hashlock.

# Security Audit

## r/datadao (DAO)

August 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

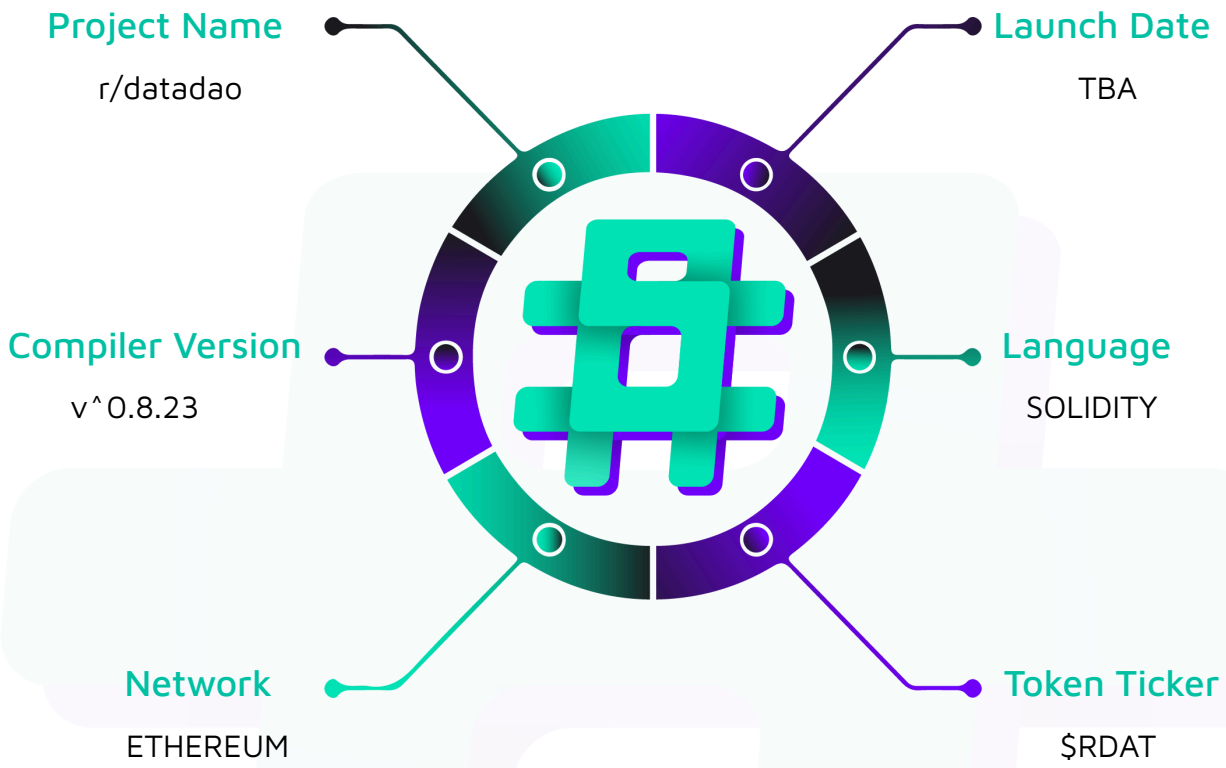
The r/datadao team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

# Project Context

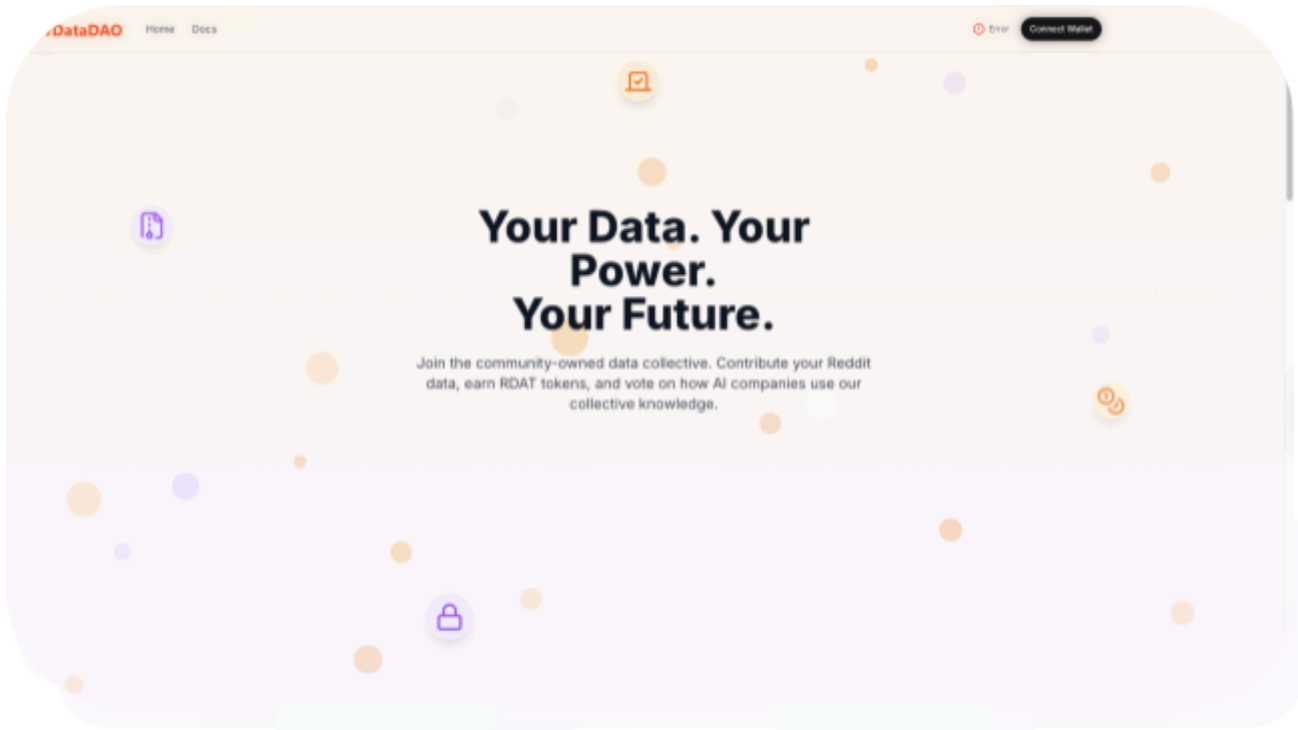
DataDAO is a decentralized community project where Reddit users contribute their data in exchange for $RDAT governance tokens. These tokens let members collectively decide how the data is used — from monetization with AI companies to negotiating with Reddit itself. Built on the Vana Network, the project focuses on user ownership, fair compensation, and transparent community governance.

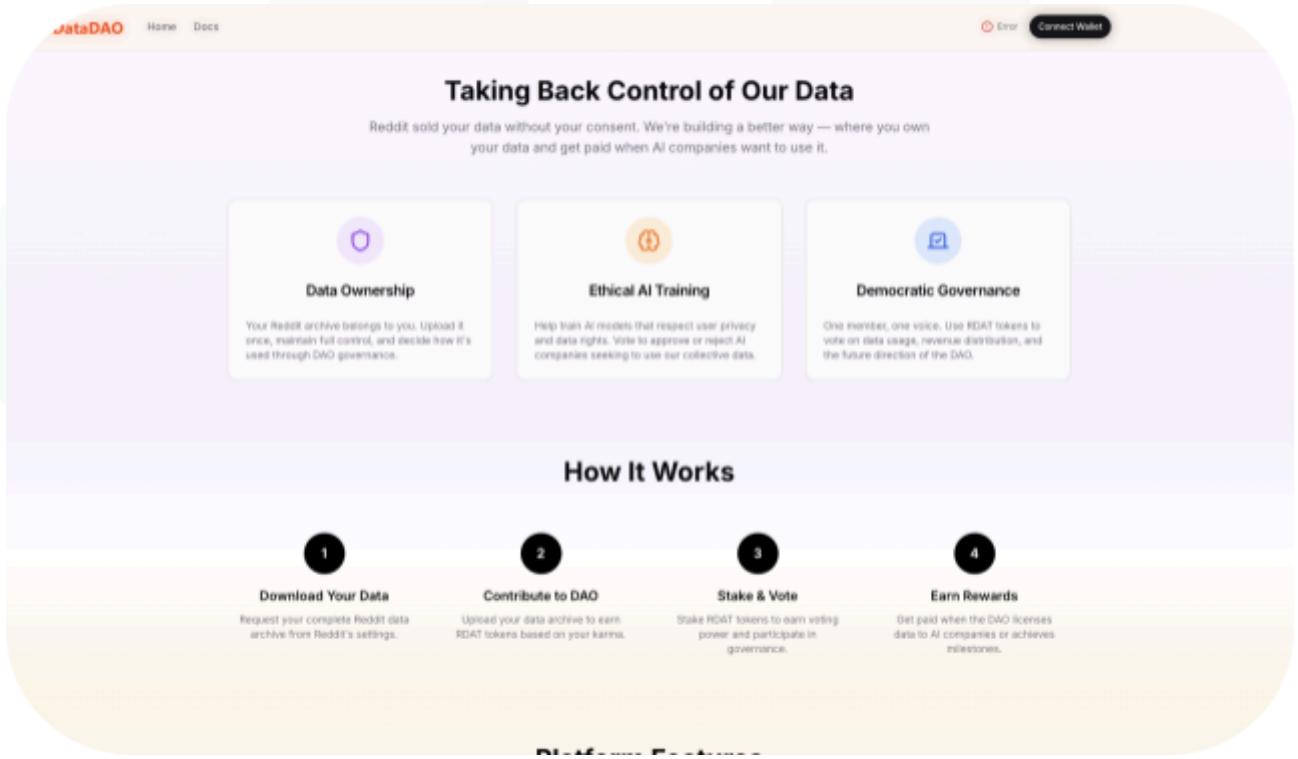
**Project Name**: r/datadao
**Project Type**: DAO
**Compiler Version**: ^0.8.23
**Website**: https://www.rdatadao.org/
**Logo**:

**Visualised Context:**



**Project Visuals:**



# Audit Scope

We at Hashlock audited the solidity code within the r/datadao 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** | **r/datadao Smart Contracts** |
|---|---|
| **Platform** | **Ethereum / Solidity** |
| **Audit Date** | **August, 2025** |
| **Contract 1** | RDATUpgradeable.sol |
| **Contract 1 MD5 Hash** | BaseMigrationBridge.sol |
| **Contract 2** | VanaMigrationBridge.sol |
| **Contract 2 MD5 Hash** | StakingPositions.sol |
| **Audited GitHub Commit Hash** | 7dcd7c1bb893927b7d88bcc05f37a3d2cfdbdd2c |
| **Fix Review GitHub Commit Hash** | 1967b164841ed3cfad893f3a72fa1d8783d1f9e2 |

# 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:**

2 High severity vulnerabilities

4 Medium severity vulnerabilities

7 Low severity vulnerabilities

3 Gas Optimisations

**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 |
|---|---|
| **RDATUpgradeable.sol**<br><br>Allows users to:<br><br>• Transfer, approve, and burn RDAT tokens.<br>• Create data pools and contribute data to them.<br>• Claim epoch rewards based on their contributions.<br>• Process payments for licensing data.<br>• Verify data ownership.<br><br>Allows admins to:<br><br>• Pause and unpause all token transfers.<br>• Manage VRC-20 compliance settings (e.g., PoC, Data Refiner, DLP).<br>• Set and fund epoch rewards for data contributors.<br>• Manage a blacklist of addresses to restrict transfers.<br>• Upgrade the contract to a new implementation. | **Contract achieves this functionality.** |
| **BaseMigrationBridge.sol**<br><br>Allows users to:<br><br>• Initiate the migration process by burning their V1 RDAT tokens.<br>• Check their total amount of burned V1 tokens.<br>• View the time remaining until the migration | **Contract achieves this functionality.** |

| | |
|---|---|
| deadline.<br><br>Allows admins to:<br><br>• Pause and unpause the migration bridge functionality.<br>• Rescue RDAT or other tokens mistakenly sent to the contract after the migration period ends.<br>• Manage contract roles for pausing and administration. | |
| **VanaMigrationBridge.sol**<br><br>Allows users to:<br><br>• Receive V2 RDAT tokens on the Vana chain after their migration is validated.<br>• Receive a migration bonus, distributed through a separate vesting contract.<br>• Query the status of their migration requests.<br><br>Allows validators to:<br><br>• Submit validation for migration requests initiated on the Base chain.<br>• Challenge suspicious migration requests to prevent fraud.<br><br>Allows admins to:<br><br>• Manage the list of approved validators.<br>• Pause and unpause the bridge during emergencies.<br>• Update the daily migration limit.<br>• Return any unclaimed tokens to a treasury | **Contract achieves this functionality.** |

| | |
|---|---|
| address after the migration period. | |
| **StakingPositions.sol**<br><br>Allows users to:<br><br>• Stake RDAT for various lock periods to create an NFT representing their position.<br>• Unstake their RDAT and burn the position NFT after the lock period has expired.<br>• Perform an emergency withdrawal before the lock period ends, which incurs a penalty.<br>• Transfer their position NFT to another address once it is unlocked and rewards are handled.<br><br>Allows admins to:<br><br>• Pause and unpause all staking and unstaking functionality.<br>• Update the vRDAT multipliers for different lock durations.<br>• Set the address for the RewardsManager contract to handle reward logic.<br>• Rescue non-RDAT tokens that have been accidentally sent to the contract. | **Contract achieves this functionality.** |

# Code Quality

This audit scope involves the smart contracts of the r/datadao 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 r/datadao 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

## High

### [H-01] StakingPositions#emergencyWithdraw - Penalty and Revenue Funds Trapped in StakingPositions Contract

**Description**

The `StakingPositions` contract accumulates RDAT tokens from two sources: penalties on emergency withdrawals and direct deposits of revenue rewards. The contract currently lacks a mechanism to withdraw these specific tokens, causing them to become inaccessible after they are collected.

**Vulnerability Details**

The contract is designed to collect a 50% penalty in RDAT when a user executes an emergencyWithdraw. The code comment `// Penalty stays in contract for treasury rescue` indicates these funds are intended for the treasury. However, the rescueTokens function, which would normally be used for this, explicitly prevents the withdrawal of the RDAT token itself.

```
// src/StakingPositions.sol:422

function rescueTokens(address token, uint256 amount) external override onlyRole(ADMIN_ROLE) {

    require(token != address(_rdatToken), "Cannot rescue RDAT");

    IERC20(token).safeTransfer(msg.sender, amount);

}
```

Similarly, the `notifyRewardAmount` function allows a `REVENUE_COLLECTOR_ROLE` to transfer RDAT into the contract to be distributed as rewards. These funds are added to the `pendingRevenueRewards` state variable, but the contract lacks any function to actually distribute them to stakers. The `claimRewards` functions are deprecated and revert, and

no alternative withdrawal or distribution mechanism exists. Consequently, both penalty and revenue RDAT tokens are permanently locked within the contract address.

**Impact**

This issue affects the intended economic flow of the protocol. Funds designated for the treasury from penalties and funds intended for distribution to stakers as revenue do not reach their destinations.

**Recommendation**

It is recommended to introduce a new, role-restricted function that permits a designated admin or treasury address to withdraw RDAT tokens from the contract.

**Status**

Resolved

## [H-02]VanaMigrationBridge#challengeMigration - Single Validator Can Permanently Block Migrations via Irreversible Challenge

### Description

The current challenge mechanism allows a single validator to irreversibly block a migration request. There is no process to review or overturn a challenge, which can permanently prevent a user's migration from being completed.

### Vulnerability Details

Any address with the `VALIDATOR_ROLE` can call `challengeMigration(requestId)` at any time after a migration has at least one validation. This function unconditionally sets the challenged flag for the request to true.

```
// src/VanaMigrationBridge.sol:171-179

function challengeMigration(bytes32 requestId) external override onlyRole(VALIDATOR_ROLE)
whenNotPaused {

    // ...

    request.challenged = true;

    // ...

}
```

Once this flag is set to true, the `_executeMigration` function will always revert, as it contains a check that stops execution for challenged requests. The contract does not provide a method to reset this flag, making the block permanent.

### Impact

This design allows a single validator to unilaterally prevent a user from completing their migration, effectively halting their transaction.

### Recommendation

The challenge mechanism should be redesigned to prevent this permanently locked state

**Status**

Resolved

# Medium

## [M-01] BaseMigrationBridge#initiateMigration - Migrated V1 Tokens Are Held in Contract Instead of Burned

### Description

The BaseMigrationBridge contract collects V1 RDAT tokens from users and holds them within the contract. According to project documentation, these tokens are intended to be burned. The current implementation allows an admin to potentially withdraw these collected tokens after the migration period has ended.

### Vulnerability Details

Project documentation, specifically in `docs/archive/MIGRATION_ARCHITECTURE.md`, indicates that the V1 migration process includes burning the V1 tokens. However, the `initiateMigration()` function in the BaseMigrationBridge.sol contract implements a transfer of the tokens to the contract itself, rather than performing a burn. A burn operation would typically involve transferring tokens to the `address(0)`:

```
// src/BaseMigrationBridge.sol:75

v1Token.safeTransferFrom(msg.sender, address(this), amount);
```

The `rescueTokens` function permits an address with the `DEFAULT_ADMIN_ROLE` to withdraw tokens held by the contract after the migration deadline.

### Impact

The current implementation creates a large pool of V1 tokens in the bridge contract, which could be at risk if the admin account is compromised. Since the tokens are not burned, they could also be reintroduced into circulation, potentially affecting the V2 tokenomics.

**Recommendation**

It is recommended to modify initiateMigration to transfer tokens to the burn address (`address(0)`) to align with the documentation. If the current behavior is intended, the project documentation should be updated to explain that the tokens are held by the contract and describe their intended purpose.

**Status**

Resolved

### [M-02] StakingPositions#_update - Staking Position NFTs Are Non-Transferable, Contradicting Intended Design

**Description**

The contract intends for staking position NFTs to be transferable after their lock period has expired. However, the current transfer requirements within the code create a logical conflict that makes the NFTs non-transferable until they are burned via unstaking or emergency withdrawal.

**Vulnerability Details**

The `_update` function, which handles transfers, requires a position to be both unlocked and to have no active `vRDAT` rewards associated with it. A check inside the function reverts any transfer if the `vrdatMinted` amount is greater than zero.

```
// src/StakingPositions.sol:363

if (position.vrdatMinted > 0) {

    revert TransferWithActiveRewards();

}
```

However, a normal staking position always has a `vrdatMinted` value. The only function that sets this value to zero, emergencyWithdraw, also burns the NFT in the same transaction. This means there is no possible state in which an NFT is mature and has a `vrdatMinted` value of zero, so the conditions for a transfer can never be met.

**Impact**

This issue removes the intended utility of the NFTs as transferable assets after they mature.

**Recommendation**

It is recommended to adjust the validation logic in the `_update` function.

**Status**

Resolved

### [M-03] RDATUpgradeable#createDataPool - Front-Running Vulnerability in createDataPool Function

**Description**

The `createDataPool` function allows a user to specify the poolId. This makes it possible for an observer to see a pending transaction and use the same `poolId` to create the pool first by paying a higher transaction fee.

**Vulnerability Details**

Because the `poolId` is provided as an external parameter, a user monitoring pending transactions can see an upcoming call to `createDataPool`. They can then copy the `poolId` and submit their own transaction with a higher fee, which would likely be confirmed first. This would make them the `creator` of the pool, causing the original user's transaction to fail on the uniqueness check.

```
// src/RDATUpgradeable.sol:231

require(_dataPools[poolId].creator == address(0), "Pool already exists");
```

**Impact**

This issue allows an attacker to prevent users from creating data pools with their intended identifiers. This can be used to disrupt the protocol's functionality by griefing users.

**Recommendation**

Consider generating the poolId within the contract instead of accepting it as a user-supplied parameter.

**Status**

Resolved

## [M-04] VanaMigrationBridge#challengeMigration - Challenges Can Be Submitted After the Challenge Period Ends

**Description**

The `challengeMigration` function does not check if the transaction is within the challenge period. This allows a validator to submit a challenge even after the challengeEndTime for a request has passed, which is inconsistent with the idea of a fixed window for challenges.

**Vulnerability Details**

When a migration request is first created, a `challengeEndTime` is set to define a window for challenges. However, the `challengeMigration` function does not validate against this timestamp and can be called at any point before the request is executed.

```
// src/VanaMigrationBridge.sol:171-179

function challengeMigration(bytes32 requestId) external override onlyRole(VALIDATOR_ROLE)
whenNotPaused {

    MigrationRequest storage request = _migrationRequests[requestId];

    // No check ensures block.timestamp < request.challengeEndTime

    if (request.validatorApprovals == 0) revert InvalidRequest();

    if (request.executed) revert AlreadyProcessed();

    if (request.challenged) revert AlreadyProcessed();



    request.challenged = true;
```

```
    emit MigrationChallenged(requestId, msg.sender);

}
```

**Impact**

This behavior undermines the assurance provided by the challenge period. It creates a condition where a migration request is never truly "safe" from being challenged. If a challenge is submitted after the period ends, it can still block the migration.

**Recommendation**

It is recommended to enforce the intended time window within the `challengeMigration` function.

**Status**

Resolved

# Low

## [L-01] Contracts - rescueTokens Function Lacks Event Emission

### Description

The `rescueTokens()` function in BaseMigrationBridge.sol allows an admin to withdraw tokens, a critical action involving the movement of assets. However, the function does not emit an event upon execution. This lack of event emission makes it difficult to track and monitor these administrative actions off-chain.

### Recommendation

It is recommended to emit an event within the `rescueTokens` function to log the key details of the transfer.

### Status

Resolved

## [L-02] Contracts - Single Address Holds Multiple Roles

### Description

A system-wide pattern was observed where multiple critical roles, including administrative and emergency pausing functions, are assigned to a single admin address across key contracts. This design centralizes significant power. If this address is a single key, its compromise creates a single point of failure. Alternatively, if it is a multisig, the coordination required to execute a transaction could delay a time-sensitive emergency pause.

### Recommendation

To mitigate both potential risks, it is recommended to separate these duties..

### Status

Resolved

### [L-03] Contracts - Misleading Comment Regarding Staking Position Limits

**Description**

The contract comment for StakingPositions.sol states that users can have "Unlimited concurrent stakes," but the implementation enforces a limit of 100 positions per user. This creates a discrepancy between the documentation and the actual behavior of the contract.

**Recommendation**

Update the comment to accurately reflect the limit on concurrent stakes.

**Status**

Resolved

### [L-04] Contracts - Timelock Mechanism Not Enforced for Critical Operations

**Description**

The RDATUpgradeable contract includes a timelock mechanism with `scheduleTimelock` and `executeTimelock` functions, but this system is not integrated with any administrative actions/functions. Critical functions only perform role-based checks and can be executed immediately, which does not align with the documented security design requiring a 48-hour delay for such operations. The current implementation only logs that a timelock was "executed" but does not enforce a delay on any function.

**Recommendation**

It is recommended to integrate a timelock mechanism that properly gates the execution of sensitive functions.

**Status**

Resolved

## [L-05] Contracts - Incomplete pending rewards accounting in setEpochRewards

**Description**

`The totalPendingRewards` in `setEpochRewards()` function is hardcoded to `0`, so the check only validates `balanceOf(this) >= amount`. Previously allocated but unclaimed rewards are not considered, which can allow over-allocation and cause later claims to fail.

**Recommendation**

Track total unclaimed rewards across epochs and require `balance >= pending + amount`.

**Status**

Resolved

## [L-06] Contracts - Misnamed revert used when the request is challenged

**Description**

`_executeMigration()` function in VanaMigrationBridge.sol, the code reverts with NotChallenged when `request.challenged` is true, which is logically inverted and misleading.

**Recommendation**

Replace `NotChallenged` with a correctly named error (e.g., `Challenged` or `MigrationChallenged`).

**Status**

Resolved

## [L-07] Contracts - Missing Critical Event Emissions

### Description

Important administrative functions in VanaMigrationBridge.sol that change critical contract settings and execute large fund transfers are missing on-chain events. This reduces transparency and makes security monitoring difficult.

### Recommendation

Add event declarations and emit them within the `setBonusVesting` and `returnUnclaimedTokens` functions to ensure these significant actions are logged on-chain.

### Status

Resolved

# QA

## [Q-01] StakingPositions - Redundant checks in deprecated claimRewards function

### Description

The `claimRewards` function contains checks for position existence and ownership. As the function's only behavior is to revert with a message directing users to the `RewardsManager`, these preceding checks are redundant.

### Recommendation

Consider removing the redundant checks.

### Status

Acknowledged

## [Q-02] StakingPositions - `calculatePendingRewards()` and `getUserTotalRewards()` functions returns a misleading value of 0 instead of reverting

### Description

The `calculatePendingRewards()` and `getUserTotalRewards()` functions are deprecated, as the contract's reward logic has been moved to a `RewardsManager`. The function currently calls `_calculateRewards`, which is hardcoded to return `0`. This can mislead users or front-ends into believing they have no pending rewards, when in fact their rewards are accumulating in a different contract. This behavior is inconsistent with the deprecated `claimRewards` function, which correctly reverts with an informative message.

### Recommendation

For consistency and to avoid user confusion, modify these functions to revert with a message.

### Status

Acknowledged

### [Q-03] StakingPositions - Position Data Not Deleted on Emergency Withdraw

**Description**

The `emergencyWithdraw` function correctly burns the user's NFT but fails to delete the associated position data from the `_positions` mapping. This leaves zombie data in the contract's storage, leading to potential inconsistencies in the future.

**Recommendation**

To ensure state consistency, delete _positions[positionId];.

**Status**

Acknowledged

# Centralisation

The r/datadao project values security and utility over decentralisation.

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

# Conclusion

After Hashlock's analysis, the r/datadao 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