol

Description: When supporting Metatransactions for a contract, defining how the caller address is resolves is critical to the proper functioning of normal as well as metatransactions. During normal transaction, msg.sender is the actual caller. But, during Metatransaction, the msg.sender is the forwarder's address, while the actual caller only known to the forwarder.

The ERC-2771 standard for meta transactions abstracts out these difference in the caller address and provides _msgSender() function to resolve the actual caller address.

The issue arises due to use of msg.sender in the below functions of the DLPRootImplementation contract. As a result, each of these functions will exhibit a different behaviour based on whether it is a normal transaction or a metatransaction.

Modifier onlyDlpOwner used in updateDlp and deregisterDlp so these will revert in multicalls

```
 modifier onlyDlpOwner(uint256 dlpId) {
      if (_dlps[dlpId].ownerAddress != msg.sender) {
          revert NotDlpOwner();
      }
      _;
  }
```

Function createStake multicall stake creations will always belong to the trustedforwarder

```
function createStake(uint256 dlpId) external payable override nonReentrant whenNotPaused {
      _createEpochsUntilBlockNumber(block.number);
      _createStake(msg.sender, dlpId, msg.value);
  }
```

Function closeStakes a stake opened by user via regular call, impossible to close stake via multicall

```
 function closeStakes(uint256[] memory stakeIds) external override nonReentrant whenNotPaused {
      _createEpochsUntilBlockNumber(block.number);
      for (uint256 i = 0; i < stakeIds.length; i++) {
          _closeStake(msg.sender, stakeIds[i]);
      }
  }
```

Function withdrawStakes a stake closed by a user via regular call, impossible to close via multicall

```
function withdrawStakes(uint256[] memory stakeIds) external override nonReentrant whenNotPaused {
        _createEpochsUntilBlockNumber(block.number);
        for (uint256 i = 0; i < stakeIds.length; i++) {
            _withdrawStake(msg.sender, stakeIds[i]);
        }
    }
```

Recommendation(s): Replace `msg.sender` with `_msgSender()` provided by `ERC-2771` standard. This will ensure, the resolving of the actual call is always delegated to `ERC-2771` standard which knows how to resolve the actual caller address.

Status: Fixed

Update from the client: Fixed

6.4 [Medium] Removing a TEE can prevent a reward from being claimed

File(s): DataLiquidityPoolImplementation.sol

Description: Contributor reward can claimed for the file after the proof is submitted by the assigned tee. In order to claim the contributor reward, function `requestReward` is invoked with `fileId` and `proofIndex`. The function computes the eligible reward amount for the contributor and transfers it. While processing the rewards, the function check if the signer of proof is a tee.

```
function requestReward(uint256 fileId, uint256 proofIndex) external override whenNotPaused nonReentrant {
    IDataRegistry.Proof memory fileProof = dataRegistry.fileProofs(fileId, proofIndex);
    //...

    address signer = _messageHash.toEthSignedMessageHash().recover(fileProof.signature);

    if (!teePool.isTee(signer)) {
        revert InvalidAttestator();
    }

    //...
}
```

Refer to the below codesnippet where an address is recognised as tee only if it is currently active.

```
function isTee(address teeAddress) external view override returns (bool) {
    return _tees[teeAddress].status == TeeStatus.Active;
}
```

Consider a case where the proof was submitted by a tee, but was removed by the `maintainer` role. In such cases, if the rewards were not claimed before the removed of tee, then such rewards cannot be claimed.

Recommendation(s): As jobs can be processed by active tees, post the proof is submitted, as long as tee is listed in `_teeList`, it should be safe to assume, the tee was a valid attestator for the proof. This will ensure that contributor can claim rewards even when tees are removed from active tee list.

Status: Acknowledged

Update from the client: Not fixed, we are ok with this edge case

6.5 [Medium] TeePool's implementation for pausing and unpausing contract is not effective

File(s): TeePoolImplementation.sol

Description: `TeePoolImplementation` contract derives from `PausableUpgradeable` library of openzeppelin to support pausing of the contract incase of an unexpected situation and unpause it when the situation is resolved.

The pausing and unpausing needs to be enabled at external function level in the implementation contract to enable and disable functionality to mitigate the risk and hence needs to be implemented by the derived contract using the `whenNotPaused` and `whenPaused` modifiers.

These modifiers should be assigned to external functions of the contract to enable or disable when the contract is paused.

In the `TeePoolImplementation` contract, these modifiers are not attached to any of the functions. Hence, even though the pause is called, it will not be effective.

Recommendation(s): Attach the `whenNotPaused` and `whenPaused` modifiers to enable and disable functions during pause.

Status: Fixed

Update from the client: Fixed

6.6 [Low] Blocking users in DAT token contract does not prevent from voting or delegation

File(s): DAT.sol

Description: DAT token contract supports blocking transfer funds of funds if the sender is on the blocked list. The function _update checks for the sender in the blockList, if found will revert the transaction. But since the block list only restricts the sender, it is possible for someone to send tokens to a user on blocked list.

```
function _update(
        address from,
        address to,
        uint256 amount
    ) internal override(ERC20, ERC20Votes) whenNotBlocked(from) {
        super._update(from, to, amount);
}

modifier whenNotBlocked(address from) {
    if (_blockList.contains(from)) {
        revert UnauthorizedUserAction(msg.sender);
    }
    _;
}
```

As DAT token is also a governance tokens, hence holders will have voting rights. Blocked holders will still have full voting powers including delegation to another address. DAT token contract does not implement restrict the blocked holders from participating in voting or delegation.

Recommendation(s): Override the votes contract related functions to apply the restriction.

Status: Fixed

Update from the client: Fixed

6.7 [Low] Changing the ProofInstruction in DLP could prevent processing the reward request

File(s): DataLiquidityPoolImplementation.sol TeePoolImplementation.sol

Description: The proof for the file is added by the Active Tee validator as part of Job processing In the Tee Pool implementation contract. When the requestReward(...) function is called on DLP, the file proof information is read from the data registry.

```
  function requestReward(uint256 fileId, uint256 proofIndex) external override whenNotPaused nonReentrant {
        IDataRegistry.Proof memory fileProof = dataRegistry.fileProofs(fileId, proofIndex);

        if (keccak256(bytes(fileProof.data.instruction)) != keccak256(bytes(proofInstruction))) {
            revert InvalidProof();
        }
```

In DLP, the owner has ability to update the proofInstruction state variable by calling updateProofInstruction(...)

```
function updateProofInstruction(string calldata newProofInstruction) external override onlyOwner {
        proofInstruction = newProofInstruction;

        emit ProofInstructionUpdated(newProofInstruction);
    }
```

But, updating the proofInstruction state variable could cause a revert for proof already submitted by the Tee Pool but, reward request were not processed. The revert will happen in the requestReward(...) function where it compares the state variable with the fileProof.data.instruction field in the fileProof already submitted.

When they dont match, the call will revert with InvalidProof error.

NOTE: You don't need to provide a fix for this one. During the meeting we had discussed how adding logic to refunds users in this very rare case would add complexity. Instead it is the responsibility of the DLP entity to reimburse the user offchain should this happen.

Recommendation(s):

Status: Acknowledged

Update from the client: This is unlikely to occur and in the case that it does, the DLP entity itself should reimburse the user offchain.

6.8 [Low] DLP owner can withdraw their stake and continue running the DLP

File(s): DLPRootImplementation.sol

Description: While registering as a new DLP, the DLP owner needs to provide stake greater than or equal to the `minDlpRegistrationStake`. As the DLP attracts more users to stake their tokens, the DLP could get listed into the rewards eligible list resulting in rewards. At this point, the owner of DLP could close and withdraw their staked amount and continue to operate as DLP owner. This allows the DLP owner to operate multiple DLPs while reducing their risk.

Recommendation(s): Consider adding a flag to stake storage data that indicates whether it is a "registration stake", and special logic preventing closure of a stake while it's still active can occur in this case. Alternatively, if the risk of DLP registrants closing their stakes is determined to be negligible by the Vana team this finding can be acknowledged.

Status: Acknowledged

Update from the client: Not fixed, we are ok with this scenario

6.9 [Low] DLP registration can be frontrun

File(s): DLPRootImplementation.sol

Description: As the economic commitment to register the DLP is low, it is possible that a bad actor can front run the registration of DLP and claim ownership of the DLP address. Once the DLP address is added the `dlpIds` mapping, that address cannot be used for registration leading to DOS for the real owner.

```
if (dlpIds[registrationInfo.dlpAddress] != 0) {
    revert InvalidDlpStatus();
}
```

DLP's registration with low registration stake amount and also the ability for the owner to claim and withdraw registration stake any time opens up this vulnerability and hence it could be abused by bad actors effecting the protocol key participant.

As the DLP address is obtained post deployment, if a bad actor claims the Dlp address and configure owner and treasury addresses, the rewards and management of DLP will be in the control of the bad actor.

Recommendation(s): As the intention of low registration stake amount was to encourage more participation, the recommendation is to lock the registration stake amount for longer periods. This will discourage the bad actors to participate in such abuse.

If a "get owner" function were exposed in the DLP contract, then it could allow for a check during registration in the root contract that the caller address is the owner of the DLP which would prevent this issue, an example code snippet is shown below:

```
require(IDataLiquidityPool(registrationInfo.dlpAddress).ownerAddress==msg.sender,"Not DLP Owner");
```

Status: Acknowledged

Update from the client: Not a problem as the DLP creators should register their DLP in the very next step after deployment

6.10 [Low] Eligible DLP list should be rotated based on staked amounts on periodic basis

File(s): DLPRootImplementation.sol

Description: The DLP Root contract intends to reward the top DLPs based on staking. That means amounts staked into DLPs defines the `_eligibleDlpsList` list. Those who have the highest stake are the ones who should be getting the rewards for the subsequent epochs.

Lets assume a scenario where `_eligibleDlpsList` is operating at max limit, say 16 DLPs are already identified. At this point, if a new DLP is registered with a stake amount exceeding the amount held by one of the members in the `_eligibleDlpsList`, the new DLP should be part of the list.

But, since the `_eligibleDlpsList` is operating at max limit, the new DLP is not considered for the `_eligibleDlpsList` even when the staked amount was higher. The new DLP will get the opportunity for rewards only if one of the DLP gets disqualifed due to unstaking or unregister. This approach of updating the `_eligibleDlpsList` list might discourage some of the new DLPs to come onboard and participate in the ecosystem.

This behavior can potentially be abused by a malicious actor by registering many new DLPs until the `eligibleDlpsLimit` is met, at which point any new genuine DLP owners who attempt to register will not be added to the eligible list. This type of attack has an economic cost for the actor since they must provide a stake for each registration

Recommendation(s): Consider implementing a mechanism to remove the a DLP with the lowest computed stake amount from `_eligibleDlpsList` when a different DLP recieves a stake that is greater than the existing lowest computed stake DLP.

Status: Acknowledged

Update from the client: We have a way to rotate Eligible DLPs by updating the `dlpEligibilityThreshold` (+ a small stake for the DLP that are above the `dlpEligibilityThreshold` to update the status)

6.11 [Low] Incorrect rewards calculation in `estimatedDlpRewardPercentages`

File(s): DLPRootImplementation.sol

Description: The DLP Reward estimates are readable using `estimatedDlpRewardPercentages(...)` function for a given list of DLPs. The DLPs are provided as input by the caller, and based on the function logic, the estimated reward amount for each DLP is returned. There are two issues identified in the logic:

1. The `topDlps` fetched for this purpose is hard coded as 16, where instead the `eligibleDlpsLimit` should be used to consider all eligible DLPs;
2. Reward should be returned for only top DLPs and for others, the reward should be 0;

Refer to the below code snippet where the logic loops over `dlpIds` provided as input by the caller to compute the estimated reward for each DLP. The function does check if each DLP is in the `topDlp` and a boolean variable `isInTop` is set to true in this case, however this variable is left unused after being set. Rewards are given to all DLPs regardless of whether they are considered a top DLP or not, as shown below:

```
function estimatedDlpRewardPercentages(
    uint256[] memory dlpIds
) external view override returns (DlpRewardApy[] memory) {
    // Hardcoded to use the top 16 DLPs, should be the size of the eligible DLPs set
    uint256[] memory topDlps = topDlpIds(16);
    //...
    for (uint256 i = 0; i < dlpIds.length; i++) {
        //...
        // Check if DLP is in top 16
        for (uint256 j = 0; j < topDlps.length; j++) {
            if (topDlps[j] == dlpId) {
                isInTop = true;
                break;
            }
        }

        // Regardless of `isInTop` rewards are distributed anyways
        // If DLP is not in the `topDlps` array they still get rewards
        uint256 rewardAmount = (dlpStake * epochRewardAmount) / totalStakeAmount;
        uint256 stakeReward = (((100e18 * rewardAmount) / dlpStake) * stakersPercentage) / 1e20;

        //...
    }
    return result;
}
```

As such, even dlps that are not in the top Dlp list are estimated to have a non zero reward amount. This looks incorrect.

Recommendation(s): Consider adding a check to ensure the DLP has the value `isInTop` set to true before assigning values to `rewardAmount` and `stakeAmount`, this will ensure that rewards are only given to the correct DLPs. For the `topDlpIds()` function call consider getting all eligible DLPs instead of using a hardcoded value.

Status: Fixed

Update from the client: We want to estimate the APY for all DLPs that are eligible even though they are not in top 16

6.12 [Low] Treasury's reward in an epoch can be lost

File(s): DLPRootImplementation.sol

Description: As the new epochs are created for a block, the previous epochs are finalised through `_finalizeEpoch(...)` function. The `_finalizeEpoch(...)` computes the reward for each of the DLP that participated in the epoch and credits the reward amount for each DLP respectively.

Each DLP can configure there `stakersPercentage`. Any reward left is transferred to the DLP's treasury address. The issue lies in the below code snippet, where the return value for the low level call function is not being checked.

```
    (bool success, ) = dlp.treasuryAddress.call{
                    value: (epochDlp.rewardAmount * (100e18 - epochDlp.stakersPercentage)) / 100e18
                }("");

    //skip this in phase 1 to avoid reverting the whole transaction when the treasury transfer fails
    //                if (!success) {
    //                    revert TransferFailed();
    //
```

Incase the call fails, it will be ignored. As a result of this, the DLP's treasury will not receive the reward for the epoch. The contract does not have a function for the treasury to claim the rewards that did not succeed during the `_finalizeEpoch(...)` function. As a result, the treasury rewards are locked in the DLPRoot and will not be available for DLP to claim.

Recommendation(s): The recommendation is to track and credit the DLP treasury rewards in a separate mapping. Provide a function for the DLP treasury to claim those rewards.

Status: Acknowledged

Update from the client: We now save which DLP successfully received the reward and, if there are any errors, we will send the reward manually from the treasury

6.13 [Info] DLP eligibility and sub-eligibility threshold invariant is not enforced

File(s): `DLPRootImplementation.sol`

Description: The logic in the `DLPRootImplementation` surrounding the eligibility and sub-eligibility thresholds are designed in a way that assume that the eligibility threshold is always greater than the sub-eligibility threshold. However, in the setter functions `updateDlpEligibilityThreshold` and `updateDlpSubEligibilityThreshold` there are no checks to ensure that this invariant is held.

Recommendation(s): Consider adding checks to `updateDlpEligibilityThreshold` and `updateDlpSubEligibilityThreshold` to ensure that the sub-eligibility threshold is always lower than the eligibility threshold.

Status: Fixed

Update from the client: Fixed

6.14 [Info] Epoch storage field `isFinalised` inconsistently utilized

File(s): `DLPRootImplementation.sol`

Description: The struct `Epoch` is used to track information about an epoch, and the mapping `_epochs` uses epoch IDs as a key to be able to track historical information about all epochs that have occurred. One of the fields in the `Epoch` struct is the boolean `isFinalised` which is used inconsistently by the `DLPRootImplementation` contract.

```
struct EpochInfo {
    uint256 startBlock;
    uint256 endBlock;
    uint256 reward;
    bool isFinalised;
    uint256[] dlpIds;
}
```

When the DLP root contract is being initialized, a "zero epoch" is created and immediately set as finalized. However, all other epochs that are generated through `_createEpochsUntilBlockNumber` and finalized through `_finalizeEpoch` will never have their `isFinalised` bit set to true. When querying information about epochs through the external view function `epochs`, only the "zero epoch" will be considered finalised, and it is not possible for any other "natually generated" epoch to be finalised, even once it has been passed through the function `_finalizeEpoch`.

Recommendation(s): Consider setting epochs as finalized through the function `_finalizeEpoch`.

Status: Fixed

Update from the client: Fixed

6.15 [Info] Epoch to claim should be bound in `claimStakeRewardUntilEpoch`

File(s): DLPRootImplementation.sol

Description: The `DLPRootImplementation` contract provides an external function to claim rewards up to a user specified epoch number by calling the `claimStakeRewardUntilEpoch(...)` function.

```
function claimStakeRewardUntilEpoch(...) external nonReentrant whenNotPaused {
    _createEpochsUntilBlockNumber(block.number);
    _claimStakeRewardUntilEpoch(stakeId, lastEpochToClaim);
}
```

The problem arises when a user passes `lastEpochToClaim` parameter with a value greater than the current epoch.

```
function _claimStakeRewardUntilEpoch(uint256 stakeId, uint256 lastEpochToClaim) internal {
    // ...

    while (epochToClaim > 0 && epochToClaim <= lastEpochToClaim) {
        Epoch storage epoch = _epochs[epochToClaim];
        EpochDlp storage epochDlp = epoch.dlps[stake.dlpId];

        if (
            epochToClaim == 0 ||
            epochDlp.totalStakesScore == 0 ||
            (stake.endBlock > 0 && epoch.endBlock > stake.endBlock)
        ) {
            break;
        }

        uint256 stakeScore = calculateStakeScore(stake.amount, stake.startBlock, epoch.endBlock);
        // ...
    }
    // ...
}
```

When the `lastEpochToClaim` is greater than `epochsCount`, then the above logic will read an empty epoch from the `_epochs` collection. As the epoch is uninitialized, `epoch.endBlock` will hold a value of 0. So, when the `stakeScore` is computed in `calculateStakeScore(...)` function, it will revert while computing the `daysStaked` as `stakeStartBlock` will have a value greater than 0 and `blockNumber` will be 0 as its value.

```
function calculateStakeScore(
    uint256 stakeAmount,
    uint256 stakeStartBlock,
    uint256 blockNumber
) public view returns (uint256) {
    uint256 daysStaked = (blockNumber - stakeStartBlock) / daySize;
    return (stakeAmount * _getMultiplier(daysStaked)) / 100;
}
```

Recommendation(s): Consider adding a validation in `claimStakeRewardUntilEpoch` to restrict the `lastEpochToClaim` to be less than `epochsCount` state variable. Alternatively, limit it to `epochsCount - 1` incase it was larger value.

```
function claimStakeRewardUntilEpoch(
    uint256 stakeId,
    uint256 lastEpochToClaim
) external nonReentrant whenNotPaused {
    _createEpochsUntilBlockNumber(block.number);
    uint256 maxEpoch = Math.min(lastEpochToClaim, epochcount-1);
    _claimStakeRewardUntilEpoch(stakeId, maxEpoch);
}
```

Status: Fixed

Update from the client: Fixed

6.16 [Info] Fetching epochs for a DLP includes epoch0

File(s): DLPRootImplementation.sol

Description: The function `dlps(...)` returns the information about the DLP based on the `dlpId`. The logic of reading the epochs for the DLP also returns Epoch 0 for every call. Refer to the below code snippet where memory array for `dlp.epochIdsCount + 1` size is created and all epoch ids applicable for the DLP are assigned to the array.

```
uint256[] memory epochIds = new uint256[](dlp.epochIdsCount + 1);
for (uint256 i = 1; i <= dlp.epochIdsCount; i++) {
    epochIds[i] = dlp.epochIds[i];
}
```

Since the `epochIds` are assigned from index 1, the first element of the `epochIds` array will always remain empty with a value of zero which refers to Epoch 0.

Recommendation(s): Consider adjusting the loop logic to fill the array from index 0 rather than index 1 to avoid Epoch 0 from being referred to as the first element.

Status: Fixed

Update from the client: Fixed

6.17 [Info] TotalStakesScore can be updated for partial list of DLPs

File(s): DLPRootImplementation.sol

Description: The `TotalStakeScore` value for each DLP plays an important role in the computation of reward for the stake amount. Hence, all the DLPs in the `_eligibleDlpsList` should be updated in one go every time the `TotalStakeScore` values are updated. The current implementation of `saveEpochDlpsTotalStakesScore(...)` function does not enforce the updation of all eligible DLPs in one transaction.

Recommendation(s): The recommendation is to ensure all `_eligibleDlpsList` are updated with their respective `TotalStakeScore` values in one call. To do that, add validation to ensure each of the DLP in the `_eligibleDlpsList` should be present in the input array.

Status: Acknowledged

Update from the client: not fixed. it's not a problem because the offline microservice might calculate the DLP scoring asynchronous and we want to upload the scoring of each DLP as soon as the scoring is calculated. There is no dependencies between DLPs.

6.18 [Info] Unnecessary use of checkpoints

File(s): **File(s)**: DLPRootImplementation.sol

Description: The checkpoints for `_stakeWithdrawalDelayCheckpoints` and `_rewardClaimDelayCheckpoints` appear to be unnecessary given how they are used in the codebase. Only the latest entries are ever queried, no historical features of the checkpoint library are ever used. A standard `uint256` storage variable can achieve the same while reducing complexity and gas costs. Both setters for these checkpoints feature event emission, meaning that even with standard storage variables it is possible to track the delay histories using off chain logging.

Recommendation(s): Consider using a standard `uint256` storage variable instead of checkpoints for tracking the withdrawal and claim delay checkpoints.

Status: Acknowledged

Update from the client: At some point we might want to use on unstake or claim the values that were active at `stake.startData`

6.19 [Best Practices] Missing function to revoke permissions on a file in Data registry

File(s): DataRegistryImplementation.sol

Description: In the DataRegistry contract the function addFilePermission can be used to set permissions for a file. If permissions need to be removed for some reason, this function can still be used to clear permission by setting the key to an empty string, however the event emission will state that this empty permission has been "added" which is counterintuitive from a UI perspective.

```
function addFilePermission(uint256 fileId, address account, string memory key) external override whenNotPaused {
    if (_msgSender() != _files[fileId].ownerAddress) {
        revert NotFileOwner();
    }

    _files[fileId].permissions[account] = key;
    emit PermissionGranted(fileId, account);
}
```

Recommendation(s): Consider adding logic to addFilePermission to detect when permissions are being removed and emit an event related to permission removal, or alternatively a function dedicated to removing permissions can be implemented.

Status: Acknowledged

Update from the client: Not fixed, we fine with this

6.20 [Best Practices] Reward calculation logic is duplicated in view and claiming flows

File(s): DLPRootImplementation.sol

Description: DLPRootImplementation contract offers a view function for users to view the claimable rewards, namely calculateStakeClaimableAmount(...) function. The contract also offers a claiming functions to claim the rewards based on stakeId and epochIds.

While the logic for computing the rewards is same for view and claiming flow, the code was implemented separately in each of the flows, essentially duplicating the same logic twice.

Recommendation(s): As a best practice, such shared logic should be placed in an internal function to be shared across the two flows. This will ensure consistency.This helps in case some areas of the code are changes, the changes will be reflected both in the state changing and calculation functions will also improve code readability, maintainability and structure of the logic.

Status: Fixed

Update from the client: Fixed

6.21 [Best Practices] Timestamp should indicate reward claim instead of reward amount

File(s): DataLiquidityPoolImplementation.sol

Description: The function requestReward only allows rewards to be claimed once, by checking that the existing file rewardAmount is zero. If the rewardAmount is nonzero, then it means a reward has already been claimed. However, it is possible for a reward amount to be zero if the fileRewardFactor for a given DLP is zero. In this case, the function requestReward can be called any number of times, each time emitting an event and changing the reward claim timestamp. It is also possible to change the proof index used on each call. The function is shown below:

```
function requestReward(uint256 fileId, uint256 proofIndex) external override whenNotPaused nonReentrant {
    // ...

    File storage file = _files[fileId];
    if (file.rewardAmount != 0) {
        revert FileAlreadyAdded();
    }

    // ...

    file.timestamp = block.timestamp;
    file.proofIndex = proofIndex;
    file.rewardAmount = (fileRewardFactor * fileProof.data.score) / 1e18;

    // ...

    emit RewardRequested(registryFile.ownerAddress, fileId, proofIndex, file.rewardAmount);
}
```

Recommendation(s): Consider checking if a file is already added by using the `file.timestamp` instead of the `file.rewardAmount` to ensure that the function `requestReward` cannot be called multiple times while the `fileRewardFactor` is zero.

Status: Acknowledged

Update from the client: Client agreed that using `timestamp` is a better way to check if the file was already rewarded. But, client will not fix for this release, will instead consider during the next release.

6.22 [Best Practices] UnauthorizedUserAction event log does not relay complete information

File(s): DAT.sol

Description: In `DAT` token contract, while the moving the tokens between two accounts, the `from` address is checked to ensure it is not in the block list. In the case where the user is in the block list, the `UnauthorizedUserAction` revert message is used and the transaction fails.

The `UnauthorizedUserAction` event only records the caller address, but does not record the `from` address which was blocked. The event does not contain complete information of why the transaction was reverted.

In the case of a `transferFrom` call, where the caller is transferring assets from a blocked `from` address to some `to` address, the revert error message will use msg.sender which is the caller address, not the blocked `from` address which may be misleading.

```
 modifier whenNotBlocked(address from) {
        if (_blockList.contains(from)) {
            revert UnauthorizedUserAction(msg.sender);
        }
        _;
    }
```

Recommendation(s): Consider passing the `from` address in addition to the caller for the revert message.

```
 // declaration
 error UnauthorizedUserAction(address account, address from);

// usage
 revert UnauthorizedUserAction(msg.sender,from);
```

Status: Fixed

Update from the client: Fixed

6.23 [Best Practices] Unnecessary payable operator in `addProof`

File(s): `TeePoolImplementation.sol`

Description: The TEE generates and submits the proof for the file associated with the job id. Since the TEE will receive the fee for attestation of the file, there is no need to have payable operator for `addProof` function.

```
function addProof(uint256 jobId, IDataRegistry.Proof memory proof) external payable override onlyActiveTee {
    //...
}
```

Recommendation(s): Remove payable operator from the function signature.

Status: Fixed

Update from the client: Fixed

6.24 [Best Practices] Unused `MAINTAINER` role in `DataLiquidityPool`

File(s): `DataLiquidityPoolImplementation.sol`

Description: The role `MAINTAINER` is not used in the DLP contract, usually the maintainer role is assigned to handle certain operations specific to the contract such as pausing. But in the case of DLP these abilities are given to the admin instead.

This breaks the common convention of the maintainer role handling contract specific management functions, which is consistently applied in all other contract implementations (TeePool, DataRegistry, DAT, DLPRoot)

Recommendation(s): Consider adjusting the role required for relevant functions in the `DataLiquidityPoolImplementation` file to use the maintainer role instead of the admin role.

Status: Acknowledged

Update from the client: Not fixed, we fine with this

6.25 [Best Practices] Unused imports

File(s): `contracts/*`

Description: Quick rough writing, can clean up later:

```
DataRegistryImplementation
    ECDSA import unused
    MessageHashUtils import unused

TeePoolImplementation
    PausableUpgradeable import but modifiers not used
```

Recommendation(s):

Status: Fixed

Update from the client: Fixed

7 Findings (Follow-up Metrics Review)

7.1 [Info] `estimatedDlpRewardPercentages` does not consider non-top DLPs

File(s): `DLPRootMetricsImplementation.sol`

Description: The DLP root metrics contract contains the function `estimatedDlpRewardPercentages` which determines the amount of rewards that some given set of DLPs should receive, for the latest epoch on the DLP root contract. To estimate rewards, the function first gets all top performing DLPs and calculates their total stake amount and ratings, before applying a pro-rata distribution of rewards based on the individual stake amounts and ratings of the DLPs provided in the `dlpIds` argument array.

When DLP IDs provided in the `dlpIds` argument array are all considered part of the "top DLPs" in the root contract then estimations will be correct. Rewards should only be given to top performing DLPs, however the estimate function does not check whether a provided DLP ID is in the `topDlpsList`.

This makes it possible to provide a non-top-performing DLP ID to the function and it will calculate a pro-rata reward based on the staked amount, even though it isn't a top performing DLP and therefore should not be eligible for rewards.

Recommendation(s): This function is only used by `DLPRoot.estimatedDlpRewardPercentages` which is an external view function, and won't have any on-chain impact. However, it should be noted that any off-chain entities that interact with this function may receive incorrect reward estimations for non-top-performing DLPs. This can be addressed with a check for each DLP in the `dlpIds` argument array and for non-top-performers their rewards can default to zero.

Status: Acknowledged

Update from the client: We are aware of this behavior. The `estimatedDlpRewardPercentages` function is designed to estimate the APY even for DLPs that are not currently eligible. The intention is to provide an approximation of the APY for cases where a DLP may enter the top *N* DLPs in the current epoch.

7.2 [Best Practices] Unnecessary `receive` function in `DLPRootImplementation`

File(s): `DLPRootImplementation.sol`

Description: With the addition of the stakes and reward treasuries for the `DLPRootImplementation` contract, funds can be sent to the treasuries directly, an all funds related to staking can be passed through the payable functions `createStake` and `registerDlp`. As a result, the `receive` function in the root contract is no longer necessary.

Recommendation(s): Consider removing the `receive` function in the `DLPRootImplementation` contract.

Status: Fixed

Update from the client: Fixed

8 Documentation Evaluation

Software documentation refers to the written or visual information describing software's functionality, architecture, design, and implementation. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about Braavos Wallet documentation

The documentation for the Vana Smart Contracts are contained in the README of the project's Github. It is written for developers, presenting each core component of the protocol, including deployment, contracts, functions, and testing.
The information is presented in a very concise and technical manner, with well-written explanations and references where necessary. It also features a section explaining development dependencies and instructions on how to build and test the code.
Code comments are also of high quality, with explanations for every function describing the purpose, context, and situations where the function will be called. Other than functions, some comments exist in other areas of the code where extra information is necessary, allowing readers to understand the codebase at a faster pace.

9 Test Suite Evaluation

9.1 Compilation Output

```
user@machine:~/NM-0383/vana-smart-contracts$ npx hardhat compile
Generating typings for: 94 artifacts in dir: typechain-types for target: ethers-v6
Successfully generated 280 typings!
Compiled 92 Solidity files successfully (evm target: paris).
```

9.2 Tests Output

```
npx hardhat test

  DataRegistry
    Setup
       should have correct params after deploy
       should change admin
       Should upgradeTo when owner
       Should upgradeTo when owner and emit event
       Should reject upgradeTo when storage layout is incompatible
       Should reject upgradeTo when non owner
    AddFile
       should addFile
       should addFile multiple times
       should reject addFiles with used fileUrl
       should reject addFile when paused
    Proof
       should addProof, one file, one tee
       should addProof, one file, multiple tee
       should addProof, multiple files, one tee
       should addProof, multiple files, multiple tees
       should reject addProof when paused
    FilePermission
       should addFilePermission, one file, one dlp
       should addFilePermission, one file, multiple dlps #1
       should addFilePermission, one file, multiple dlps #2
       should addFilePermission, multiple files, one dlp
       should addFilePermission, multiple files, multiple dlps
       should reject addFilePermission when non-owner
       should reject addFilePermission when paused
    AddFileWithPermissions
       should addFileWithPermissions, one file, one dlp
       should addFilePermission, one file, multiple dlps #1
       should addFilePermission, one file, multiple dlps #2
       should addFilePermission, multiple files, one dlp
       should addFilePermission, multiple files, multiple dlps
       should reject addFilePermission when non-owner
       should reject addFilePermission when paused

  Deposit
    Setup
       should have correct params after deploy
       should updateMinDepositAmount when owner
       should reject updateMinDepositAmount when non-owner
       should updateMaxDepositAmount when owner
       should reject updateMaxDepositAmount when non-owner
       should updateRestricted when owner
       should reject updateRestricted when non-owner
       Should transferOwnership in 2 steps
       Should reject transferOwnership when non-owner
       Should reject acceptOwnership when non-newOwner
       Should upgradeTo when owner
       Should upgradeTo when owner and emit event
       Should reject upgradeTo when storage layout is incompatible
       Should reject upgradeTo when non owner
```

```
    Validators
      should addAllowedValidators when owner
      should reject addAllowedValidators when non-owner
      should addAllowedValidators when owner
      should removeAllowedValidators when owner
      should reject removeAllowedValidators when non-owner
    Deposit
      should deposit when allowed validator #1
      should deposit when allowed validator #2
      should reject deposit when non-allowed validator
      should reject deposit when permission was removed
      should reject deposit when already deposited
      should deposit when restricted = false
  Deposit2
    Setup
      should have correct params after deploy
      should updateMinDepositAmount when owner
      should reject updateMinDepositAmount when non-owner
      should updateMaxDepositAmount when owner
      should reject updateMaxDepositAmount when non-owner
      should updateRestricted when owner
      should reject updateRestricted when non-owner
      Should transferOwnership in 2 steps
      Should reject transferOwnership when non-owner
      Should reject acceptOwnership when non-newOwner
      Should upgradeTo when owner
      Should upgradeTo when owner and emit event
      Should reject upgradeTo when storage layout is incompatible
      Should reject upgradeTo when non owner
    Validators
      should addAllowedValidators when owner
      should reject addAllowedValidators when non-owner
      should addAllowedValidators when owner
      should removeAllowedValidators when owner
      should reject removeAllowedValidators when non-owner
    Deposit
      should deposit when allowed validator #1
      should deposit when allowed validator #2
      should reject deposit when non-allowed validator
      should reject deposit when permission was removed
      should reject deposit when already deposited
      should deposit when restricted = false

  DataLiquidityPool
    Setup
      should have correct params after deploy
      Should pause when owner
      Should reject pause when non-owner
      Should unpause when owner
      Should reject unpause when non-owner
      Should updateFileRewardFactor when owner
      Should reject updateFileRewardFactor when non-owner
      Should updateTeePool when owner
      Should reject updateFileRewardFactor when non-owner
      Should updatePublicKey when owner
      Should reject updatePublicKey when non-owner
      Should updateProofInstruction when owner
      Should reject updateProofInstruction when non-owner
      should change admin
      Should upgradeTo when owner
      Should upgradeTo when owner and emit event
      Should reject upgradeTo when storage layout is incompatible
      Should reject upgradeTo when non owner
    RequestProof
      should requestReward #1
      should requestReward #2
```

```
Multisend
  Setup
     should have correct params after deploy
     Should transferOwnership in 2 steps
     Should reject transferOwnership when non-owner
     Should reject acceptOwnership when non-newOwner
     Should upgradeTo when owner
     Should upgradeTo when owner and emit event
     Should reject upgradeTo when storage layout is incompatible
     Should reject upgradeTo when non owner
  MultisendToken
     should multisendToken to 2 users
     should multisendToken to 500 users
     should reject multisendToken when not enough allowance
     should reject multisendToken when not enough balance
  MultisendVana
     should multisendVana to 2 users
     should multisendVana to 500 users
     should reject multisendVana when invalid amount
  MultisendVanaWithDifferentAmounts
     should multisendVanaWithDifferentAmounts to 2 users, same amount
     should multisendVanaWithDifferentAmounts to 2 users, different amount
     should reject multisendVanaWithDifferentAmounts when not enough funds
     should multisendVanaWithDifferentAmounts to 500 users
     should reject when amounts and recipients arrays have different lengths
  MultisendTokenWithDifferentAmounts
     should multisendTokenWithDifferentAmounts to 2 users with different amounts
     should multisendTokenWithDifferentAmounts to 500 users with random amounts
     should reject when amounts and recipients arrays have different lengths
     should reject when not enough allowance
     should reject when not enough balance

DLPRoot
  Setup
     should have correct params after deploy
     should pause when maintainer
     should reject pause when non-maintainer
     should unpause when maintainer
     should reject unpause when non-maintainer
     should updateEligibleDlpsLimit when maintainer
     should reject updateEligibleDlpsLimit when non-maintainer
     should updateEpochDlpsLimit when maintainer
     should reject updateEpochDlpsLimit when non-maintainer
     should updateEpochSize when maintainer
     should reject updateEpochSize when non-maintainer
     should updateEpochRewardAmount when maintainer
     should reject updateEpochSize when non-maintainer
     should updateMinStakeAmount when maintainer
     should reject updateMinStakeAmount when non-maintainer
     should updateMinDlpStakersPercentage when maintainer
     should reject updateMinDlpStakersPercentage when non-maintainer
     should updateMinDlpRegistrationStake when maintainer
     should reject updateMinDlpRegistrationStake when non-maintainer
     should updateDlpEligibilityThreshold when maintainer
     should reject updateDlpEligibilityThreshold when non-maintainer
     should updateDlpSubEligibilityThreshold when maintainer
     should reject updateDlpSubEligibilityThreshold when non-maintainer
     should updateStakeWithdrawalDelay when maintainer
     should reject updateStakeWithdrawalDelay when non-maintainer
     should change admin
     should upgradeTo when owner
     should upgradeTo when owner and emit event
     should reject upgradeTo when storage layout is incompatible
     should reject upgradeTo when non owner
  Dlps - registration
     should registerDlp when stake <  dlpEligibilityThreshold
     should registerDlp when stake = dlpEligibilityThreshold
     should change eligibility after staking and unstaking
     should registerDlp after epoch1.startBlock
     should registerDlp and add stake
     should registerDlp multiple times
     should reject registerDlp when paused
     should reject registerDlp when stake amount too small
     should reject registerDlp when stakersPercentage too small
```

```
      should reject registerDlp when stakersPercentage too big
      should reject registerDlp when already registered
      should reject registerDlp when deregistered
      should reject registerDlp when name already taken
      should reject registerDlp with empty name
      should deregisterDlp when dlp owner
      should reject deregisterDlp when non dlp owner
      should reject deregisterDlp when deregistered
      should updateDlp when dlp owner
      should updateDlp when dlp owner
      should reject updateDlp when not dlp owner
      should reject updateDlp when owner address is zero
      should reject updateDlp when treasury address is zero
      should reject updateDlp when stakers percentage below minimum
      should reject updateDlp when stakers percentage above 100%
      should reject updateDlp when trying to change DLP address
      should updateDlp and update stakersPercentage in next epoch
      should reject updateDlp when paused
   Update DLP sub-eligibility threshold
      should updateDlpSubEligibilityThreshold when maintainer
      should reject updateDlpSubEligibilityThreshold when non-maintainer
      should update DLP status from eligible to registered when below new threshold
      should update multiple DLP statuses when updateDlpSubEligibilityThreshold
      should not affect deregistered DLPs when updateDlpSubEligibilityThreshold
      should not affect DLPs above eligibility threshold when updateDlpSubEligibilityThreshold
      should handle empty eligible DLPs list when updateDlpSubEligibilityThreshold
   Update DLP eligibility threshold
      should updateDlpEligibilityThreshold when maintainer
      should reject updateDlpEligibilityThreshold when non-maintainer
      should updateDlpEligibilityThreshold and update DLP status from eligible to sub-eligible when below new threshold
      should updateDlpEligibilityThreshold and maintain DLP eligibility list
      should updateDlpEligibilityThreshold and not affect deregistered DLPs
      should updateDlpEligibilityThreshold with multiple stakes
      should handle empty eligible DLPs list when updating threshold
      should handle updating threshold to same value
   Epochs
      should createEpochs after the end of the previous one
      should createEpochs after updating rewardAmount
      should createEpochs after updating epochSize
      should createEpochs after long time
      should createEpochsUntilBlockNumber after long time
      should createEpochsUntilBlockNumber with limit
      should createEpochsUntilBlockNumber just until current block number
      should create epochs with no active dlps
      should createEpochs with one registered dlp #1
      should createEpochs after dlpStakersPercentage changes
      should createEpochs with multiple registered dlps #1
      should createEpochs with multiple registered dlps #2
      should createEpochs with multiple registered dlps #3
      should createEpochs with multiple registered dlps #4
      should createEpochs with multiple registered dlps #5
      should createEpochs after staking
      should createEpochs when 100 dlps and 16  epochDlpsLimit
    - should createEpochs when 1000 dlps and 32  epochDlpsLimit
      should overrideEpoch when maintainer
      should revert overrideEpoch when not maintainer
   Staking
      should createStake and emit event
      should create missing epochs when createStake
      should reject createStake when dlp doesn't exist
      should reject createStake when dlp is deregistered
      should reject createStake when stakeAmount < minStakeAmount
      should createStake multiple times, one dlp
      should createStake multiple times, multiple dlps
      should createStake and set lastClaimedIndexEpochId after many epochs
   Close stake
      should closeStake and emit event
      should closeStake multiple stakes in one call
      should closeStake multiple stakes
      should create missing epochs when closeStake
      should reject closeStake when not stake owner
      should reject closeStake when already closed
      should reject closeStake when invalid stake
```

```
    should closeStake and update dlp status (eligible -> subEligible)
    should closeStake and update dlp status (eligible -> registered)
    should closeStake and update dlp status (subEligible -> registered)
    should closeStake and keep dlp status (eligible)
    should closeStake and keep dlp status (subEligible)
  Withdraw stake
    should withdrawStake after delay period
    should withdraw multiple stakes in one call
    should create missing epochs when withdrawStake
    should reject withdrawStake when not stake owner
    should reject withdrawStake when already withdrawn
    should reject withdrawStake when not closed
    should reject withdrawStake when withdrawal delay not passed
    should withdraw stake after delay update
  TopDlps
    should set topDlps when creating new epoch (dlpsCount = 0,  epochDlpsLimit = 16)
    should set topDlps when creating new epoch (dlpsCount = 1,  epochDlpsLimit = 1)
    should set topDlps when creating new epoch (dlpsCount = 2,  epochDlpsLimit = 2)
    should set topDlps when creating new epoch (dlpsCount = 3,  epochDlpsLimit = 3)
    should set topDlps when creating new epoch (dlpsCount = 16,  epochDlpsLimit = 16)
    should set topDlps when creating new epoch (dlpsCount = 32,  epochDlpsLimit = 32)
    should set topDlps when creating new epoch (dlpsCount = 2,  epochDlpsLimit = 1)
    should set topDlps when creating new epoch (dlpsCount = 3,  epochDlpsLimit = 1)
    should set topDlps when creating new epoch (dlpsCount = 16,  epochDlpsLimit = 1)
    should set topDlps when creating new epoch (dlpsCount = 32,  epochDlpsLimit = 1)
    should set topDlps when creating new epoch (dlpsCount = 1,  epochDlpsLimit = 16)
    should set topDlps when creating new epoch (dlpsCount = 2,  epochDlpsLimit = 16)
    should set topDlps when creating new epoch (dlpsCount = 3,  epochDlpsLimit = 16)
    should set topDlps when creating new epoch (dlpsCount = 16,  epochDlpsLimit = 16)
    should set topDlps when creating new epoch (dlpsCount = 30,  epochDlpsLimit = 16)
    should set topDlps when creating new epoch (dlpsCount = 40,  epochDlpsLimit = 16)
    should set topDlps when creating new epoch (dlpsCount = 50,  epochDlpsLimit = 16)
    should set topDlps when creating new epoch (dlpsCount = 60,  epochDlpsLimit = 16)
    should set topDlps when creating new epoch (dlpsCount = 100,  epochDlpsLimit = 16)
    should set topDlps when creating new epoch (dlpsCount = 200,  epochDlpsLimit = 16)
    should set topDlps when creating new epoch (dlpsCount = 300,  epochDlpsLimit = 16)
    should set topDlps when creating new epoch (dlpsCount = 1000,  epochDlpsLimit = 16)
    should set topDlps when creating new epoch after dlpOwner staking
    should set topDlps when creating new epoch after user staking
    should set topDlps when creating new epoch after unstaking
    should set topDlps when creating new epoch after unstaking #2
    should set topDlps when creating new epoch after registering new DLPs
    should set topDlps when creating new epoch after a DLP deregisters
    should set topDlps when creating new epoch after updating the maximum number of DLPs #updateEpochDlpsLimit
   should set topDlps when creating new epoch #staking, unstaking, registration, deregistration, updateEpochDlpsLimit
  Save epoch DLPs total stakes score
    should saveEpochDlpsTotalStakesScore and emit event
    should reject saveEpochDlpsTotalStakesScore when non-manager
    should reject saveEpochDlpsTotalStakesScore for unregistered dlpId
    should reject saveEpochDlpsTotalStakesScore for future epochs
    should reject saveEpochDlpsTotalStakesScore when score already exists
    should saveEpochDlpsTotalStakesScore for multiple valid scores
    should reject saveEpochDlpsTotalStakesScore when any score in batch is invalid
    should saveEpochDlpsTotalStakesScore for deregistered DLPs past epochs
    should overrideEpochDlpsTotalStakesScore for new score and emit event
    should reject overrideEpochDlpsTotalStakesScore when called by non-maintainer
    should reject overrideEpochDlpsTotalStakesScore for unregistered DLP ID
    should reject overrideEpochDlpsTotalStakesScore for future epochs
    should overrideEpochDlpsTotalStakesScore for existing score and emit event
    should overrideEpochDlpsTotalStakesScore with same value and emit event
    should overrideEpochDlpsTotalStakesScore for zero value and emit event
    should overrideEpochDlpsTotalStakesScore for deregistered DLPs past epochs
  Calculate stake score
    should calculateStakeScore return correct values for 0-65 days
    should calculateStakeScore same block (no multiplier)
    should calculateStakeScore for less than one day
    should calculateStakeScore for exactly one day
    should calculateStakeScore for one week
    should calculateStakeScore for one month
    should calculateStakeScore for maximum multiplier
    should calculateStakeScore with fractional days
    should calculateStakeScore with zero stake amount
    should calculateStakeScore with small stake amounts
```

```
    Claim stakes reward - rewardClaimDelay = 0
      should claimStakesReward
      should reject claimStakesReward when paused
  TeePool
    Setup
      should have correct params after deploy
      should change admin
      Should updateDataRegistry when owner
      Should reject updateDataRegistry when non-owner
      Should updateTeeFee when owner
      Should reject updateTeeFee when non-owner
      Should updateCancelDelay when owner
      Should reject updateCancelDelay when non-owner
      Should multicall
      Should upgradeTo when owner
      Should upgradeTo when owner and emit event
      Should reject upgradeTo when storage layout is incompatible
      Should reject upgradeTo when non owner
    Tee management
      should addTee when owner
      should addTee #multiple tees
      should reject addTee when already added
      should reject addTee when non-owner
      should removeTee when owner #1
      should removeTee when multiple tees
      should reject removeTee when non-owner
      should reject removeTee when not added
    Job
      should requestContributionProof
      should submitJob
      should requestContributionProof #same user multiple files
      should requestContributionProof for same file #multiple users same file
      should requestContributionProof #multiple users multiple files
      should requestContributionProof without bid when teeFee = 0
      should reject requestContributionProof when insufficient fee
      should cancelJob with bid when teeFee != 0
      should cancelJob when multiple jobs #1
      should cancelJob when multiple jobs #2
      should cancelJob without bid when teeFee = 0
      should reject cancelJob before cancelDelay
      should reject cancelJob when not job owner
    Proof
      should addProof when assigned tee #1
      should addProof when assigned tee #2
      should reject addProof when not tee
      should reject addProof when not active tee
      should reject addProof when proof already submitted
      should addProof for multiple files
      Claim
        should claim
        should reject withdraw when not tee
        should reject withdraw when nothing to claim
        should reject claim when already claimed
        should claim multiple times
    End to End
      example 1

  ERC20
    DLPT - basic
      should have correct params after deploy
      Should transferOwnership in 2 steps
      Should reject transferOwnership when non-owner
      Should changeAdmin when owner
      Should reject changeAdmin when non-owner
      Should blockMint when owner
      Should reject blockMint when non-owner
      Should mint when owner
      Should reject mint when non-owner
      Should reject mint when minting is blocked
      Should blockAddress when admin
      Should reject blockAddress when non-admin
      Should unblockAddress when admin #1
```

```
      Should reject unblockAddress when non-admin
      Should unblockAddress when admin #2
      Should transfer
      Should reject transfer when blocked
      Should transfer when unblocked
    DLPT - voting
      should delegate
      should have 0 votes when blocked
      should reject delegate when blocked
      should cancel delegate when blocked

  Treasury
    Setup
      should have correct params after deploy
      Should transferOwnership in 2 steps
      Should reject transferOwnership when non-owner
      Should reject acceptOwnership when non-newOwner
      Should upgradeTo when owner
      Should upgradeTo when owner and emit event
      Should reject upgradeTo when storage layout is incompatible
      Should reject upgradeTo when non owner
    Receive
      should receive VANA
    Withdraw
      should withdraw token when owner
      should not withdraw token when non owner
      should withdraw VANA when owner
      should not withdraw VANA when non owner
  386 passing (1m)
  1 pending
```

10 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development procehttps://www.overleaf.com/project/65c0e737f41a29601bda5c48ss, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer