hashlock.

Security Audit

Narra Layer (AI Agent)

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

Table of Contents

CAUTION

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

Executive Summary

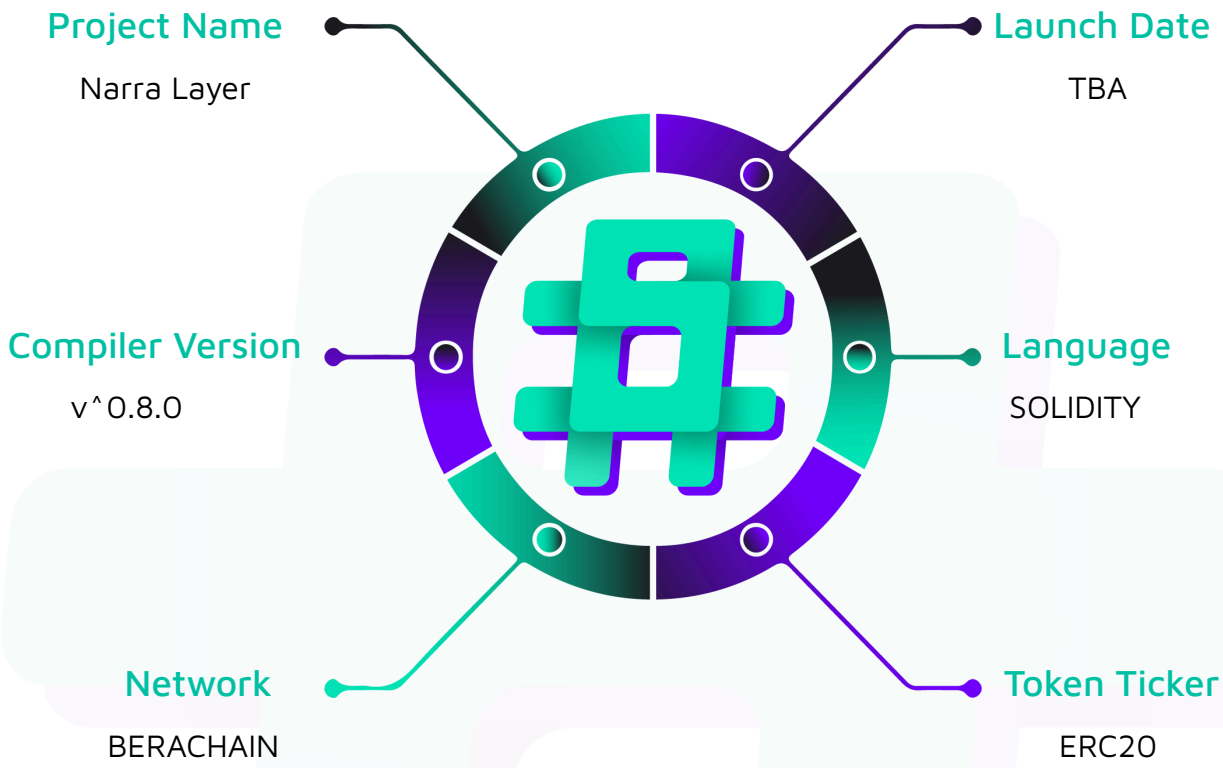
The Narra Layer team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

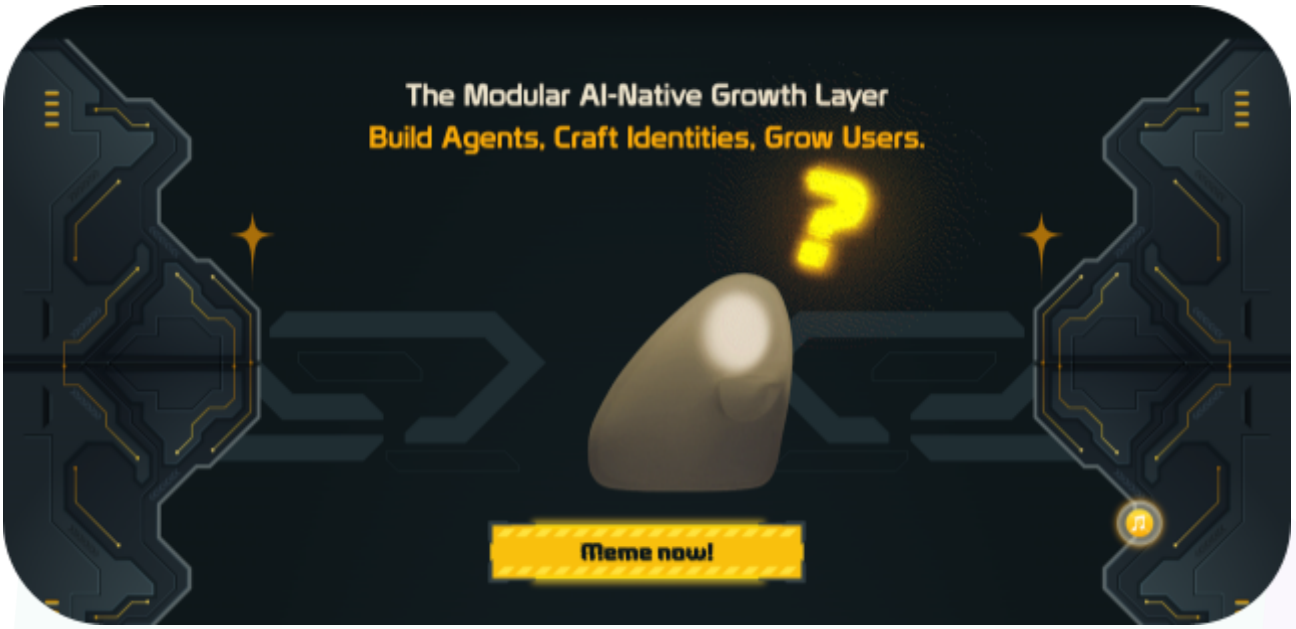
NarraLayer creates custom anime-style AI agents that interact with users across both Web2 platforms and Web3 environments. These agents drive community engagement through gamified experiences, like Tamagotchi-style interfaces, and can be used for education, yield optimization, user onboarding, and social activation. The platform is modular and no-code, making it easy for Web3 protocols to integrate these characters and boost user engagement and retention.

Project Name: Narra Layer
Project Type: Gamefi, AI Agent.
Compiler Version: ^0.8.0
Website: www.narralayer.ai
Logo:

Visualised Context:



Project Visuals:



Audit Scope

We at Hashlock audited the solidity code within the Narra Layer 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	**Narra Layer Smart Contracts**
Platform	**Ethereum / Solidity**
Audit Date	**July, 2025**
Contract 1	StakingToken.sol
Contract 2	NarraLayerVault.sol
Audited GitHub Commit Hash	0589e04180432329e460022887e49e0c2401ae6c
Fix Review GitHub Commit Hash	

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Vulnerable"**. 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 we have identified have yet to be resolved or acknowledged.

Hashlock found:

1 Medium severity vulnerability

3 Low severity vulnerabilities

1 QA

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

Intended Smart Contract Functions

Claimed Behaviour	Actual Behaviour
StakingToken.sol - The ERC20 standard allows only the owner to mint and burn tokens.	**Contract achieves this functionality.**
NarraLayerVault.sol - Allows users to: - Burn tokens via burnToStake() to earn staking rewards - Receive receipts for tracking their staking positions - Earn rewards through a delegated staking mechanism - Allows admins to: - Add/remove supported tokens with custom weights - Set cooldown periods and cleanup parameters - Clean expired stakes manually or automatically - Upgrade contract implementation	**Contract achieves this functionality.**

Code Quality

This audit scope involves the smart contracts of the Narra Layer 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 Narra Layer project smart contract code in the form of GitHub access.

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

Dependencies

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

Severity Definitions

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

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

Status Definitions

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

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

Audit Findings

Medium

[M-01] NarraLayerVault - Misleading NatSpec Comments

Description

Since the purpose of the Ethereum Natural Specification (`NatSpec`) is to describe the code to the user, misleading statements should be considered a violation of the public API that may confuse or mislead users.

Vulnerability Details

There is a discrepancy between the `NatSpec` comment and the actual implementation for functions such as `updateSupportedToken`, `burnToStake`, `clearStaking`, and `cleanExpiredStakes`.

The comment for `updateSupportedToken` states that the weight has a precision of 1e4, although the comment for the `supportedTokens` mapping states that the precision is 1e18.

```
    mapping(address => uint256) public supportedTokens; // weight 精度 1e18

     //...code

     /**
     * @notice Update the weight per token for a supported token.
     *
     * @param token The address of the token to update (must be a valid ERC20 token)
=>    * @param _weightPerToken The new weight per token (multiplied by 1e4, e.g., 10000 =
1.0)
     *
     * @notice This function:
     *         1. Can only be called by addresses with ADMIN_ROLE
=>   *         2. Weight is stored as a multiplier of 1e4 for precision
     *         3. Setting weight to 0 effectively removes the token
```

```
     *
     * @custom:emits SupportedTokenUpdated event
     */

    function updateSupportedToken(
        address token,
        uint256 _weightPerToken
    ) external override onlyRole(ADMIN_ROLE) {
        supportedTokens[token] = _weightPerToken;
        emit SupportedTokenUpdated(token, _weightPerToken);
    }
```

The comments for `burnToStake`, `_clearStaking`, and `cleanExpiredStakes` state that these functions can only be called by an address with `ADMIN_ROLE`.

Also, the `cleanExpiredStakes` function has a comment that `maxCount` must be greater than 0 and that the default value is 100. But since this is an external function and anyone can call it, there are no checks to ensure that this argument is greater than 0 and that the default value of 100 is used.

```
/**
     * @notice Burn tokens to stake.
     *
     * @param token The address of the token to burn
     * @param amount The amount of tokens to burn
     *
     * @notice This function:
     *         1. Can only be called by addresses with ADMIN_ROLE
     *         2. Amount must be greater than 0
     *         3. Token must be supported
     *         4. Burn token to get receipt
     *         5. Mint staking token
     *         6. Approve reward vault to spend staking token
     *         7. Delegate stake to reward vault
     *         8. Create receipt record
     *         9. Clear expired stakes
     */
    function burnToStake(address token, uint256 amount) external override {
    //...code
```

```
    }

     /**
     * @notice Clear expired stakes.
     *
     * @notice This function:
     *         1. Can only be called by addresses with ADMIN_ROLE
     *         2. Max count must be greater than 0
     *         3. Default max count is 100
     */
    function _clearStaking() internal {
    //...code
    }

     /**
     * @notice Clear expired stakes.
     *
     * @param maxCount The maximum count to clean expired stakes
     *
     * @notice This function:
     *         1. Can only be called by addresses with ADMIN_ROLE
     *         2. Max count must be greater than 0
     *         3. Default max count is 100
     */
    function cleanExpiredStakes(uint256 maxCount) external override {
    //...code
    }
```

Also, in the NatSpec comments to the contract, there is no implementation of "unstaking tokens in emergency."

```
/**
 * @title NarraLayerVault
  * @dev A vault contract supporting multiple tokens, receipt issuance, and reward claiming.
 *      Integrates with staking, reward vault, and BeraPawForge for reward minting.
 *      Uses role-based access control for admin and upgrade operations.
 *
 * @notice This contract allows users to:
```

```
 *         1. Burn supported tokens to receive receipts
 *         2. Claim rewards after cooldown period
 *         3. Admin can manage supported tokens and cooldown time
=> *         4. Admin can unstake tokens in emergency
 */
contract NarraLayerVault is
    INarraLayerVault,
    Initializable,
    UUPSUpgradeable,
    AccessControlUpgradeable
{
//...code
}
```

Impact

This inconsistency will lead to confusion among users, developers, and auditors.

Recommendation

Correct NatSpec comments in accordance with function implementations.

Status

Unresolved

Low

[L-01] Contracts - Inconsistent Pragma Solidity Versions

Description

Narra Layer contracts and interfaces use multiple versions of the Solidity compiler pragma, specifically ^0.8.0, ^0.8.19, ^0.8.26, and ^0.8.13. This inconsistency can lead to compatibility issues, as different pragma versions may have varying behavior and features.

Recommendation

It is recommended to use a consistent and up-to-date pragma version across all contracts. It is recommended to upgrade to the latest stable Solidity version.
It is also not recommended to use a floating pragma

Status

Unresolved

[L-02] NarraLayerVault#setupStakingToken - Lack of Event Emission

Description

It has been observed that some functionalities are missing from emitting events.

Events are a method of informing the transaction initiator about the actions taken by the called function. It logs its emitted parameters in a specific log history, which can be accessed outside of the contract using some filter parameters. Events help non-contract tools to track changes, and events prevent users from being surprised by changes.

```
   /**
     * @notice Setup staking token and reward vault
     * @dev This function can only be called by addresses with ADMIN_ROLE
     * @dev This function will create a new staking token and a new reward vault
     * @dev This function will set the staking token address and the reward vault address
=>   * @dev This function will emit a StakingTokenCreated event
=>   * @dev This function will emit a RewardVaultCreated event
     */
    function setupStakingToken() external override onlyRole(ADMIN_ROLE) {
        // Create new staking token
        StakingToken stakingToken = new StakingToken();
        stakingTokenAddress = address(stakingToken);

        // Create vault for newly created token
        address vaultAddress = IRewardVaultFactory(rewardVaultFactory)
            .createRewardVault(address(stakingToken));

        rewardVault = address(IRewardVault(vaultAddress));
    }
```

Recommendation

All functions updating important parameters should emit events.

Status

Unresolved

[L-03] NarraLayerVault#_cleanExpiredStakes Users will wait longer than the set cooldown.

Description

This function takes into account the distribution of payments. It iterates through unique receipt numbers using the variable `nextToCleanReceiptID`, which will increase in two cases:

1) if `receipt.cleared == true`

2) if `block.timestamp > receipt.clearedAt`

Otherwise, it breaks out of the loop.

If the cooldown changes from longer to shorter, for example, from 8 days to 7 days. If `user A` stakes tokens with a cooldown of 8 days. After that, the cooldown will change to 7 days. `User B` stakes tokens with a 7-day cooldown, then they will have to wait until `user A` receives their `StakingToken` in 8 days. Only after that will `user B` be able to receive their tokens.

```
function _cleanExpiredStakes(uint256 maxCount) internal {
     uint256 cleaned = 0;
     while (nextToCleanReceiptID < nextReceiptID && cleaned < maxCount) {
         Receipt storage receipt = receipts[nextToCleanReceiptID];
         if (receipt.cleared) {
             // Already cleared, skip to next
             nextToCleanReceiptID++;
         } else if (block.timestamp > receipt.clearedAt) {
             // Eligible for cleaning
             IRewardVault(rewardVault).delegateWithdraw(
                 receipt.user,
                 receipt.receiptWeight
             );
             receipt.cleared = true;
             cleaned++;
             nextToCleanReceiptID++;
         } else {
             // Not yet eligible, stop here
             break;
```

```
            }
        }
    }
```

Recommendation

Improve the function based on this finding.

Consider creating a function that allows users to request tokens themselves, provided that `block.timestamp > receipt.clearedAt.`

Status

Unresolved

QA

[Q-01] IPOLErrors - Typo

Description

This interface contains a Typo

```
/*                  BLOCK REWARD CONTROLLLER                  */
```

Recommendation

Fix `CONTROLLLER` to `CONTROLLER`

Status

Unresolved

Centralisation

The Narra Layer project values security and utility over decentralisation.

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

Conclusion

After Hashlock's analysis, the Narra Layer project seems to have a sound and well-tested code base; however, our findings need to be resolved to achieve full security. 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