NodeSet Constellation v1.0.1 Audit Report

Prepared by Riley Holterhus
November 22, 2024

Contents

1 Introduction **3**

1.1 About NodeSet Constellation 3

1.2 About the Auditor 3

1.3 Disclaimer 3

2 Audit Overview **4**

2.1 Scope of Work 4

2.2 Summary of Findings 4

3 Findings **5**

3.1 Medium Severity Findings 5

3.1.1 `initialize()` logic may be invoked unexpectedly in future upgrades 5

3.2 Informational Findings 6

3.2.1 `rebalanceRplVault()` prioritization considerations 6

3.2.2 `sna.bond()` return value can be cached 7

3.2.3 `queueableDepositsLimitEnabled` should be initialized in `reinitialize101()` 8

1 Introduction

1.1 About NodeSet Constellation

NodeSet Constellation is a liquid staking protocol built on top of Rocket Pool. It enables registered node operators to run validators funded by a deposit pool of ETH and RPL, with protocol actions managed by protocol admins. For more information, visit NodeSet's website: nodeset.io.

1.2 About the Auditor

Riley Holterhus is an independent security researcher that focuses on Solidity smart contracts. Other than conducting independent security reviews, he works as a Lead Security Researcher at Spearbit, and also searches for vulnerabilities in live codebases. Riley can be reached by email at rileyholterhus@gmail.com, by Telegram at @holterhus and on Twitter/X at @rileyholterhus.

1.3 Disclaimer

This report is intended to detail the identified vulnerabilities of the reviewed smart contracts and should not be construed as an endorsement or recommendation of any project, individual or entity. While the authors have made reasonable efforts to detect potential issues, the absence of any undetected vulnerabilities or issues cannot be guaranteed. Additionally, the security of the smart contracts may be affected by future changes or updates. By using the information in this report, you acknowledge that you are doing so at your own risk and that you should exercise your own judgment when implementing any recommendations or making decisions based on the findings. This report has been provided on an "as-is" basis and DOES NOT CONSTITUTE A GUARANTEE OR WARRANTY OF ANY FORM.

2 Audit Overview

2.1 Scope of Work

On November 18, 2024, Riley Holterhus conducted an audit of NodeSet's v1.0.1 Constellation smart contract upgrade. During this period, a manual analysis was undertaken to identify various security issues and logic flaws.

The audit was conducted on the codebase found in the `nodeset-org/constellation` GitHub repository, specifically reviewing PR 411. The audit started on commit `36f0651cc70ec9ed2e6ae387673aebc07dab7462`, and all changes added by the team up to commit `70fdcda15130dde8f6dfd72a3cd3c47b7f92dcac` were reviewed.

2.2 Summary of Findings

Each finding from the audit has been assigned a severity level of "Critical", "High", "Medium", "Low" or "Informational". These severities are subjective, but aim to capture the impact and feasibility of each potential issue. In total, 1 medium-severity finding and 3 informational findings were identified. All issues identified in the code have been either addressed or acknowledged. The resulting changes were reviewed, and all mitigations have been documented in this report.

3 Findings

3.1 Medium Severity Findings

3.1.1 `initialize()` logic may be invoked unexpectedly in future upgrades

Description: The following issue was noted during the review, but does not directly relate to the new changes in the v1.0.1 upgrade.

In the current codebase, many contracts inherit from the `UpgradeableBase` abstract contract, which includes a public `initialize()` function:

```
abstract contract UpgradeableBase is UUPSUpgradeable, ReentrancyGuard {
    // ...
    function initialize(address directoryAddress) public virtual onlyInitializing {
        require(directoryAddress != address(0), 'UpgradeableBase: invalid directory address');
        _directory = Directory(directoryAddress);
        __UUPSUpgradeable_init();
    }
    // ...
}
```

In the above code, the `onlyInitializing` modifier restricts this function so it can only be invoked within the execution of a function marked with `initializer()` or `reinitializer()`. However, note that this behavior also means that any function with these modifiers must carefully avoid giving up control-flow during execution. If control-flow is lost (e.g. through an external call), the public `initialize()` function could potentially be invoked by an attacker, which would overwrite the `_directory` storage variable.

In most derived contracts, this risk is mitigated because they override the `initialize()` function, which prevents the `UpgradeableBase` version of `initialize()` from being directly invoked. For example:

```
contract MerkleClaimStreamer is UpgradeableBase {
    // ...
    function initialize(address _directory) public override initializer {
        super.initialize(_directory);
        // ...
    }
    // ...
}
```

However, there are some contracts that do not override the `initialize()` function, leaving the `UpgradeableBase` implementation exposed. For example:

```
contract WETHVault is UpgradeableBase, ERC4626Upgradeable, IRateProvider {
    // ...
    function initializeVault(address directoryAddress, address weth) public virtual initializer {
```

```
        super.initialize(directoryAddress);
        // ...
    }
    // ...
}
```

Fortunately, in the current codebase, no contracts with exposed `initialize()` functions have `initializer()` or `reinitializer()` implementations that give up control-flow, so this behavior does not create any immediate vulnerabilities.

Recommendation: Since making significant changes to this logic would impact critical parts of the code, it is recommended to keep this behavior in mind for now, while continuing to avoid relinquishing control-flow in `initializer()` and `reinitializer()` functions.

In a future upgrade, consider refactoring the `UpgradeableBase` implementation to rename the `initialize()` function and mark it as `internal`.

NodeSet: Following the suggestion above, the code will remain unchanged for v1.0.1, and a GitHub issue has been created here to track this for a larger update in the future.

Auditor: Acknowledged.

3.2 Informational Findings

3.2.1 `rebalanceRplVault()` prioritization considerations

Description: A key aspect of the v1.0.1 upgrade is the rearrangement of the `processMinipool()` function to prioritize topping up RPL liquidity in the RPL vault over rebalancing RPL stake in Rocket Pool. This change is reflected in the following diff:

```
  this.rebalanceWethVault();
+ this.rebalanceRplVault();
  this.rebalanceRplStake(sna.getEthStaked());
- this.rebalanceRplVault();
```

While this adjustment generally results in more RPL liquidity being deposited into the RPL vault (if necessary), there is a niche scenario where the previous logic might have deposited more RPL into the vault. Specifically, if the `rebalanceRplStake()` function entered the `targetStake < rplStaked` case, the following withdrawal to the `OperatorDistributor` would occur:

```
function rebalanceRplStake(uint256 _ethStaked) public onlyProtocol {
    // ...
    if (targetStake < rplStaked) {
        // ...
        if (/* ... */) {
```

```
                this.unstakeRpl(excessRpl);
            }
        }
    }
```

In the new v1.0.1 logic, this RPL withdrawal happens after the RPL vault top-up logic, which may result in slightly less RPL entering the vault compared to before.

However, since this is a niche situation and subsequent calls to `processMinipool()` will handle the withdrawn RPL, this behavior does not appear to pose any issues.

Recommendation: As this behavior is not problematic, consider maintaining the current implementation with no code changes, and keep this behavior in mind for potential future optimizations.

NodeSet: Acknowledged.

Auditor: Acknowledged.

3.2.2 `sna.bond()` return value can be cached

Description: In the new `WETHVault` implementation, the `calculateQueueableDepositLimit()` function determines the amount of new ETH that can be added to pair against the current excess rETH in the Rocket Pool deposit queue. A portion of this calculation is as follows:

```
// this is the amount of ETH from constellations that can be paired with the available rETH
uint256 pairableEth = availableREth / ((32 ether - sna.bond()) / sna.bond());
```

In this code, `sna.bond()` is called twice. To improve efficiency, the result of `sna.bond()` can be cached and reused in the calculation instead.

Recommendation: Consider changing `calculateQueueableDepositLimit()` to cache the result of `sna.bond()` for the calculation:

```
function calculateQueueableDepositLimit() public view returns (uint256) {
    // ...
    // this is the amount of ETH from constellations that can be paired with the available rETH
-   uint256 pairableEth = availableREth / ((32 ether - sna.bond()) / sna.bond());
+   uint256 bond = sna.bond();
+   uint256 pairableEth = availableREth / ((32 ether - bond) / bond);
    // ...
}
```

NodeSet: Fixed in commit 690caeb.

Auditor: Verified.

3.2.3 `queueableDepositsLimitEnabled` should be initialized in `reinitialize101()`

Description: The v1.0.1 upgrade introduces two new storage variables in the `WETHVault`: `oracleUpdateThreshold` and `queueableDepositsLimitEnabled`. Given the implementation of the `WETHVault`, initializing new variables is best achieved by using a function with the `reinitializer()` modifier that atomically sets their values during an upgrade. The new code includes a `reinitialize101()` function for this purpose:

```
function reinitialize101() public reinitializer(2) {
    // This can be called on upgrade to set new values
    oracleUpdateThreshold = 88200; // 24.5 hrs in seconds
}
```

However, note that `queueableDepositsLimitEnabled` is not initialized in `reinitialize101()`. Instead, its initialization logic has been placed in the `initializeVault()` function, which has already been executed during the deployment of the previous version of the `WETHVault` and therefore will not be invoked again.

Recommendation: Consider moving the initialization of `queueableDepositsLimitEnabled` from `initializeVault()` to `reinitialize101()`:

```
function initializeVault(address directoryAddress, address weth) public virtual initializer {
    super.initialize(directoryAddress);
    ERC4626Upgradeable.__ERC4626_init(IERC20Upgradeable(weth));
    ERC20Upgradeable.__ERC20_init(NAME, SYMBOL);

    liquidityReservePercent = 0.1e18; // 10% of TVL
    maxWethRplRatio = 4e18; // 400% at start (4 ETH of xrETH for 1 ETH of xRPL)

    // default fees with 14% rETH commission mean WETHVault share returns are equal to base ETH
    staking rewards
    treasuryFee = 0.14788e18;
    nodeOperatorFee = 0.14788e18;
    mintFee = 0.0003e18; // .03% by default
    depositsEnabled = true;
-   queueableDepositsLimitEnabled = false;
}

/**
 * @notice Reinitializer function to allow updates on contract upgrades specifically related to
    oracle update threshold
 */
function reinitialize101() public reinitializer(2) {
    // This can be called on upgrade to set new values
    oracleUpdateThreshold = 88200; // 24.5 hrs in seconds
+   queueableDepositsLimitEnabled = false;
}
```

Note that with the current behavior of initializing as `false`, the behavior does not change, since `false` is the default value of unset storage.

NodeSet: Moved the logic to `reinitialize101()` in commit 33e8fac, with the initialization value updated to `true`

as well.

Auditor: Verified.