Security Review Report
NM-0411-0519 Vana DLP Incentives

NETHERMIND
SECURITY

(May 22, 2025)

Contents

1 Executive Summary

This document presents the results of the security review conducted by Nethermind Security on the Vana DLP Incentive code refactor, which changes how DLP rewards are distributed as well as an overall improvement to the smart contract structure. DLP performance is tracked by an off-chain microservice which pushes performance data on-chain, where this performance is normalized and each DLP is allocated a portion of each epoch's rewards.

Each DLP will own a Uniswap V3 position between Wrapped Wana (WVANA) and the DLP's token. The reward distribution mechanism operates by adding WVANA liquidity to the pool and swapping WVANA for the DLP token, sending the swap output tokens to the DLP's treasury, and increasing the token's price.

The audit comprises 2658 lines of solidity code, excluding interface files. **The audit was performed using** (a) manual analysis of the codebase, (b) automated analysis tools, and (c) creation of test cases.

Along this document, we report 9 points of attention, where two are classified as `Low`, five are classified as `Informational`, and two are classified as `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 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.

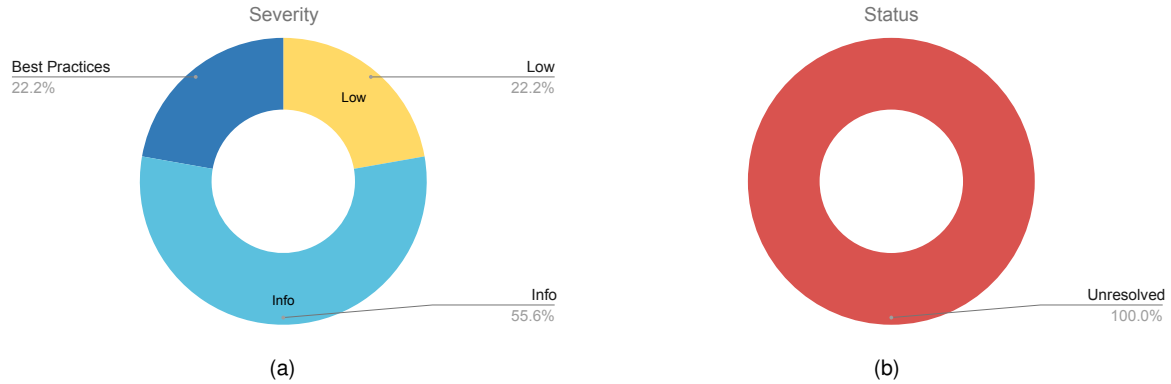


Fig. 1: Distribution of issues: Critical 0, **High** (0), **Medium** (0), **Low** (2), **Undetermined** (0), **Informational** (5), **Best Practices** (2). **Distribution of status: Fixed** (0), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (9)

Summary of the Audit

Audit Type	Security Review
Initial Report	May 22, 2025
Final Report	-
Repositories	`smart-contracts-dlp-rewards-public` `vana-smart-contracts`
Initial Commit	`238a186fddb90a8b8b34f3a598eaa84629f8d2ac` `a64234b78ae6a18543fbbb75981450d63c080f93`
Final Commit	-
Documentation	`README.md`
Documentation Assessment	Medium
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	contracts/dlpPerformance/DLPPerformanceImplementation.sol	100	2	2.0%	24	126
2	contracts/dlpPerformance/DLPPerformanceProxy.sol	5	1	20.0%	2	8
3	contracts/swapHelper/SwapHelperProxy.sol	5	1	20.0%	2	8
4	contracts/swapHelper/SwapHelperImplementation.sol	349	76	21.8%	74	499
5	contracts/swapHelper/libraries/LiquidityAmounts.sol	88	44	50.0%	13	145
6	contracts/swapHelper/libraries/PoolAddress.sol	30	12	40.0%	4	46
7	contracts/swapHelper/libraries/TickMath.sol	176	22	12.5%	16	214
8	contracts/swapHelper/libraries/SqrtPriceMath.sol	144	73	50.7%	20	237
9	contracts/swapHelper/libraries/FullMath.sol	67	51	76.1%	10	128
10	contracts/swapHelper/libraries/SwapMath.sol	76	17	22.4%	7	100
11	contracts/dlpRegistry/DLPRegistryProxy.sol	5	1	20.0%	2	8
12	contracts/dlpRegistry/DLPRegistryImplementation.sol	325	16	4.9%	81	422
13	contracts/dlpRewardDeployer/DLPRewardDeployerImplementation.sol	141	2	1.4%	36	179
14	contracts/dlpRewardDeployer/DLPRewardDeployerProxy.sol	5	1	20.0%	2	8
15	contracts/dlpRewardSwap/DLPRewardSwapImplementation.sol	521	59	11.3%	74	654
16	contracts/dlpRewardSwap/DLPRewardSwapProxy.sol	5	1	20.0%	2	8
17	contracts/treasury/TreasuryImplementation.sol	67	7	10.4%	18	92
18	contracts/treasury/DlpRewardDeployerTreasuryProxy.sol	5	1	20.0%	2	8
19	contracts/treasury/DLPRegistryTreasuryProxy.sol	5	1	20.0%	2	8
20	contracts/vanaEpoch/VanaEpochImplementation.sol	191	52	27.2%	51	294
21	contracts/vanaEpoch/VanaEpochProxy.sol	5	1	20.0%	2	8
	Total	**2315**	**441**	**19.0%**	**444**	**3200**

	Contract	LoC	Comments	Ratio	Blank	Total
1	contracts/dat/DATPausable.sol	45	2	4.4%	9	56
2	contracts/dat/DATFactoryProxy.sol	5	1	20.0%	2	8
3	contracts/dat/DATVotes.sol	58	16	27.6%	12	86
4	contracts/dat/DATFactoryImplementation.sol	132	27	20.5%	33	192
5	contracts/dat/DAT.sol	103	25	24.3%	23	151
	Total	**343**	**71**	**20.7%**	**79**	**493**

3 Summary of Issues

	Finding	Severity	Update
1	Blocked `DAT` addresses still have voting power	Low	Unresolved
2	DLP ownership is not checked during registration	Low	Unresolved
3	DLP eligibility is updated and managed inconsistently	Info	Unresolved
4	Previous epoch final check in `_setDlpEligibility` may fail	Info	Unresolved
5	Token factory `maxCapDefault` may revert token deployments	Info	Unresolved
6	Unnecessary role admin assignments	Info	Unresolved
7	`DAT` tokens have inconsistent maximum supplies	Info	Unresolved
8	Unused code	Best Practices	Unresolved
9	`IVanaEpoch` interface is missing functions	Best Practices	Unresolved

4 System Overview

Each contract is described below, with the next page presenting a protocol interaction diagram detailing function calls between contracts:

DlpRegistry: DLP owners can register their DLP addresses to this contract, and become eligible for rewards by meeting requirements.

VanaEpoch: Contract which tracks epochs as they elapse, allowing finalization and performance tracking.

DlpPerformance: An intermediary contract that an offchain microservice sends performance data to, and forwards to `VanaEpoch`.

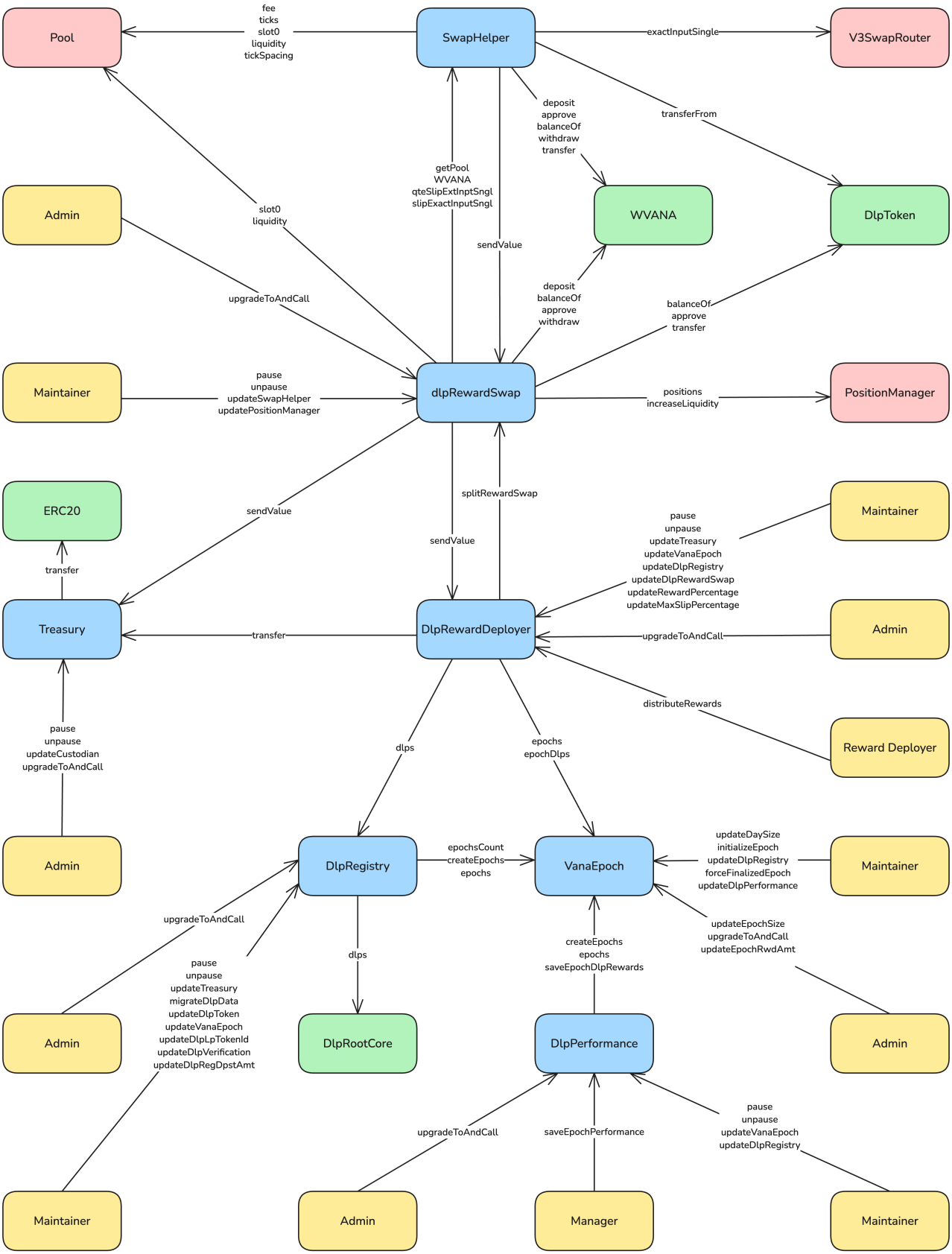
DlpRewardDeployer: Moves assets from the treasury and assists in the distribution of rewards using performance data from `VanaEpoch`.

DlpRewardSwap: Assists during reward distribution by swapping assets to DLP's treasury addresses and providing liquidity to DLP LPs.

SwapHelper: Contract with a series of utility functions to assist in swapping assets, used by the `DlpRewardSwap` contract.

Treasury: Stores rewards that will be distributed to DLPs, assets are taken from the `DlpRewardDeployer` contract.

4.1 Contract Interaction Diagram



4.2 Reward mechanism design notes

Upon registration each DLP provides a treasury where a portion of the rewards will be sent, and to be eligibly for rewards they must also have a DLP token and own a Uniswap V3 position in a pool between `WVANA` and their token. Once an epoch has been finalized a privileged role can call `distributeRewards` which will allocate some amount of `VANA` to provide liquidity to the DLP's LP pair. If the LP deposit requires some amount of DLP token and `WVANA` then a swap will be done beforehand to have the amount of tokens needed. After liquidity has been added, the remaining about of `VANA` will be used to swap directly for the DLP token, with those tokens being sent to the DLP's treasury. There are onchain slippage protection checks applied to ensure that the price impact cannot exceed a specified amount.

This reward delivery approach does introduce a risk of frontrunning, where an attacker could anticipate reward distribution and interact with relevant pools before and after the rewards are sent. The Vana team has explained that the mempool private, and that reward distribution will be done at regular intervals, but with small random time deviation prevent any observation on consistent reward distribution timings.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security 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 the finding is to be uncovered and exploited by an attacker. 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 Security 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 pass to the client formally;

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 Issues

6.1 [Low] Blocked DAT addresses still have voting power

File(s): DATVotes.sol

Description: The DATVotes contract introduces voting to the base DAT contract, and integrates the blocking feature into some of the newly introduced functions. Both _delegate and _transferVotingUnits include the modifier whenNotBlocked to ensure that addresses cannot assign their voting power to another address while blocked.

```
// Modifier whenNotBlocked has been added
function _delegate(...) internal override whenNotBlocked(delegator, delegatee) {
    super._delegate(delegator, delegatee);
}

function _transferVotingUnits(...) internal override whenNotBlocked(from, to) {
    super._transferVotingUnits(from, to, amount);
}
```

However, the functions getVotes and getPastVotes have no such protection and if called by a governance contract, they will return the full voting power regardless of whether an address has been blocked. Given that blocked addresses are not able to transfer their voting rights to vote using another address, it is implied that they should also not be able to vote from their own address.

Recommendation(s): In order to prevent blocked addresses from using their voting power we propose a potential solution; however, it has a weakness. The solution is to add the modifier whenNotBlocked to the functions getVotes and getPastVotes. This will prevent a blocked address from using both its historical and current voting weight, as any attempt to use their votes would revert.

The weakness of this approach is that the historical (tracked in checkpoints) and current total token supply will remain unchanged. If an address with a significant amount of tokens is blocked, reaching quorum may be more difficult or even impossible. For example, if governance quorum is 50% but a blocked address holds 60% of the supply, the remaining 40% of unblocked tokens will not be able to reach quorum.

A potential solution exists that allows proper accounting for blocked addresses in getVotes and getPastVotes however, it may include modifying checkpoint data for both user voting power and total supplies, and might require handling historical voting unit transfers. This approach is not recommended as it changes the inner workings of the voting mechanism and may have unexpected side effects if not implemented properly.

Status: Unresolved

Update from the client:

6.2 [Low] DLP ownership is not checked during registration

File(s): DLPRegistryImplementation.sol

Description: The function registerDlp doesn't check if the caller address is the owner of the DLP contract. This makes it possible to register any DLP as your own to the registry, leading to a situation where a legitimate DLP owner tries to register the DLP and the call will revert since the DLP has already been registered by the non-owner.

An attacker could frontrun or preemptively register DLPs using known DLP addresses with their own treasury and their own address in the ownerAddress field, without any option for the legitimate DLP owner to gain control at a later time.

Recommendation(s): Consider adding a check to ensure that the caller address is registered as the owner on the DLP contract, to prevent unauthorized registrations with incorrect data.

```
function _registerDlp(DlpRegistration calldata registrationInfo) internal {
    // ...
    bool registrationOwnerIsDlpAdmin = IDLP(registrationInfo.dlpAddress).hasRole(
        DEFAULT_ADMIN_ROLE,
        registrationInfo.ownerAddress
    );

    if(!registrationOwnerIsDlpAdmin) {
        revert InvalidOwner();
    }
    // ...
}
```

Status: Unresolved

Update from the client:

6.3 [Info] DLP eligibility is updated and managed inconsistently

File(s): DLPRegistryImplementation.sol

Description: In order for a DLP to be marked as eligible, it must be verified, have a DLP token, and an LP token ID. The DLP token and LP token ID can be set through privileged setter functions only available to the maintainer, and once they are set the `updateDlpVerification` function can be called. This will set the DLP as verified and place it in the eligible DLPs list.

The final check before placing the DLP in the eligible list is done in `_setDlpEligibility`. However, this only partially verifies the eligibility prerequisites, checking the verified status and DLP token, but omits checking the LP token ID. A snippet from the function is shown below:

```
function _setDlpEligibility(Dlp storage dlp) internal {
    //...
    if (dlp.status == DlpStatus.Registered) {
        if (dlp.isVerified && dlp.tokenAddress != address(0)) {
            dlp.status = DlpStatus.Eligible;
            _eligibleDlpsList.add(dlp.id);
        }
    } else if (dlp.status == DlpStatus.Eligible) {
        if (!dlp.isVerified || dlp.tokenAddress == address(0)) {
            dlp.status = DlpStatus.Registered;
            _eligibleDlpsList.remove(dlp.id);
        }
    }
}
```

Furthermore, the setter functions `updateDlpToken` and `updateDlpLpTokenId` do not make a call to `_setDlpEligibility`. This makes it possible to change fields related to the eligibility requirements after a DLP has already been marked as eligible, without the DLP losing eligibility, even though it no longer meets the requirements.

Recommendation(s): Consider changing `_setDlpEligibility` to check for all requirements (verified, DLP token, LP token ID) and adding this function to the `updateDlpToken` and `updateDlpLpTokenId` setters.

Status: Unresolved

Update from the client:

6.4 [Info] Previous epoch final check in `_setDlpEligibility` may fail

File(s): DLPRegistryImplementation.sol

Description: The function `_setDlpEligibility` has a check to ensure that the previous epoch is finalized before making changes to a DLP's eligibility status. However, this function does not make a call to `vanaEpoch.createEpochs` before running this check. Since new epochs are not created, this check may rely on stale data where a new epoch should have been created but has not. This can cause the check to pass when it should have failed, as a new epoch should have been created, triggering a revert..

```
function _setDlpEligibility(Dlp storage dlp) internal {
    // Missing call to create epochs
    uint256 epochsCount = vanaEpoch.epochsCount();
    if (epochsCount > 1 && !vanaEpoch.epochs(epochsCount - 1).isFinalized) {
        revert LastEpochMustBeFinalized();
    }
    // ...
}
```

Recommendation(s): Consider adding a call to `vanaEpoch.createEpochs` to ensure that up-to-date epoch data is used for the check.

Status: Unresolved

Update from the client:

6.5 [Info] Token factory `maxCapDefault` may revert token deployments

File(s): DATFactoryImplementation.sol

Description: The `DATFactoryImplementation` contract uses a `maxCapDefault` variable which is defined by the admin upon deployment. This variable is used when users create new tokens to ensure that the user input parameter `cap_` stays within `minCapDefault` and `maxCapDefault` limits. The `createToken` function allows users to deploy tokens with `cap_ == 0` in which case the default value used will be `type(uint256).max` as can be seen below.

```
if (cap_ == 0) cap_ = type(uint256).max;
```

However, if `maxCapDefault` was set by the admin to any value below `type(uint256).max` then all the calls to the `createToken` method where users pass the parameter `cap_ == 0` will revert due to this check:

```
if (cap_ > maxCapDefault) revert ExcessiveCap();
```

Recommendation(s): Consider setting the `cap_ == maxCapDefault` when users deploy new tokens and don't specify a `cap_` amount.

Status: Unresolved

Update from the client:

6.6 [Info] Unnecessary role admin assignments

File(s): contracts/

Description: Multiple contracts assign the `DEFAULT_ADMIN_ROLE` as the role admin for other rules during contract initialization. However, the behavior of the OpenZeppelin `AccessControl` logic is to have the `DEFAULT_ADMIN_ROLE` assigned to a role's admin by default. Therefore, manually setting the default admin as the role manager for these roles is unnecessary.

The following is a list of all unnecessary role assignments in the codebase:

```
DLPPerformanceImplementation
    L44 -> MAINTAINER_ROLE

DLPRewardDeployerImplementation
    L58 -> MAINTAINER_ROLE

DLPRewardSwapImplementation
    L51 -> MAINTAINER_ROLE

SwapHelperImplementation
    L61 -> MAINTAINER_ROLE

TreasuryImplementation
    L50 -> CUSTODIAN_ROLE
```

Recommendation(s): Consider removing the unnecessary role admin assignments.

Status: Unresolved

Update from the client:

6.7 [Info] DAT tokens have inconsistent maximum supplies

File(s): DATFactoryImplementation.sol

Description: There are three variants of the DAT token available: Default, Votes, and Pausable. The Default token supports a max supply of type(uint256).max however, the Votes and Pausable tokens are constrained by checkpoint data for voting purposes and are limited to a maximum supply of type(uint208).max.

Users and DLP creators would generally expect that all DAT token types have the same behavior when it comes to maximum supply, however this is not the case. Furthermore, the token factory interprets a cap of zero as type(uint256).max meaning that using this feature on Votes and Pausable tokens will always fail.

```
function createToken(...) external override returns (...) {
    // ...
    uint256 cap_ = params.cap;
    if (cap_ == 0) cap_ = type(uint256).max;
    // ...
}
```

Recommendation(s): Consider adjusting the max supply for the Default token to align with Votes and Pausable to ensure consistency between all token types. The factory interpreting a cap as zero to be type(uint256).max should be adjusted to the new limit as well.

Status: Unresolved

Update from the client:

6.8 [Best Practices] Unused code

File(s): contracts/

Description: The following is a list of unused code that can be removed from the codebase to reduce gas costs and/or improve readability.

```
DLPRegistryImplementation.sol
    L96     Unused call to VanaEpochs
    L306    Unused call to VanaEpochs

DLPRegistryStorageV1.sol
    L17     Unused storage variable
```

Recommendation(s): Consider removing the unused code shown above.

Status: Unresolved

Update from the client:

6.9 [Best Practices] IVanaEpoch interface is missing functions

File(s): IVanaEpoch.sol

Description: The VanaEpochImplementation contract declares the functions initializeEpoch and updateDaySize, however these functions are not declared in the interface file IVanaEpoch.

Recommendation(s): Consider adding the missing functions to IVanaEpoch.

Status: Unresolved

Update from the client:

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. 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 Vana documentation

The `Vana` team has provided a `README` file in the repository explaining the environment setup, dependency installation, and how to run the test suites. The codebase includes comments that explain the purpose of functions, as well as inline comments for sections with complex logic that require additional clarification. Throughout the audit engagement, there have been calls with the `Vana` team, which have helped provide comprehensive insight into the intended functionalities of the project.

8 Test Suite Evaluation

8.1 Compilation Output

```
> npx hardhat compile

Generating typings for: 112 artifacts in dir: typechain-types for target: ethers-v6
Successfully generated 312 typings!
Compiled 110 Solidity files successfully (evm target: paris).
```

8.2 Tests Output

```
> npx hardhat test test/dlpReward.ts

  DLP System Tests
    Dlp Reward Deployer
       should distributeRewards
       should distributeRewards multiple times

> npx hardhat test test/forkDlpReward.ts

  DLP fork tests
    Tests
       should savePerformance

> npx hardhat test test/uniswap.ts

  UniswapV3
    SwapHelper
       should be set up correctly
       should quoteExactInputSingle
       should exactInputSingle without initializedTicksCrossed
       should exactInputSingle with initializedTicksCrossed
       should simulateSwap when zeroForOne, exactIn
       should simulateSwap when zeroForOne, exactIn, with SLIPPAGE_TOLERANCE
       should simulateSwap when oneForZero, exactIn
       should simulateSwap when oneForZero, exactIn, with SLIPPAGE_TOLERANCE
       should simulateSwap when oneForZero, exactOut
       should simulateSwap when oneForZero, exactOut, with SLIPPAGE_TOLERANCE
       should simulateSwap when zeroForOne, exactOut
       should simulateSwap when zeroForOne, exactOut, with SLIPPAGE_TOLERANCE
       should slippageExactInputSingle
    LpSwap
       should quoteLpSwap (500 / 500) 250000 / 250000
       should lpSwap (fuzzing) 25000 / 25000
       should lpSwap (single input)
       should splitRewardSwap (fuzzing) 25000 / 25000
       should splitRewardSwap (single)
       should splitRewardSwap with existing position lpTokenId (moksha) 50000 / 50000
       should quoteExactInputSingle on moksha
```

8.3 Automated Tools

8.3.1 AuditAgent

All the relevant issues raised by the AuditAgent have been incorporated into this report. The AuditAgent is an AI-powered smart contract auditing tool that analyses code, detects vulnerabilities, and provides actionable fixes. It accelerates the security analysis process, complementing human expertise with advanced AI models to deliver efficient and comprehensive smart contract audits. Available at https://app.auditagent.nethermind.io.

9 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 process, 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

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.