Security Review Report
NM-0451-0486 VANA STAKING

NETHERMIND
SECURITY

(April 9, 2025)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind Security for VANA protocol smart contracts. The audit focused on pull request #13, which introduces a new stacking mechanism, i.e. `VanaPool`, and changes to allow migration to it. `VanaPool` is a staking protocol allowing users to stake VANA tokens into community-managed entities with configurable APY rates.

The audited code comprises 1116 lines of code written in the Solidity language. The audit includes the new staking contract and changes to allow migration of current stakes to the new staking mechanism.

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** 14 points of attention, where five are classified as `Critical`, one is classified as `High`, three are classified as `Medium`, two are classified as `Low`, and three are classified as `Informational`. 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.

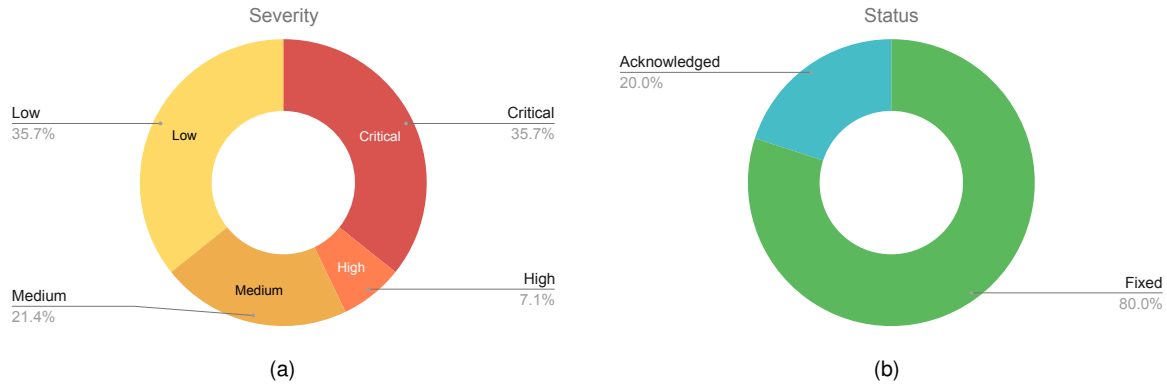

Fig. 1: Distribution of issues: Critical (5), **High** (1), **Medium** (3), **Low** (2), **Undetermined** (0), **Informational** (3), **Best Practices** (0). **Distribution of status: Fixed** (12), **Acknowledged** (2), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Response from Client	Regular responses during audit engagement
Final Report	April 9th, 2025
Repository	vana-smart-contracts
Commit (Audit)	1da399c41f3f6f3fcc00b704d47e02fc6f3a56ca
Commit (Final)	32b77010b99b5dc34652ef271b5816218c542da5
Documentation Assessment	Medium
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	contracts/vanaPoolTreasury/VanaPoolTreasuryImplementation.sol	46	2	4.3%	13	61
2	contracts/vanaPoolTreasury/VanaPoolTreasuryProxy.sol	5	1	20.0%	2	8
3	contracts/vanaPoolTreasury/interfaces/VanaPoolTreasuryStorageV1.sol	5	6	120.0%	2	13
4	contracts/vanaPoolTreasury/interfaces/IVanaPoolTreasury.sol	7	1	14.3%	1	9
5	contracts/root/DLPRootImplementation.sol	468	59	12.6%	132	659
6	contracts/root/interfaces/DLPRootStorageV1.sol	39	10	25.6%	12	61
7	contracts/vanaPoolStaking/VanaPoolStakingProxy.sol	5	1	20.0%	2	8
8	contracts/vanaPoolStaking/VanaPoolStakingImplementation.sol	177	114	64.4%	58	349
9	contracts/vanaPoolStaking/interfaces/IVanaPoolStaking.sol	25	1	4.0%	8	34
10	contracts/vanaPoolStaking/interfaces/VanaPoolStakingStorageV1.sol	11	6	54.5%	5	22
11	contracts/vanaPoolEntity/VanaPoolEntityProxy.sol	5	1	20.0%	2	8
12	contracts/vanaPoolEntity/VanaPoolEntityImplementation.sol	252	108	42.9%	76	436
13	contracts/vanaPoolEntity/interfaces/IVanaPoolEntity.sol	57	3	5.3%	13	73
14	contracts/vanaPoolEntity/interfaces/VanaPoolEntityStorageV1.sol	14	7	50.0%	8	29
	Total	**1116**	**320**	**28.7%**	**334**	**1770**

3 Summary of Issues

	Finding	Severity	Update
1	Entity owners can withdraw their minimum registration stake	Critical	Fixed
2	Excessive calls to `processRewards(...)` cause higher reward rate, regardless of time elapsed	Critical	Fixed
3	Same stake can be migrated multiple times	Critical	Fixed
4	Stakers funds are locked on entity removal	Critical	Fixed
5	Users can withdraw their DLP stakes and migrate to the VANA pool simultaneously	Critical	Fixed
6	`processRewards(...)` is called too late in `unstake(...)`	High	Fixed
7	The `stake(...)` and `unstake(...)` functions lack slippage protection	Medium	Fixed
8	The `unstake(...)` function should use `entityShareToVana(...)` to round down to favor protocol	Medium	Fixed
9	Unauthorized execution of `migrateStakeToVanaPool(...)` on behalf of others	Medium	Fixed
10	Excessive locked rewards are not handled upon entity removal	Low	Acknowledged
11	`removeEntity(...)` does not clear `entityNameToId`	Low	Acknowledged
12	Inconsistent check for minimum stake amount	Info	Fixed
13	Rounding issue in `unstake(...)` allows owner to unstake below `minRegistrationStake`	Info	Fixed
14	`migrateStakeToVanaPool(...)` doesn't need to be payable	Info	Fixed

4 System Overview

The VANA team introduced new changes into their existing protocol, VanaPool. VanaPool is a staking protocol allowing users to stake VANA tokens into community-managed entities with configurable APY rates. The system implements a share-based model where stakers receive proportional claims on entity reward pools that appreciate in value as rewards accrue. The protocol features a secure treasury component for custody of staked tokens, with proper access controls governing all system operations. VanaPool supports staking on behalf of others and implements time-weighted reward distribution based on entity-specific APY settings. The new changes introduce three main contracts, each implemented behind its own proxy contract

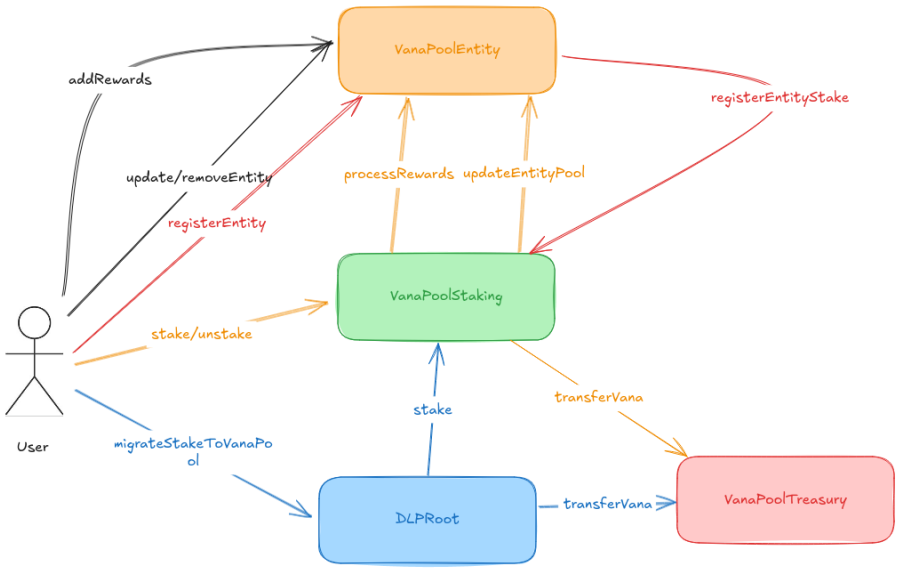

Fig. 2: Vana new changes overview

4.1 VanaPoolEntityImplementation

The VanaPoolEntityImplementation contract serves as the management layer for staking entities (pools). It handles entity creation, configuration, and reward distribution mechanics. When an admin or authorized user creates a new entity, they must provide a registration stake and basic parameters like the entity name and owner address. The contract calculates reward distribution rates based on the configured APY and manages pool rewards. Entity owners add rewards to their pools, which are then gradually released to stakers based on the configured APY.

4.2 VanaPoolStakingImplementation

The VanaPoolStakingImplementation contract handles users' core staking and unstaking operations. It acts as the interface between users and the entity system, managing individual staker positions and share calculations. When users stake VANA, the contract calculates the appropriate number of shares based on the current conversion rate from the corresponding entity. These shares represent the user's proportional claim on the entity's reward pool.

The staking process involves transferring funds to the treasury and updating both the user's share balance and the entity's total shares. For unstaking, the contract performs the reverse operation, converting shares back to VANA at the current rate and ensuring the entity owner maintains their minimum required stake.

4.3 VanaPoolTreasuryImplementation

The VanaPoolTreasuryImplementation contract serves as the secure custodian for all staked funds and rewards. Its primary function is to receive, hold, and disburse VANA tokens as directed by the other contracts in the system. The treasury operates with strict access controls, only processing transactions initiated by authorized contracts or administrators.

Main flows through the treasury include receiving staked funds during user deposits, holding rewards added by entity owners, and processing withdrawals when users unstake.

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 [Critical] Entity owners can withdraw their minimum registration stake

File(s): VanaPoolEntityImplementation.sol

Description: When creating an entity, the createEntity function requires the owner to deposit a minimum registration stake, which is intended to remain locked indefinitely

```
function unstake(uint256 entityId, uint256 shareAmount) external override nonReentrant whenNotPaused {
    // --SNIP
    if (entityInfo.ownerAddress == _msgSender()) {
        uint256 ownerMinShares = (vanaToShare * vanaPoolEntity.minRegistrationStake()) / 1e18;

        if (stakerShares - shareAmount < ownerMinShares) {
            revert CannotRemoveRegistrationStake();
        }
    }
    // --SNIP
}
```

However, the above check can be easily bypassed. The `updateEntity(...)` allows the update of the owner and **does not move the registration shares to the new owner**.

```
function updateEntity(
    uint256 entityId,
    EntityRegistrationInfo calldata entityRegistrationInfo
) external override whenNotPaused nonReentrant onlyEntityOwner(entityId) {
    // --SNIP

    // Update fields
    entity.ownerAddress = entityRegistrationInfo.ownerAddress;

    emit EntityUpdated(entityId, entityRegistrationInfo.ownerAddress, entityRegistrationInfo.name);
}
```

The entity owner can leverage this to change the `ownerAddress` to another andress he controls via `updateEntity` and then calls `unstake` to freely withdraw his registration stake.

Recommendation(s): Consider moving the registration shares to the new entity owner when updating the owner.

Status: Fixed.

Update from the client: Fixed in commit 97de94d

6.2 [Critical] Excessive calls to `processRewards(...)` cause higher reward rate, regardless of time elapsed

File(s): VanaPoolEntityImplementation.sol

Description: The `processRewards(...)` is a permissionless function which updates the rewards for an entity. It calculates the rewards to distribute based on the time elapsed and the rate of reward.

```
function processRewards(uint256 entityId) public override whenNotPaused {
    // ...

    // Calculate theoretical rewards based on maxAPY
    uint256 toDistribute = (entity.activeRewardPool * entity.ratePerSecond * timeElapsed) / 1e18;

    if (toDistribute > entity.lockedRewardPool) {
        toDistribute = entity.lockedRewardPool;
    }

    entity.lockedRewardPool -= toDistribute;
    // @audit activeRewardPool itself is increased by toDistribute
    entity.activeRewardPool += toDistribute;

    // ...
}
```

However, the function uses `activeRewardPool` to calculate the rewards and `activeRewardPool` itself is increased by the rewards to distribute. This creates an accumulating growth in the reward amount over time, which could lead to excessively increasing rewards rather than a more predictable, linear growth based on time elapsed. As a result, the rewards are influenced not only by the time elapsed but also by the number of times `processRewards(...)` is called.

For example, let's say `activeRewardPool` is `1000` and `ratePerSecond` is `2e18`. Consider these scenarios:

Scenario A:

1. The first call to `processRewards(...)` with a time elapsed of `100` results in an `activeRewardPool` of `201,000`;
2. A second call to `processRewards(...)` with a time elapsed of `200` results in an `activeRewardPool` of `80,400,000`;

Scenario B:

1. A call to `processRewards(...)` with a time elapsed of `300` results in an `activeRewardPool` of `600,000`;

In both the scenarios, time elapsed is the same (`300`), but the resulting `activeRewardPool` is significantly different.

Recommendation(s): Consider using two separate variables: one for the `activeRewardPool` and another for the `totalStake`. The `totalStake` should be used to calculate the rewards to distribute, ensuring that the rewards grow linearly with respect to time elapsed.

Status: Fixed.

Update from the client: Fixed in commits 97de94d, 265bcde, and 2177752.

6.3 [Critical] Same stake can be migrated multiple times

File(s): DLPRootImplementation.sol

Description: The DLPRootImplementation contract introduces the migrateStakeToVanaPool(...) function, which allows users to migrate their existing DLP stakes to the VANA pool. This function stakes the user's existing amount plus any claimable rewards into the pool for a specific entity.

```
function migrateStakeToVanaPool(uint256 stakeId, uint256 entityId) external payable nonReentrant whenNotPaused {
    dlpRootEpoch.createEpochs();

    Stake storage stake = _stakes[stakeId];

    if (stake.endBlock == 0) {
        stake.movedAmount = stake.amount;
        _closeStake(_msgSender(), stakeId);
        stake.movedAmount = 0;
    }

    uint256 toClaim = stake.amount + _calculateStakeRewardUntilEpoch(stakeId, dlpRootEpoch.epochsCount() - 1, true);

    if (toClaim == 0) {
        revert NothingToClaim();
    }

    bool success = dlpRootRewardsTreasury.transferVana(payable(address(this)), toClaim);
    if (!success) {
        revert TransferFailed();
    }

    vanaPoolStaking.stake{value: toClaim}(entityId, stake.stakerAddress);

    emit StakeMigratedToVanaPool(stakeId, stake.amount, entityId);
}
```

However, the function does not update the stake.amount to 0 or delete its entry from the _stakes mapping, allowing users to call this function repeatedly.

Recommendation(s): Consider clearing the _stakes entry or updating stake.amount to 0 for the migrated stake.

Status: Fixed.

Update from the client: Fixed in commit 97de94d

6.4 [Critical] Stakers funds are locked on entity removal

File(s): VanaPoolEntityImplementation

Description: Users can create entities via `createEntity(...)`, and stakers can allocate their funds to a specific entity ID through `VanaPoolStakingImplementation::stake(...)`. However, entity owners can remove an entity at any time, updating its status to `Removed`:

```
function removeEntity(uint256 entityId) external override whenNotPaused nonReentrant onlyEntityOwner(entityId) {
    Entity storage entity = _entities[entityId];

    if (entity.status != EntityStatus.Active) {
        revert InvalidEntityStatus();
    }

    // Process any pending rewards
    processRewards(entityId);
==>    entity.status = EntityStatus.Removed;
    _activeEntityIds.remove(entityId);

    emit EntityStatusUpdated(entityId, EntityStatus.Removed);
}
```

Once an entity is removed, stakers who have already staked funds into it become unable to unstake, as the contract restricts unstaking from non-active entities:

```
function unstake(uint256 entityId, uint256 shareAmount) external override nonReentrant whenNotPaused {
    // --SNIP
==>    vanaPoolEntity.updateEntityPool(entityId, shareAmount, exactVanaAmount, false);

    bool success = vanaPoolTreasury.transferVana(payable(_msgSender()), exactVanaAmount);
    if (!success) {
        revert TransferFailed();
    }
}

function updateEntityPool(
    uint256 entityId,
    uint256 shares,
    uint256 amount,
    bool isStake
) external override whenNotPaused onlyRole(VANA_POOL_ROLE) {
    if (!hasRole(VANA_POOL_ROLE, msg.sender)) {
        revert NotAuthorized();
    }

    Entity storage entity = _entities[entityId];

    if (entity.status != EntityStatus.Active) {
==>        revert InvalidEntityStatus();
    }

    // --SNIP
}
```

As a result, stakers' funds become permanently locked, as they are unable to unstake from a removed entity.

Recommendation(s): There are many approaches to fix the issue, it depends on how the team intends to fix the issue. Two possible approaches include:

1. Allow unstaking from removed entities. Since the stake funds are held in the treasury contract rather than the entities themselves, enabling unstaking from removed entities would resolve the issue without security risks;
2. Restrict entity owners from removing an entity while it has active stakes;

Status: Fixed.

Update from the client: Fixed by removing the `removeEntity(...)` function in db7afd9

6.5 [Critical] Users can withdraw their DLP stakes and migrate to the VANA pool simultaneously

File(s): DLPRootImplementation.sol

Description: The `DLPRootImplementation` contract introduces the `migrateStakeToVanaPool(...)` function, which allows users to migrate their existing VANA stakes to the VANA pool. This function stakes the user's existing amount plus any claimable rewards into the pool for a specific entity.

```
function migrateStakeToVanaPool(uint256 stakeId, uint256 entityId) external payable nonReentrant whenNotPaused {
    dlpRootEpoch.createEpochs();

    Stake storage stake = _stakes[stakeId];

    if (stake.endBlock == 0) {
        stake.movedAmount = stake.amount;
        _closeStake(_msgSender(), stakeId);
        stake.movedAmount = 0;
    }

    uint256 toClaim = stake.amount + _calculateStakeRewardUntilEpoch(stakeId, dlpRootEpoch.epochsCount() - 1, true);

    if (toClaim == 0) {
        revert NothingToClaim();
    }

    bool success = dlpRootRewardsTreasury.transferVana(payable(address(this)), toClaim);
    if (!success) {
        revert TransferFailed();
    }

    vanaPoolStaking.stake{value: toClaim}(entityId, stake.stakerAddress);

    emit StakeMigratedToVanaPool(stakeId, stake.amount, entityId);
}
```

However, the function does not properly update the `_stakes` mapping to mark the stake as withdrawn or delete its entry, leading to a double withdrawal exploit which can be exploited by users to migrate their stakes to vana pools and withdraw them from the `DLPRoot` contract as well. Consider the following exploit path:

1. Bob has an active stake in `DLPRootImplementation`;
2. Bob closes his stake by calling `closeStake(...)`;
3. Bob migrates his stake to VANA pool using `migrateStakeToVanaPool(...)`;
4. Bob withdraws his stake using `withdrawStake(...)`, even though it has already been migrated;

Since the contract does not properly mark the stake as migrated/withdrawn, Bob effectively retrieves his original stake while also having it staked in the VANA pool, leading to contract losses.

Similarly, as there's no check to ensure that already withdrawn stakes cannot be migrated, users can migrate their already withdrawn stakes to the VANA pool.

Recommendation(s): Consider adding a check to restrict already withdrawn stakes and clearing the `_stakes` entry or set the `withdrawn` status to true for the migrated stake.

Status: Fixed.

Update from the client: Fixed in commit 97de94d

6.6 [High] `processRewards(...)` is called too late in `unstake(...)`

File(s): VanaPoolStakingImplementation.sol

Description: The `unstake(...)` function allows users to withdraw their staked VANA tokens, receiving an amount proportional to their shares in the entity. However, `processRewards(...)` is called after the proportion calculation, which means rewards that should be included in the `vanaToShare` conversion are not accounted for

```
function unstake(uint256 entityId, uint256 shareAmount) external override nonReentrant whenNotPaused {
    uint256 stakerShares = _stakers[_msgSender()].entities[entityId].shares;
    if (stakerShares == 0 || shareAmount == 0) {
        revert InvalidAmount();
    }

    IVanaPoolEntity.EntityInfo memory entityInfo = vanaPoolEntity.entities(entityId);

    uint256 vanaToShare = vanaPoolEntity.vanaToEntityShare(entityId);

==>     vanaPoolEntity.processRewards(entityId);

    // --SNIP

    uint256 exactVanaAmount = (1e18 * shareAmount) / vanaToShare;
    // --SNIP
```

Since `vanaToShare` is calculated before processing rewards, it does not reflect the most up-to-date active rewards, causing users to receive fewer VANA tokens than they should.

Recommendation(s): Consider calling `vanaPoolEntity::processRewards(...)` before calculating the `vanaToShare`.

Status: Fixed.

Update from the client: Fixed in commit 97de94d

6.7 [Medium] The `stake(...)` and `unstake(...)` functions lack slippage protection

File(s): VanaPoolStakingImplementation.sol

Description: Users interact with `VanaPoolStakingImplementation` via the `stake(...)` and `unstake(...)` functions to allocate and deallocate funds from specific entities. However, these functions lack slippage protection, exposing users to potential value losses due to fluctuations in share distribution.

For example, if a user attempts to unstake their shares, they might receive fewer VANA tokens than expected if other stake transactions are processed just before them, increasing the total shares and decreasing their withdrawal amount.

Recommendation(s): Allow users to specify acceptable minimum values in both `stake(...)` and `unstake(...)` functions.

Status: Fixed.

Update from the client: Fixed in commit 97de94d

6.8 [Medium] The `unstake(...)` function should use `entityShareToVana(...)` to round down to favor protocol

File(s): VanaPoolStakingImplementation.sol

Description: The `unstake(...)` function allows users to unstake a specified amount of shares and transfer equivalent amount of VANA tokens to the user. To convert the shares to VANA tokens, it divides the share amount by the result of the `vanaToEntityShare(...)` function.

```
function unstake(uint256 entityId, uint256 shareAmount) external override nonReentrant whenNotPaused {
    // ...

    // @audit vanaToEntityShare(...) rounds down the result
    uint256 vanaToShare = vanaPoolEntity.vanaToEntityShare(entityId);

    // ...

    // Store the exact VANA amount corresponding to shares at this point
    // @audit Dividing by a smaller value increases the result
    uint256 exactVanaAmount = (1e18 * shareAmount) / vanaToShare;

    // Update staker's position
    _stakers[_msgSender()].entities[entityId].shares -= shareAmount;

    // Update entity staking data in VanaPoolEntity contract
    vanaPoolEntity.updateEntityPool(entityId, shareAmount, exactVanaAmount, false);

    bool success = vanaPoolTreasury.transferVana(payable(_msgSender()), exactVanaAmount);
    if (!success) {
        revert TransferFailed();
    }

    emit Unstaked(entityId, _msgSender(), exactVanaAmount, shareAmount);
}
```

However, the `vanaToEntityShare(...)` rounds down the result. As dividing by a smaller value results in a larger outcome, this leads to the user receiving more tokens than they should.

```
function vanaToEntityShare(uint256 entityId) external view override returns (uint256) {
    Entity storage entity = _entities[entityId];

    return entity.activeRewardPool > 0 ? (entity.totalShares * 1e18) / entity.activeRewardPool : 1e18;
}
```

For example, if `activeRewardPool` is `43197679269104491383` and `totalShares` is `3499999999999991547`, consider the following exploit scenario:

- Bob stakes `640854177891528377162` VANA tokens and receives `51923845460479575293` shares;
- Bob then unstakes all his shares (`51923845460479575293` shares)and receives `640854177891528385059` VANA tokens;

In this case, Bob receives an extra `7897` VANA tokens due to the rounding error.

Recommendation(s): Consider using `entityShareToVana(...)` function instead of `vanaToEntityShare(...)` for converting shares to VANA tokens. This can be done by multiplying the share amount by the result of `entityShareToVana(...)` function.

Status: Fixed.

Update from the client: Fixed in commit 2177752

6.9 [Medium] Unauthorized execution of `migrateStakeToVanaPool(...)` on behalf of others

File(s): DLPRootImplementation.sol.sol

Description: The `DLPRootImplementation` contract introduces the `migrateStakeToVanaPool(...)` function, which allows users to migrate their existing DLP stakes to the VANA pool. This function stakes the user's existing amount plus any claimable rewards into the pool for a specific entity.

```
function migrateStakeToVanaPool(uint256 stakeId, uint256 entityId) external payable nonReentrant whenNotPaused {
    dlpRootEpoch.createEpochs();

    Stake storage stake = _stakes[stakeId];

    if (stake.endBlock == 0) {
        stake.movedAmount = stake.amount;
        _closeStake(_msgSender(), stakeId);
        stake.movedAmount = 0;
    }

    uint256 toClaim = stake.amount + _calculateStakeRewardUntilEpoch(stakeId, dlpRootEpoch.epochsCount() - 1, true);

    if (toClaim == 0) {
        revert NothingToClaim();
    }

    bool success = dlpRootRewardsTreasury.transferVana(payable(address(this)), toClaim);
    if (!success) {
        revert TransferFailed();
    }

    vanaPoolStaking.stake{value: toClaim}(entityId, stake.stakerAddress);

    emit StakeMigratedToVanaPool(stakeId, stake.amount, entityId);
}
```

However, it can be executed by anyone for any `stakeId`. Thus, allowing anyone to migrate stake of other users to any entity.

Recommendation(s): Consider adding a check to ensure that the `migrateStakeToVanaPool(...)` can only be executed by the owner of the `stakeId`.

Status: Fixed.

Update from the client: Fixed in commit 2177752

6.10 [Low] Excessive locked rewards are not handled upon entity removal

File(s): VanaPoolEntityImplementation

Description: The VanaPoolEntityImplementation contract allows anyone to add rewards to be locked and distributed to stakers who have allocated their funds to a specific entity. The processRewards(...) function is periodically called to distribute these rewards. However, when an entity is removed via the removeEntity(...) function, the contract does not check for excessive locked rewards:

```
function removeEntity(uint256 entityId) external override whenNotPaused nonReentrant onlyEntityOwner(entityId) {
    Entity storage entity = _entities[entityId];

    if (entity.status != EntityStatus.Active) {
        revert InvalidEntityStatus();
    }

    // Process any pending rewards
    processRewards(entityId);
       entity.status = EntityStatus.Removed;
    _activeEntityIds.remove(entityId);

    emit EntityStatusUpdated(entityId, EntityStatus.Removed);
}
```

The entity may have an excess rewards that are not processed yet, and since there is no mechanism to handle any remaining undistributed rewards, these funds become permanently locked within the treasury contract once the entity is removed.

Recommendation(s): Consider moving any exessive rewards when removing an entity.

Status: Acknowledged

Update from the client: We are aware of the issue and we didn't decide yet what to do with the excessive rewards.

6.11 [Low] removeEntity(...) does not clear entityNameToId

File(s): VanaPoolEntityImplementation

Description: When creating or updating an entity, the contract checks that the entity name isn't already used by verifying that the mapping value is zero. However, when an entity is removed via removeEntity(...), the mapping entry entityNameToId is not cleared:

```
function removeEntity(uint256 entityId) external override whenNotPaused nonReentrant onlyEntityOwner(entityId) {
    Entity storage entity = _entities[entityId];

    if (entity.status != EntityStatus.Active) {
        revert InvalidEntityStatus();
    }

    // Process any pending rewards
    processRewards(entityId);
       entity.status = EntityStatus.Removed;
    _activeEntityIds.remove(entityId);

    emit EntityStatusUpdated(entityId, EntityStatus.Removed);
}
```

Since the mapping entry for entityNameToId is not cleared, the removed entity's name remains blocked from future use. This can be exploited by malicious users to monopolize desirable entity names by repeatedly creating and removing entities, thereby preventing legitimate users from registering those names.

Recommendation(s): Consider clearing the entityNameToId upon entity removal.

Status: Acknowledged.

Update from the client: This is an intended behavior.

6.12 [Info] Inconsistent check for minimum stake amount

File(s): VanaPoolStakingImplementation.sol

Description: The `stake(...)` function ensures that the amount staked is greater than or equal to a minimum threshold (`minStakeAmount`).

```
function stake(uint256 entityId, address recipient) external payable override nonReentrant whenNotPaused {
    // ...

    uint256 stakeAmount = msg.value;

    if (stakeAmount < minStakeAmount) {
        revert InsufficientStakeAmount();
    }

    // ...
}
```

However, the `unstake(...)` function does not have a similar check to ensure that after unstaking, the remaining stake is still above the minimum stake amount. This could result in a scenario where users stake an amount above the minimum, but after unstaking a portion, their remaining stake falls below the required minimum threshold.

Additionally, if a user had staked an amount above the minimum threshold and later tries to stake an amount smaller than the minimum, the transaction will revert, even though the user's total stake is still above the minimum.

Recommendation(s): Consider updating the check for the minimum stake amount in the `stake(...)` function to ensure it properly enforces the minimum threshold. Additionally, add a similar check in the `unstake(...)` function to ensure that users cannot have a stake less than the minimum required amount, unless they are unstaking all of their shares. This will help maintain consistency between the two functions and prevent users from inadvertently reducing their stake below the minimum threshold.

Status: Fixed.

Update from the client: Fixed by removing restrictions from stakers to stake below minStakeAmount in commit 97de94d

6.13 [Info] Rounding issue in the `unstake(...)` function allows owner to unstake below `minRegistrationStake`

File(s): VanaPoolStakingImplementation.sol

Description: The `unstake(...)` function allows users to unstake a specified amount of shares and transfer equivalent amount of VANA tokens to the user. It also ensures that entity owner retains at least the `minRegistrationStake` remaining after unstaking.

```
    function unstake(uint256 entityId, uint256 shareAmount) external override nonReentrant whenNotPaused {
        // ...

        // Get entity info from VanaPoolEntity
        IVanaPoolEntity.EntityInfo memory entityInfo = vanaPoolEntity.entities(entityId);

        // @audit `vanaToEntityShare(...)` rounds down the result
        uint256 vanaToShare = vanaPoolEntity.vanaToEntityShare(entityId);

        // ...

        // If this is the entity owner, ensure they can't unstake below the registration stake
        if (entityInfo.ownerAddress == _msgSender()) {
            // @audit This results in an underestimated value
            uint256 ownerMinShares = (vanaToShare * vanaPoolEntity.minRegistrationStake()) / 1e18;

            if (stakerShares - shareAmount < ownerMinShares) {
                revert CannotRemoveRegistrationStake();
            }
        }

        // ...
    }
```

The function calculates the minimum shares that the owner should retain after unstaking by converting `minRegistrationStake` (in VANA tokens) to shares using the result of `vanaToEntityShare(...)`. However, because `vanaToEntityShare(...)` rounds down the result, the calculated `ownerMinShares` is less than the actual required amount. This allows the owner to unstake more shares than intended, leaving them with fewer shares than required to meet the `minRegistrationStake` value.

Recommendation(s): Consider rounding up the result of `vanaToShare` when calculating `ownerMinShares`.

Alternatively, consider converting the remaining shares the owner will have after unstaking to VANA tokens by multiplying them with the result of `entityShareToVana(...)` and comparing that value with the `minRegistrationStake`.

Status: Fixed.

Update from the client: Fixed in commit 97de94d

6.14 [Info] `migrateStakeToVanaPool(...)` doesn't need to be payable

File(s): DLPRootImplementation.sol

Description: The `migrateStakeToVanaPool(...)` function is designed to migrate stakes from the DLP system to the VANA pool. However, the function is marked as payable unnecessarily, even though it does not require native tokens to be sent.

```
function migrateStakeToVanaPool(uint256 stakeId, uint256 entityId) external payable nonReentrant whenNotPaused {}
```

Recommendation(s): Consider removing the `payable` keyword from the function declaration.

Status: Fixed.

Update from the client: Fixed in commit 97de94d

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 the VANA Protocol documentation

The VANA team has provided a comprehensive walkthrough of the project. Moreover, the VANA team addressed all questions and concerns raised by the Nethermind Security team, providing valuable insights and a comprehensive understanding of the project's technical aspects.

8 Test Suite Evaluation

8.1 Compilation Output

```
$ npx hardhat compile
Warning: SPDX license identifier not provided in source file. Before publishing, consider adding a comment containing
 ↪  "SPDX-License-Identifier: <SPDX-License>" to each source file. Use "SPDX-License-Identifier: UNLICENSED" for
 ↪  non-open-source code. Please see https://spdx.org for more information.
--> contracts/rootMetrics/interfaces/ERC2771ContextUpgradeableMock.sol


Warning: Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.24;"
--> contracts/rootMetrics/interfaces/ERC2771ContextUpgradeableMock.sol


Warning: This declaration has the same name as another declaration.
   --> contracts/rootMetrics/DLPRootMetricsImplementation.sol:331:9:
    |
331 |         address payable foundationWalletAddress
    |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Note: The other declaration is here:
  --> contracts/rootMetrics/DLPRootMetricsImplementation.sol:68:5:
   |
68 |     function foundationWalletAddress() external view override returns (address payable) {
   |     ^ (Relevant source part starts here and spans across multiple lines).


Warning: This declaration shadows an existing declaration.
   --> contracts/rootMetrics/DLPRootMetricsImplementation.sol:472:14:
    |
472 |         for (uint256 i = 0; i < topDlps.length; ) {
    |              ^^^^^^^^^
Note: The shadowed declaration is here:
   --> contracts/rootMetrics/DLPRootMetricsImplementation.sol:469:9:
    |
469 |         uint256 i;
    |         ^^^^^^^^^


Warning: This declaration shadows an existing declaration.
   --> contracts/rootMetrics/DLPRootMetricsImplementation.sol:461:9:
    |
461 |         IDLPRootMetrics.DlpRating[] memory topDlps = topDlpsCustomized(
    |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
   --> contracts/rootMetrics/DLPRootMetricsImplementation.sol:171:5:
    |
171 |     function topDlps(uint256 numberOfDlps) public view override returns (DlpRating[] memory) {
    |     ^ (Relevant source part starts here and spans across multiple lines).


Warning: This contract has a payable fallback function, but no receive ether function. Consider adding a receive ether
 ↪  function.
  --> contracts/l1Deposit/DepositProxy2.sol:13:1:
   |
13 | contract DepositProxy2 is Proxy {
   | ^ (Relevant source part starts here and spans across multiple lines).
Note: The payable fallback function is defined here.
  --> @openzeppelin/contracts/proxy/Proxy.sol:66:5:
   |
66 |     fallback() external payable virtual {
   |     ^ (Relevant source part starts here and spans across multiple lines).


Warning: Return value of low-level calls not used.
  --> contracts/multisend/MultisendImplementation.sol:57:13:
   |
57 |             recipients[i].call{value: amount}("");
   |             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
Warning: Return value of low-level calls not used.
  --> contracts/multisend/MultisendImplementation.sol:78:13:
   |
78 |             recipients[i].call{value: amounts[i]}("");
   |             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^


Warning: Since the VM version paris, "difficulty" was replaced by "prevrandao", which now returns a random number based
 ↪  on the beacon chain.
   --> contracts/multicall3/Multicall3.sol:199:22:
    |
199 |         difficulty = block.difficulty;
    |                      ^^^^^^^^^^^^^^^^


Generating typings for: 134 artifacts in dir: typechain-types for target: ethers-v6
Successfully generated 388 typings!
Compiled 132 Solidity files successfully (evm target: paris).
```

8.2 Tests Output

```
$ npx hardhat test test/vanaPool.ts

  VanaPool
    Setup
       should have correct initial values
       should pause and unpause when maintainer
       should reject pause and unpause when non-maintainer
       should update minStakeAmount when maintainer
       should reject updateMinStakeAmount when non-maintainer
       should update trustedForwarder when maintainer
       should reject updateTrustedForwarder when non-maintainer
    Entity Management
       should create an entity successfully
       should reject entity creation with same name
       should reject entity creation with name too short
       should reject entity creation with wrong registration stake
       should update entity information successfully
       should reject entity update when not owner
       should remove entity successfully
       should reject entity removal when not owner
    Staking Operations
       should stake successfully in an entity for self
       should stake successfully in an entity for another user
       should reject stake with invalid recipient
       should reject stake with amount below minimum
       should reject stake for non-existent entity
       should unstake successfully from an entity
       should reject unstake with zero amount
       should reject unstake for stakes user doesn't own
       should prevent entity owner from unstaking below registration stake
       should allow entity owner to unstake above registration stake
       should handle share to VANA conversion
    Rewards
       should add rewards to an entity successfully
       should reject adding zero rewards
       should process rewards correctly
       should update entity maxAPY
       should reject updateEntityMaxAPY when non-admin
       should use rewards to increase share value
    Complex Scenarios
       should handle multiple entities with stakers and rewards
       should handle entity removal with rewards
       should handle share/VANA conversions with rewards accrual
    Treasury Operations
       should receive VANA in treasury
       should transfer VANA from treasury when authorized
       should reject treasury transfer when non-admin
       should reject treasury transfer when paused
```

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.