hashlock.

Security Audit

Vana 3rd (DeFi)

July 2025 info@hashlock.com.au Hashlock Pty Ltd

Table of Contents

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE WHICH COULD BE USED TO COMPROMISE THE PROJECT. THIS DOCUMENT SHOULD ONLY BE FOR INTERNAL USE UNTIL ISSUES ARE RESOLVED. ONCE VULNERABILITIES ARE REMEDIATED, THIS REPORT CAN BE MADE PUBLIC. THE CONTENT OF THIS REPORT IS OWNED BY HASHLOCK PTY LTD FOR USE OF THE CLIENT.

Executive Summary

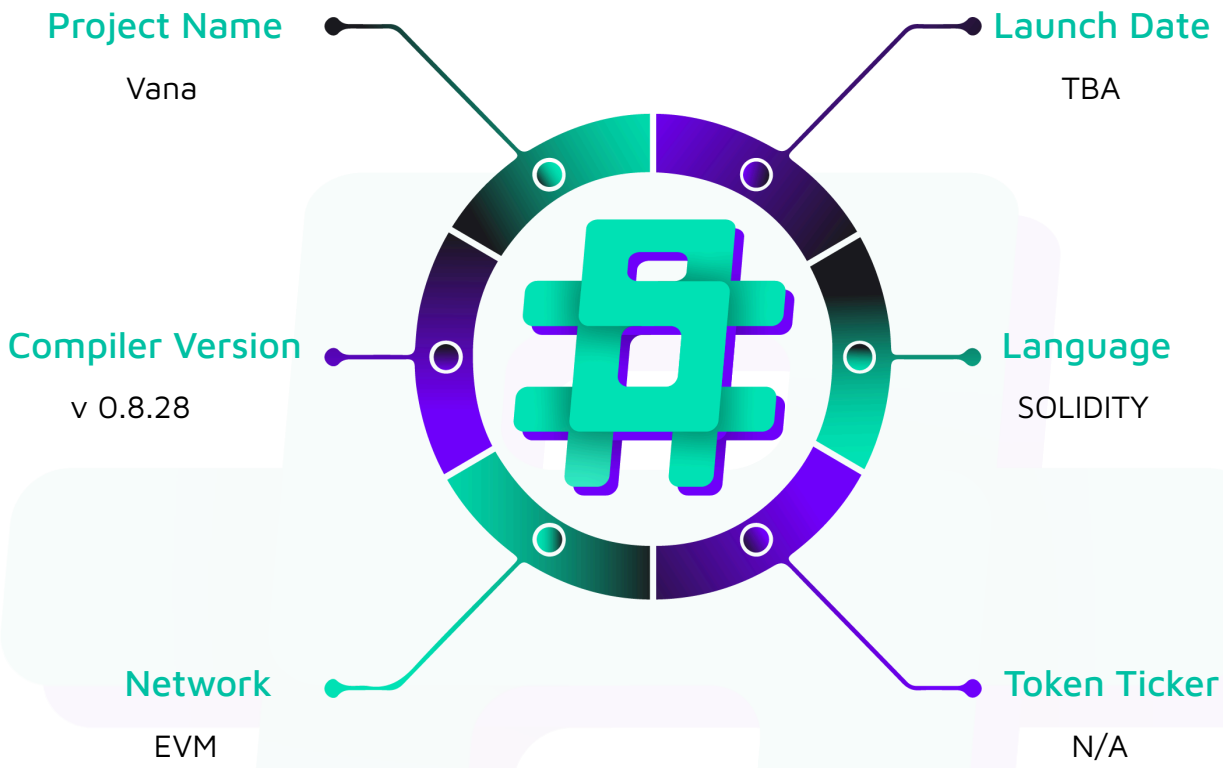
The Vana team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

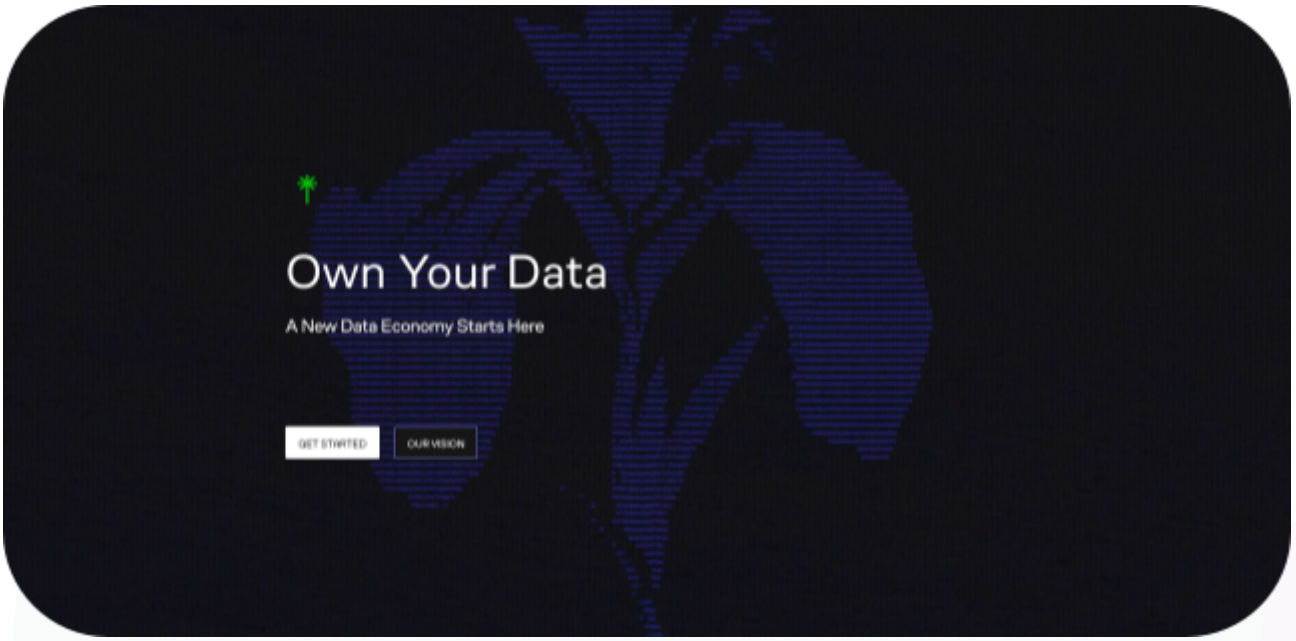
Vana is pioneering a transformative approach to data ownership and social impact by harnessing the power of Web3 and decentralized finance (DeFi). Through a groundbreaking ecosystem, individuals gain full sovereignty over their digital identities, turning data into an asset they truly own and control. By integrating DeFi principles, Vana enables users to monetize their data contributions via impact tokens, unlocking liquidity and new revenue streams within a decentralized marketplace. Unique NFT-backed data assets empower contributors with secure, tradable ownership, while DAO governance ensures the community collectively shapes the platform's evolution. Transparent smart contracts and a robust data commons framework create a circular economy where value flows seamlessly between personal gain and global impact. With Vana, your data becomes a catalyst for innovation, financial inclusion, and ethical transformation, redefining how individuals, communities, and economies thrive in the Web3 era

Project Name: Vana
Project Type: DeFi
Compiler Version: 0.8.28
Website: https://www.vana.org/
Logo:

Visualised Context:



Project Visuals:



Audit Scope

We at Hashlock audited the solidity code within the Vana project, the scope of work included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

Description	Vana Smart Contracts
Platform	EVM / Solidity
Audit Date	July, 2025
Contract 1	QueryEngineImplementation.sol
Contract 2	DLPRegistryImplementation.sol
Contract 3	VanaEpochImplementation.sol
Contract 4	VanaEpochStorageV1.sol
Contract 5	DLPPerformanceImplementation.sol
Contract 6	QueryEngineStorageV1.sol
Contract 7	DLPRewardDeployerImplementation.sol
Contract 8	DLPPerformanceStorageV1.sol
Audited GitHub Commit Hash 1 (PR 26)	c8c62254443149e434e4278dc2ff244ba5a81a7f
Audited GitHub Commit Hash 2 (PR 27)	d09f55f7e395f67648af35aa8e495c8dcaec018e
Audited GitHub Commit Hash 3 (PR 29)	c38f6311fe354f84bcedd2a6d851da4f36de13b3
Audited GitHub Commit Hash 4 (PR 31)	d85ef19db015803d5fd7b4261bbbabcda046e494
Audited GitHub Commit Hash 5 (PR 32)	2a141307fd559adb85946688a7636c8daafa17a3
Fix Review GitHub Commit Hash	b5d7b6c1d90ca2ce12d5ee7a8da13cab28102cc2

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Secure"**. The contracts all follow simple logic, with correct and detailed ordering. They use a series of interfaces, and the protocol uses a list of Open Zeppelin contracts.

The 'Hashlocked' rating is reserved for projects that ensure ongoing security via bug bounty programs or on chain monitoring technology.

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section. The list of audited assets is presented in the Audit Scope section and the project's contract functionality is presented in the Intended Smart Contract Functions section.

All vulnerabilities initially identified have now been resolved and acknowledged.

Hashlock found:

3 Medium severity vulnerabilities

4 Low severity vulnerabilities

2 Gas Optimisations

1 QA

Caution: *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

Intended Smart Contract Functions

Claimed Behaviour	Actual Behaviour
QueryEngineImplementation.sol - Allows users to: - Create/manage data access permissions with custom pricing - Claim accumulated DLP payments from the data access fees - Allows query services to: - Request payments in VANA or whitelisted tokens for computational jobs - Auto-split fees between DLP and Vana treasuries - Allows admins to: - Add/remove payment tokens - Update contract addresses and fee percentages - Pause/unpause operations - Upgrade implementation	**Contract achieves this functionality.**
DLPRegistryImplementation.sol - Allows users to: - Register new DLPs with a deposit payment - Update DLP information - Deregister their DLPs - Allows maintainers to: - Update DLP verification status and token information - Set registration deposit amounts - Migrate DLP data from other contracts - Pause/unpause operations - Allows the system to: - Track DLP eligibility based on verification, token, and LP token status	**Contract achieves this functionality.**

- Maintain an eligible DLPs list for the reward distribution - Enforce epoch finalization requirements for status changes	
VanaEpochImplementation.sol - Allows the system to: - Automatically create new epochs based on block progression - Track the epoch lifecycle - Store DLP rewards and penalties for each epoch - Manage the epoch finalization process - Allows DLP Performance contracts to: - Save calculated DLP rewards for completed epochs - Override DLP rewards in finalized epochs for corrections - Allows maintainers to: - Update epoch parameters - Force finalize epochs manually - Update core contract addresses - Manually update epoch data and counts - Pause/unpause operations	**Contract achieves this functionality.**
VanaEpochStorageV1.sol - A storage contract that defines: - Epoch configuration parameters - Epoch tracking state - Core contract references - Provides storage for: - Epoch reward distribution amounts - Block-based timing parameters for epoch cycles - Internal epoch data structures and mappings - Interface implementations for the epoch management system	**Contract achieves this functionality.**
DLPPerformanceImplementation.sol - Allows managers to:	**Contract achieves this functionality.**

- Save DLP performance metrics for epochs - Calculate normalized scores for each metric across all eligible DLPs - Allows maintainers to: - Update metric weights for reward calculations - Override performance data for finalized epochs - Set penalty multipliers for specific DLPs and metrics - Finalize epoch scores and trigger reward distribution - Update core contract addresses - Pause/unpause operations - Allows the system to: - Calculate weighted rewards and penalties for each DLP is based on performance scores - Track performance history across epochs with configurable metric weights - Enforce epoch lifecycle rules	
QueryEngineStorageV1.sol - A storage contract that defines: - Permission management state - Payment system configuration - Core contract references - Token management - Provides storage for: - Data access permissions with granular control per refiner and grantee - Payment distribution logic between DLP and Vana treasuries - DLP public keys for encryption/verification - Internal mappings for permission tracking and token whitelist management	**Contract achieves this functionality.**
DLPRewardDeployerImplementation.sol	**Contract achieves**

- Allows reward deployers to: - Distribute DLP rewards in scheduled tranches across multiple epochs - Execute token swaps and send rewards to DLP treasuries - Allows maintainers to: - Initialize epoch reward parameters - Withdraw undistributed penalty amounts to specified recipients - Update reward distribution settings - Update core contract addresses - Pause/unpause operations - Allows the system to: - Track distributed amounts and tranche history per DLP per epoch - Enforce time-based distribution intervals between tranches - Automatically handle reward splitting between VANA and DLP tokens via swap mechanisms	**this functionality.**
DLPPerformanceStorageV1.sol - Provides storage for: - Performance metric weights and calculations - Epoch-based performance data structures - Internal performance mappings and evaluations	**Contract achieves this functionality.**

Code Quality

This audit scope involves the smart contracts of the Vana project, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices and to help avoid unnecessary complexity that increases the likelihood of exploitation; however, some refactoring was recommended to optimize security measures.

The code is very well commented on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the Vana project smart contract code in the form of GitHub access.

As mentioned above, code parts are well commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments are helpful in providing an understanding of the protocol's overall architecture.

Dependencies

As per our observation, the libraries used in this smart contracts infrastructure are based on well-known industry standard open source projects.
Apart from libraries, its functions are used in external smart contract calls.

Severity Definitions

The severity levels assigned to findings represent a comprehensive evaluation of both their potential impact and the likelihood of occurrence within the system. These categorizations are established based on Hashlock's professional standards and expertise, incorporating both industry best practices and our discretion as security auditors. This ensures a tailored assessment that reflects the specific context and risk profile of each finding.

Significance	Description
High	High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium-level difficulties should be solved before deployment, but won't result in loss of funds.
Low	Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future.
Gas	Gas Optimisations, issues, and inefficiencies.
QA	Quality Assurance (QA) findings are informational and don't impact functionality. Supports clients improve the clarity, maintainability, or overall structure of the code.

Status Definitions

Each identified security finding is assigned a status that reflects its current stage of remediation or acknowledgment. The status provides clarity on the handling of the issue and ensures transparency in the auditing process. The statuses are as follows:

Significance	Description
Resolved	The identified vulnerability has been fully mitigated either through the implementation of the recommended solution proposed by Hashlock or through an alternative client-provided solution that demonstrably addresses the issue.
Acknowledged	The client has formally recognized the vulnerability but has chosen not to address it due to the high cost or complexity of remediation. This status is acceptable for medium and low-severity findings after internal review and agreement. However, all high-severity findings must be resolved without exception.
Unresolved	The finding remains neither remediated nor formally acknowledged by the client, leaving the vulnerability unaddressed.

Audit Findings

Medium

[M-01] DLPPerformanceImplementation#saveEpochPerformances - Missing DLP ID Uniqueness Validation

Description

The `saveEpochPerformances` function in the `DLPPerformanceImplementation` contract lacks validation to ensure that each `dlpId` appears only once in the input array. The function only validates the array length against the number of eligible DLPs but does not check for duplicate entries, allowing the same DLP to be processed multiple times within a single transaction.

Vulnerability Details

In this function, the only checks performed are for array length equality and whether the given ID is contained in the `_eligibleDlpsList` list.

An array containing duplicates can be passed either by a compromised manager or by malware. It can also happen accidentally (human factor)

```
function saveEpochPerformances(
    uint256 epochId,
    EpochDlpPerformanceInput[] calldata newEpochDlpPerformances
) external override onlyRole(MANAGER_ROLE) whenNotPaused {
    vanaEpoch.createEpochs();

    IVanaEpoch.EpochInfo memory epoch = vanaEpoch.epochs(epochId);

    if (epoch.isFinalized) {
        revert EpochAlreadyFinalized();
    }

    uint256 newEpochDlpPerformancesCount = newEpochDlpPerformances.length;
```

```
        if (newEpochDlpPerformancesCount != dlpRegistry.eligibleDlpsListCount()) {
            revert InvalidEpochDlpPerformancesCount();
        }

        uint256 tradingVolumeTotalScore;
        uint256 uniqueContributorsTotalScore;
        uint256 dataAccessFeesTotalScore;

        for (uint256 i = 0; i < newEpochDlpPerformancesCount; ) {
            if (dlpRegistry.isEligibleDlp(newEpochDlpPerformances[i].dlpId) == false) {
                revert DlpNotEligible(newEpochDlpPerformances[i].dlpId);
            }

                            EpochDlpPerformanceInput  calldata  newEpochDlpPerformance  =
newEpochDlpPerformances[i];

                                    EpochDlpPerformance  storage  epochDlpPerformance  =
_epochPerformances[epochId].epochDlpPerformances[
                newEpochDlpPerformance.dlpId
            ];

            epochDlpPerformance.dataAccessFees = newEpochDlpPerformance.dataAccessFees;
            epochDlpPerformance.tradingVolume = newEpochDlpPerformance.tradingVolume;
                                                epochDlpPerformance.uniqueContributors   =
newEpochDlpPerformance.uniqueContributors;
                                                epochDlpPerformance.tradingVolumeScore   =
newEpochDlpPerformance.tradingVolumeScore;
                                            epochDlpPerformance.uniqueContributorsScore   =
newEpochDlpPerformance.uniqueContributorsScore;
                                                epochDlpPerformance.dataAccessFeesScore   =
newEpochDlpPerformance.dataAccessFeesScore;

            tradingVolumeTotalScore += newEpochDlpPerformance.tradingVolumeScore;
                                                            uniqueContributorsTotalScore    +=
newEpochDlpPerformance.uniqueContributorsScore;
            dataAccessFeesTotalScore += newEpochDlpPerformance.dataAccessFeesScore;

            //...code
}
```

Impact

DLP, which is missing from the input data, receives zero scores, while duplicate IDs receive scores that are significantly higher than expected.

Recommendation

It is recommended to add a check for a unique ID, which will prevent duplication.

Status

Resolved

[M-02] DLPRegistryImplementation/QueryEngineImplementation - Frontrunning due to uninitialized variables

Description

The variables `dlpRegistrationDepositAmount` in the `DLPRegistryImplementation` contract and `dlpPaymentPercentage` in the `QueryEngineImplementation` contract are not initialized during deployment and remain at their default value of `0`.

Vulnerability Details

This allows users to register DLP without paying the required deposit until the `MAINTAINER` manually sets the deposit amount using the `updateDlpRegistrationDepositAmount` function. The same applies to the QueryEngineImplementation contract. Until `MAINTAINER` sets the `dlpPaymentPercentage` variable via the `updateDlpPaymentPercentage` function, the internal `_requestPayment` function will execute a scenario in which `dlpPaymentAmount` is equal to 0, and `queryEngineTreasury` will not receive any payments.

```
//DLPRegistryImplementation
   function initialize(address ownerAddress) external initializer {
        __AccessControl_init();
        __UUPSUpgradeable_init();
        __ReentrancyGuard_init();
        __Pausable_init();

        _grantRole(DEFAULT_ADMIN_ROLE, ownerAddress);
        _grantRole(MAINTAINER_ROLE, ownerAddress);
    }

    //...code
    function _registerDlp(DlpRegistration calldata registrationInfo) internal {
                            if    (registrationInfo.ownerAddress    ==    address(0)    ||
registrationInfo.treasuryAddress == address(0)) {
            revert InvalidAddress();
        }

        if (dlpIds[registrationInfo.dlpAddress] != 0) {
```

```
            revert InvalidDlpStatus();
        }

                                if    (dlpNameToId[registrationInfo.name]    !=    0    ||
!_validateDlpNameLength(registrationInfo.name)) {
            revert InvalidName();
        }

        if (msg.value < dlpRegistrationDepositAmount) {
            revert InvalidDepositAmount();
        }
        //...code
        (bool success, ) = payable(address(treasury)).call{value: msg.value}("");
        if (!success) {
            revert TransferFailed();
        }
    }
//QueryEngineImplementation
    function initialize(
        address ownerAddress,
        address initRefinerRegistryAddress,
        DataAccessTreasuryProxyFactory initDataAccessTreasuryFactory
    ) external initializer {
        __UUPSUpgradeable_init();
        __Pausable_init();
        __AccessControl_init();
        __ReentrancyGuard_init();

        refinerRegistry = IDataRefinerRegistry(initRefinerRegistryAddress);

        /// @dev Deploy a new data access treasury for the query engine via beacon proxy
        address impl = initDataAccessTreasuryFactory.implementation();
        address proxy = initDataAccessTreasuryFactory.createBeaconProxy(
               abi.encodeCall(DataAccessTreasuryImplementation(payable(impl)).initialize,
(ownerAddress, address(this)))
        );
        queryEngineTreasury = IDataAccessTreasury(proxy);

        _setRoleAdmin(MAINTAINER_ROLE, DEFAULT_ADMIN_ROLE);
        _setRoleAdmin(QUERY_ENGINE_ROLE, DEFAULT_ADMIN_ROLE);
```

```
        _grantRole(DEFAULT_ADMIN_ROLE, ownerAddress);
        _grantRole(MAINTAINER_ROLE, ownerAddress);
    }
    //...code
 function _requestPayment(address token, uint256 amount, bytes memory metadata) internal
{
       //...code
        uint256 dlpPaymentAmount = (amount * dlpPaymentPercentage) / ONE_HUNDRED_PERCENT;
        uint256 vanaPaymentAmount = receivedAmount - dlpPaymentAmount;

         /// @dev Transfer the DLP payment portion to the treasury to be distributed to
the DLP treasury
        if (isVana) {
            payable(address(queryEngineTreasury)).sendValue(dlpPaymentAmount);
            payable(vanaTreasury).sendValue(vanaPaymentAmount);
        } else {
            IERC20(token).safeTransfer(address(queryEngineTreasury), dlpPaymentAmount);
            IERC20(token).safeTransfer(vanaTreasury, vanaPaymentAmount);
        }

        /// @dev Store the payment for the DLP treasury to be claimed by the DLP later.
           /// @dev This prevents errors when transferring funds to the DLP treasury
disrupts the payment flow.
        _dlpPayments[dlpId][token] += dlpPaymentAmount;
    }
```

An attacker could exploit this and launch a frontrunning attack.

Proof of Concept

This test demonstrates that it is possible to register dlp with a zero deposit.

```
   function createDlpRegistration(
        address dlpAddr,
        address ownerAddr,
        address treasuryAddr,
        string memory name
    ) internal pure returns (IDLPRegistry.DlpRegistration memory) {
        return IDLPRegistry.DlpRegistration({
            dlpAddress: dlpAddr,
```

```
            ownerAddress: ownerAddr,
            treasuryAddress: payable(treasuryAddr),
            name: name,
            iconUrl: "https://example.com/icon.png",
            website: "https://example.com",
            metadata: "Test DLP metadata"
        });
    }

   function test_RegisterDlpWithZeroDeposit() public {
        vm.assertEq(dlpRegistry.dlpRegistrationDepositAmount(), 0);
        vm.assertEq(dlpRegistry.dlpsCount(), 0);
        vm.startPrank(dlpOwner1);
           dlpRegistry.registerDlp{value:0}(createDlpRegistration(dlpAddress1, dlpOwner1,
treasuryAddress1, "Test DLP"));
       vm.assertEq(dlpRegistry.dlpsCount(), 1);
}
```

Test result:

```
forge test --mt test_RegisterDlpWithZeroDeposit --via-ir

Ran 1 test for test/TestContract.t.sol:TestContract
[PASS] test_RegisterDlpWithZeroDeposit() (gas: 462973)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 10.14ms (1.49ms CPU time)

Ran 1 test suite in 252.44ms (10.14ms CPU time): 1 tests passed, 0 failed, 0 skipped (1
total tests)
```

The situation is similar to the `QueryEngineImplementation` contract.

Impact

Potential spam attack.

Potential loss of profit for the protocol.

Recommendation

Important variables for the protocol should be initialized in the `initialize` function.

Status

Acknowledged

[M-03] DLPPerformanceImplementation#calculateEpochDlpRewards - Using current metric weights instead of historical ones

Description

The contract saves a snapshot of metric weights at the end of the epoch, but uses the current global weights `_metricWeights` instead of historical `_epochPerformances[epochId].metricWeights` when calculating rewards.

Vulnerability Details

The `confirmEpochFinalScores` function calculates and saves rewards for the epoch and finalizes the epoch.

First, the current metrics are saved in the `_epochPerformances` mapping.

Then, the view function `calculateEpochDlpRewards` is used to calculate rewards.

```
            function    confirmEpochFinalScores(uint256    epochId)    external    override
onlyRole(MAINTAINER_ROLE) {
        vanaEpoch.createEpochs();

        IVanaEpoch.EpochInfo memory epoch = vanaEpoch.epochs(epochId);

        if (epoch.isFinalized) {
            revert EpochAlreadyFinalized();
        }

        if (epoch.endBlock > block.number) {
            revert EpochNotEnded();
        }

=>      _epochPerformances[epochId].metricWeights = _metricWeights;

        uint256[] memory eligibleDlps = dlpRegistry.eligibleDlpsListValues();
        uint256 eligibleDlpsCount = eligibleDlps.length;
```

```
                                IVanaEpoch.Rewards[]    memory    dlpRewards    =    new
IVanaEpoch.Rewards[](eligibleDlpsCount);
        for (uint256 i = 0; i < eligibleDlpsCount; ) {
=>      (uint256 rewardAmount, uint256 penaltyAmount) = calculateEpochDlpRewards(epochId,
eligibleDlps[i]);

            dlpRewards[i] = IVanaEpoch.Rewards({
                dlpId: eligibleDlps[i],
                rewardAmount: rewardAmount,
                penaltyAmount: penaltyAmount
            });

            unchecked {
                ++i;
            }
        }

        vanaEpoch.saveEpochDlpRewards(epochId, dlpRewards);
    }
```

This function also takes global metrics.

This is where the problem lies.

```
   function calculateEpochDlpRewards(
        uint256 epochId,
        uint256 dlpId
    ) public view override returns (uint256 rewardAmount, uint256 penaltyAmount) {
          EpochDlpPerformance          storage          epochDlpPerformance          =
_epochPerformances[epochId].epochDlpPerformances[dlpId];

=>      MetricWeights memory weights = _metricWeights;

        uint256 epochRewardAmount = vanaEpoch.epochs(epochId).rewardAmount;
        uint256 dataAccessFeesRewardAmount = (epochRewardAmount * weights.dataAccessFees)
/ 1e18;
        uint256 tradingVolumeRewardAmount = (epochRewardAmount * weights.tradingVolume) /
1e18;
```

```
        uint256 uniqueContributorsRewardAmount = (epochRewardAmount *
weights.uniqueContributors) / 1e18;
      //...code
    }
```

Once the epoch is complete, `MAINTAINER` can update the metrics via the `updateMetricWeights` function.

```
  function updateMetricWeights(MetricWeights calldata newMetricWeights) external override
onlyRole(MAINTAINER_ROLE) {
        if (
                newMetricWeights.tradingVolume + newMetricWeights.uniqueContributors +
newMetricWeights.dataAccessFees !=
            1e18
        ) {
            revert InvalidMetricWeights();
        }
=>      _metricWeights = newMetricWeights;

        emit MetricWeightsUpdated(
            newMetricWeights.tradingVolume,
            newMetricWeights.uniqueContributors,
            newMetricWeights.dataAccessFees
        );
    }
```

Then call the `overrideEpochDlpPenalty` , `overrideEpochPerformances` or `overrideEpochDlpReward` function.

```
function overrideEpochPerformances(
        uint256 epochId,
        EpochDlpPerformanceInput[] calldata newEpochDlpPerformances
    ) external override onlyRole(MAINTAINER_ROLE) whenNotPaused {
            //...code
            overrideEpochDlpReward(epochId, newEpochDlpPerformance.dlpId);

            unchecked {
                ++i;
            }
```

```
        }
    }

    function overrideEpochDlpPenalty(
        uint256 epochId,
        uint256 dlpId,
        uint256 tradingVolumeScorePenalty,
        uint256 uniqueContributorsScorePenalty,
        uint256 dataAccessFeesScorePenalty
    ) external onlyRole(MAINTAINER_ROLE) {
       //...code
        overrideEpochDlpReward(epochId, dlpId);
    }

       function overrideEpochDlpReward(uint256 epochId, uint256 dlpId) public override
onlyRole(MAINTAINER_ROLE) {
        IVanaEpoch.EpochInfo memory epoch = vanaEpoch.epochs(epochId);

        if (!epoch.isFinalized) {
            return;
        }

=>      (uint256 rewardAmount, uint256 penaltyAmount) = calculateEpochDlpRewards(epochId,
dlpId);

                    uint256   distributedPenaltyAmount   =   vanaEpoch.epochDlps(epochId,
dlpId).distributedPenaltyAmount;
        if (penaltyAmount < distributedPenaltyAmount) {
            revert PenaltyAmountLessThanPenaltyDistributed(epochId, dlpId, penaltyAmount,
distributedPenaltyAmount);
        }

        vanaEpoch.overrideEpochDlpReward(epochId, dlpId, rewardAmount, penaltyAmount);
    }
```

The `overrideEpochDlpReward` function also calls the `calculateEpochDlpRewards` view function. As it turns out, this function uses global metrics that have already been changed. Accordingly, the results will also be changed. This can be manipulated.

This is possible in several cases

1) If `MAINTAINER` is compromised

2) If `MAINTAINER` is malicious, it leads to the risk of centralization

3) Human factor - Unaware of this problem, `MAINTAINER` may accidentally change the results.

Impact

Incorrect distribution in this era will lead to a loss of trust and financial losses.

Recommendation

Use historical metrics from `epochId`

```
 function calculateEpochDlpRewards(
        uint256 epochId,
        uint256 dlpId
    ) public view override returns (uint256 rewardAmount, uint256 penaltyAmount) {
           EpochDlpPerformance           storage           epochDlpPerformance           =
_epochPerformances[epochId].epochDlpPerformances[dlpId];

-       MetricWeights memory weights = _metricWeights;
+       MetricWeights memory weights = _epochPerformances[epochId].metricWeights
        //...code
}
```

Status

Acknowledged

Low

[L-01] DLPPerformanceImplementation#saveEpochPerformances - Missing Score Validation

Description

The `saveEpochPerformances` function in the `DLPPerformanceImplementation` contract contains validation checks that have been commented out. These checks were designed to ensure that the sum of scores for each metric equals 1e18 (representing 100% distribution) with a tolerance of ±1e9 for calculation errors.

```
    //commented just on moksha
         //        if (tradingVolumeTotalScore > 1e18 || tradingVolumeTotalScore < 1e18 -
1e9) {
        //            revert InvalidTradingVolumeScore();
        //        }
        //
        //        if (uniqueContributorsTotalScore > 1e18 || uniqueContributorsTotalScore
< 1e18 - 1e9) {
        //            revert InvalidUniqueContributorsScore();
        //        }
        //
         //        if (dataAccessFeesTotalScore > 1e18 || dataAccessFeesTotalScore < 1e18
- 1e9) {
        //            revert InvalidDataAccessFeesScore();
        //        }
```

Recommendation

Consider uncommenting this check.

Status

Resolved

[L-02] VanaEpochImplementation - Inconsistent Error Message in `overrideEpochDlpReward`

Description

In the `overrideEpochDlpReward` function, during the epoch finish check, the "`EpochAlreadyFinalized`" error is used, which is incompatible with the check. Initially, the error should be called if the epoch is not finished. This can cause confusion during debugging.

```
if (!epoch.isFinalized) {
            revert EpochAlreadyFinalized();
        }
```

Recommendation

Update the error message for consistency

Status

Resolved

[L-03] Contracts - No checks for zero values

Description

All contracts lack checks for zero values. The absence of this check can lead to unforeseen consequences in the future and unpredictable protocol behavior. This is especially important given that some variables do not have update functions.

Recommendation

Add the appropriate checks to the functions.

Status

Acknowledged

[L-04] DLPRewardDeployerImplementation#distributeRewards - DoS

Description

This function calculates the tranche amount, which is calculated as follows.

```
uint256 trancheAmount = (totalRewardToDistribute - epochDlpReward.totalDistributedAmount)
/(numberOfTranches - epochDlpReward.tranchesCount);

++epochDlpReward.tranchesCount;
```

There is a DoS risk because `numberOfTranches` is only defined once in the `initialize` function, and `epochDlpReward.tranchesCount` will increase without any checks. This creates an even bigger problem because during initialization, there is no check to ensure that `numberOfTranches` is greater than `0`. The deployer could accidentally set this value to `0`.

Recommendation

Add a check to ensure that the difference between the two values is always greater than `0`.

Status

Resolved

Gas

[G-01] DLPPerformanceImplementation/VanaEpochImplementation - Removal of Hardhat Console Log Import

Description

The import statement for `hardhat/console.sol` found in the Solidity contract is intended for development purposes, facilitating debugging during the development and testing phases by allowing developers to print variable values and contract states.

Recommendation

Delete the import statement for `hardhat/console.sol` from the contract files. This change ensures that debugging tools are not inadvertently included in the production deployment process.

Status

Acknowledged

[G-02] VanaEpochImplementation - Dead Code

Description

This contract contains "dead code", which was used for debugging and testing the code. Unused errors increase the bytecode of the smart contract, thereby increasing deployment costs.

```
error Test(uint256 a, uint256 b);
```

Recommendation

Remove the unused error.

Status

Acknowledged

QA

[Q-01] DLPRewardDeployerImplementation - `ReentrancyGuardUpgradeable` is not initialized

Description

All OpenZeppelin's upgradeable contracts provide a `__{ContractName}__init` function that should be called on initialization.

However, the `DLPRewardDeployerImplementation` contract fails to call the `__ReentrancyGuard_init` function on initialization. While the outcome of this missed initialization is just a higher gas cost for the first execution of the `nonReentrant` modifier, it is still a good practice to call the initialization function, and it could prevent future issues if the contract is modified.

```
function initialize(
    address ownerAddress,
    address dlpRegistryAddress,
    address vanaEpochAddress,
    address dlpRewardSwapAddress,
    uint256 newNumberOfTranches,
    uint256 newRewardPercentage,
    uint256 newMaximumSlippagePercentage
) external initializer {
    __AccessControl_init();
    __UUPSUpgradeable_init();
    __Pausable_init();

    dlpRegistry = IDLPRegistry(dlpRegistryAddress);
    vanaEpoch = IVanaEpoch(vanaEpochAddress);
    dlpRewardSwap = IDLPRewardSwap(dlpRewardSwapAddress);

    numberOfTranches = newNumberOfTranches;
    rewardPercentage = newRewardPercentage;
    maximumSlippagePercentage = newMaximumSlippagePercentage;

    _setRoleAdmin(REWARD_DEPLOYER_ROLE, MAINTAINER_ROLE);
```

```
        _grantRole(DEFAULT_ADMIN_ROLE, ownerAddress);
        _grantRole(MAINTAINER_ROLE, ownerAddress);
        _grantRole(REWARD_DEPLOYER_ROLE, ownerAddress);
    }
```

Recommendation

Initialize `ReentrancyGuardUpgradeable`

Status

Acknowledged

Centralisation

The Vana project values security and utility over decentralisation.

The owner executable functions within the protocol increases security and functionality but depend highly on internal team responsibility.

Conclusion

After Hashlock's analysis, the Vana project seems to have a sound and well-tested code base, now that our vulnerability findings have been resolved and acknowledged. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented on as well. To the best of our ability, Hashlock is not able to identify any further vulnerabilities.

Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits is to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:

In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behaviour when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our methodologies include manual code analysis, user interface interaction, and white box penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high-level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyse the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contract details are made public.

Disclaimers

Hashlock's Disclaimer

Hashlock's team has analysed these smart contracts in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Hashlock is not responsible for the safety of any funds and is not in any way liable for the security of the project.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.

About Hashlock

Hashlock is an Australian-based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3-oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

Website: hashlock.com.au
Contact: info@hashlock.com.au