NodeSet Constellation Audit Report

Prepared by Riley Holterhus
August 30, 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 . . . 5

3 Findings **6**
- 3.1 Critical Severity Findings . . . 6
  - 3.1.1 `close()` does not clear all necessary storage . . . 6
  - 3.1.2 `processMinipool()` does not account for refund balance . . . 6
- 3.2 High Severity Findings . . . 8
  - 3.2.1 `close()` can decrement arbitrary validator counts . . . 8
  - 3.2.2 `processMinipool()` can be called on unrelated addresses . . . 9
  - 3.2.3 Validator counts are never decremented . . . 9
  - 3.2.4 Discrete jumps in vault exchange rates can be sandwiched . . . 10
  - 3.2.5 Frontrunning oracle updates will double-count ETH . . . 11
- 3.3 Medium Severity Findings . . . 11
  - 3.3.1 `getRequiredCollateral()` does not consider contract's existing balance . . . 11
  - 3.3.2 Constellation minipools are susceptible to donation attack . . . 12
- 3.4 Low Severity Findings . . . 13
  - 3.4.1 Signatures are invalidated even if `adminServerCheck == false` . . . 13
  - 3.4.2 `sendEthToDistributors()` can unnecessarily change the `_gateOpen` variable . . . 14
  - 3.4.3 Frontrunning `withdrawal_credentials` considerations . . . 15
- 3.5 Informational Findings . . . 16
  - 3.5.1 `subNodeOperatorHasMinipool` mapping can be removed . . . 16
  - 3.5.2 `getNextMinipool()` iteration isn't always uniform . . . 17
  - 3.5.3 Hash collision considerations . . . 18
  - 3.5.4 WETH variables can be renamed . . . 19
  - 3.5.5 `Treasury` and `CORE_PROTOCOL_ROLE` considerations . . . 19
  - 3.5.6 `execute()` can't use contract's own balance . . . 20
  - 3.5.7 Signature malleability considerations . . . 21

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

From August 5th, 2024 through August 9th, 2024, Riley Holterhus conducted an audit of NodeSet's Constellation smart contracts. During this period, a manual analysis was undertaken to identify various security issues and logic flaws.

This audit was conducted on the codebase found in the `nodeset-org/constellation` GitHub repository, starting on commit 2ca3363. The following files were in scope for the audit:

- contracts/Operator/OperatorDistributor.sol
- contracts/Operator/SuperNodeAccount.sol
- contracts/Operator/YieldDistributor.sol
- contracts/Interfaces/RocketPool/IRocketNodeStaking.sol
- contracts/Interfaces/RocketPool/IRocketStorage.sol
- contracts/Interfaces/RocketPool/IMinipool.sol
- contracts/Interfaces/RocketPool/IRocketMinipoolManager.sol
- contracts/Interfaces/RocketPool/IRocketNodeDeposit.sol
- contracts/Interfaces/RocketPool/IRocketNodeManager.sol
- contracts/Interfaces/RocketPool/IRocketDAOProtocolSettingsMinipool.sol
- contracts/Interfaces/RocketPool/IRocketDAOProtocolProposal.sol
- contracts/Interfaces/RocketPool/IRocketDAOProtocolSettingsRewards.sol
- contracts/Interfaces/RocketPool/IRocketMerkleDistributorMainnet.sol
- contracts/Interfaces/RocketPool/RocketTypes.sol
- contracts/Interfaces/RocketPool/IRocketNetworkPrices.sol
- contracts/Interfaces/RocketPool/IRocketNetworkPenalties.sol
- contracts/Interfaces/RocketPool/IRocketNetworkVoting.sol
- contracts/Interfaces/IWETH.sol
- contracts/Interfaces/Oracles/IBeaconOracle.sol
- contracts/Interfaces/ISanctions.sol
- contracts/Treasury.sol
- contracts/Tokens/RPLVault.sol
- contracts/Tokens/WETHVault.sol
- contracts/Directory.sol
- contracts/UpgradeableBase.sol
- contracts/AssetRouter.sol
- contracts/Utils/ProtocolMath.sol
- contracts/Utils/Constants.sol
- contracts/Utils/Errors.sol
- contracts/Utils/RocketPoolEncoder.sol
- contracts/Oracle/PoABeaconOracle.sol
- contracts/Oracle/ZKBeaconOracle.sol

- contracts/PriceFetcher.sol
- contracts/Whitelist/Whitelist.sol

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, 2 critical-severity findings, 5 high-severity findings, 2 medium-severity findings, 3 low-severity findings and 7 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 Critical Severity Findings

3.1.1 `close()` does not clear all necessary storage

Description: The `close()` function in the `SuperNodeAccount` contract is implemented as follows:

```
function close(address _subNodeOperator, address _minipool) external onlyRecognizedMinipool(
    _minipool) {
    IMinipool minipool = IMinipool(_minipool);
    OperatorDistributor(_directory.getOperatorDistributorAddress()).onNodeMinipoolDestroy(
        _subNodeOperator);
    _stopTrackingMinipool(_minipool);

    minipool.close();
}
```

In this implementation, the node operator's validator count is decremented via `onNodeMinipoolDestroy()`, and the minipool is removed from the `minipools` and `minipoolIndex` storage variables through `_stopTrackingMinipool()`. However, this logic does not remove the minipool from the `subNodeOperatorMinipools` and `subNodeOperatorHasMinipool` storage variables.

The `subNodeOperatorMinipools` variable is used in the `stopTrackingOperatorMinipools()` function, which is the only way to prevent a scrubbed node operator from eventually reclaiming their locked ETH. As such, the protocol can be in a situation where it needs to call `stopTrackingOperatorMinipools()`, but this call will incorrectly attempt to clear storage that has already been cleared. Since `_stopTrackingMinipool()` will succeed due to mistakenly considering the previously-deleted minipool to be at index 0, other users' nodes will be incorrectly deleted from the system.

Recommendation: Update the `close()` function to ensure that all relevant storage associated with the minipool and the node operator is properly cleared.

NodeSet: Fixed in PR 280 and PR 285.

Auditor: Verified.

3.1.2 `processMinipool()` does not account for refund balance

Description: In the `OperatorDistributor`, the permissionless `processMinipool()` function allows for skimming accrued ETH from a minipool back into the Constellation smart contracts. Part of this functionality requires determining how much ETH was transferred from the minipool into Constellation and distinguishing between bond repayment amounts and rewards. This is currently implemented as follows:

```solidity
uint256 balanceAfterRefund = address(minipool).balance - minipool.getNodeRefundBalance();
if(balanceAfterRefund >= depositBalance) { // it's an exit, and any extra nodeShare is rewards
    uint256 remainingBond = minipool.calculateNodeShare(balanceAfterRefund) < depositBalance ?
        minipool.calculateNodeShare(balanceAfterRefund) : depositBalance;
    rewards = minipool.calculateNodeShare(balanceAfterRefund) > depositBalance ? minipool.
        calculateNodeShare(balanceAfterRefund) - depositBalance : 0;
    // withdrawal address calls distributeBalance(false)
    ar.onExitedMinipool(minipool);
    // stop tracking
    this.onNodeMinipoolDestroy(sna.getSubNodeOpFromMinipool(address(minipool)));
    // both bond and rewards are received
    ar.onEthRewardsAndBondReceived(rewards, remainingBond, treasuryFee, noFee, true);
} else if (balanceAfterRefund < depositBalance) { // it's still staking
    rewards = minipool.calculateNodeShare(balanceAfterRefund);
    // withdrawal address calls distributeBalance(true)
    ar.onClaimSkimmedRewards(minipool);
    // calculate only rewards
    ar.onEthRewardsReceived(rewards, treasuryFee, noFee, true);
}
```

Notice that the `calculateNodeShare()` function is being used to determine the amount of ETH rewards that will be sent from the minipool. This is not entirely accurate, because the `calculateNodeShare()` function does not consider the node's refund balance, which is the amount of ETH the node operator is owed but has not yet claimed. While the `balanceAfterRefund` variable does consider this refund balance, it is not factored back into the reward calculations by either Rocket Pool or Constellation. Consequently, this can lead to an underestimation of the amount of ETH rewards transferred from the minipool.

Since the node's refund balance increases whenever a user independently calls the minipool's `distributeBalance()` function, and because the `AssetRouter` permanently loses any excess ETH sent to it, this flaw could result in most ETH rewards becoming untracked and stuck in the `AssetRouter`.

Recommendation: Consider simplifying the calculations by tracking the difference in ETH balance before and after each call to `distributeBalance()`. When rewards are skimmed from the minipool, this difference can be used as the `rewards` value passed to `onEthRewardsReceived()`. If a bond is also returned from the minipool, the logic can subtract the bond amount from the difference.

Also, it's important to note that this approach assumes the minipool's refund balance consists entirely of accrued rewards. While this assumption holds true for Constellation, it may not be accurate in general. In Rocket Pool, functions like `reduceBondAmount()` and `prepareVacancy()` can increment the minipool's refund balance with non-reward amounts. However, since this functionality is not supported by Constellation, documenting this limitation should suffice for now.

NodeSet: Fixed in PR 282, PR 285, and PR 304.

Auditor: Verified.

3.2 High Severity Findings

3.2.1 `close()` can decrement arbitrary validator counts

Description: In the `SuperNodeAccount` contract, the `close()` function has the following documentation and implementation:

```
/**
 * @notice Closes a dissolved minipool and updates the tracking and financial records accordingly.
 * @dev This function handles the administrative closure of a minipool, ensuring that the
     associated
 *      records are updated. Only callable by an admin.
 * @param _subNodeOperator Address of the sub-node operator associated with the minipool.
 * @param _minipool Address of the minipool to close.
 */
function close(address _subNodeOperator, address _minipool) external hasConfig(_minipool) {
    IMinipool minipool = IMinipool(_minipool);
    OperatorDistributor(_directory.getOperatorDistributorAddress()).onNodeMinipoolDestroy(
        _subNodeOperator);
    _stopTrackingMinipool(_minipool);

    minipool.close();
}
```

Since this function is permissionless, there should be verification to ensure that the `_subNodeOperator` argument is indeed the correct address associated with the `_minipool` address. However, this check is currently missing, allowing an arbitrary `_subNodeOperator` to be passed, which can result in decrementing the wrong user's validator count.

Recommendation: Add the following check to the code:

```
function close(address _subNodeOperator, address _minipool) external hasConfig(_minipool) {
+   require(minipoolData[_minipool].subNodeOperator == _subNodeOperator);
    IMinipool minipool = IMinipool(_minipool);

    OperatorDistributor(_directory.getOperatorDistributorAddress()).onNodeMinipoolDestroy(_subNodeOperator);
    _stopTrackingMinipool(_minipool);

    minipool.close();
}
```

Also, since this function is permissionless, consider updating the documentation to remove the phrase "`Only callable by an admin.`"

NodeSet: Validation added in PR 291. Comment removed in PR 280.

Auditor: Verified.

3.2.2 `processMinipool()` can be called on unrelated addresses

Description: The `processMinipool()` function includes the following check to ensure that the minipool is part of the Constellation system:

```
function processMinipool(IMinipool minipool) public {
    // ...
    SuperNodeAccount sna = SuperNodeAccount(_directory.getSuperNodeAddress());
    require(sna.minipoolIndex(address(minipool)) < sna.getNumMinipools(), "Must be a minipool
        managed by Constellation");
    // ...
}
```

This check is important because the minipool is later called in the function, and its return values are trusted to influence control flow and determine the amount of ETH that needs to be accounted for in the system.

However, the check is currently flawed. If the minipool is a random address unrelated to Constellation, its `minipoolIndex` (a mapping value in the `SuperNodeAccount` contract) will default to zero, which mistakenly passes the check.

Recommendation: Implement a different method of verifying that the minipool address is indeed part of Constellation. This could be achieved by using a different storage mapping from the `SuperNodeAccount`, for example, the `minipoolData` mapping.

NodeSet: Fixed in PR 267.

Auditor: Verified.

3.2.3 Validator counts are never decremented

Description: When a node operator has a minipool removed or closed, the `removeValidator()` function within the `Whitelist` contract is eventually called. The current implementation of this function is as follows:

```
function removeValidator(address nodeOperator) public onlyProtocol {
    // ensure this is a real validator
    if(nodeMap[nodeOperator].currentValidatorCount != 0) {
        return;
    }
    nodeMap[nodeOperator].currentValidatorCount--;
}
```

This implementation incorrectly decrements the `currentValidatorCount` only if it is already zero, which is the opposite of the intended behavior. As a result, validator counts are not properly decremented, leading to downstream issues in the `createMinipool()` and `harvest()` functions.

Recommendation: Correct the equality check as follows:

```
function removeValidator(address nodeOperator) public onlyProtocol {
    // ensure this is a real validator
-   if(nodeMap[nodeOperator].currentValidatorCount != 0) {
+   if(nodeMap[nodeOperator].currentValidatorCount == 0) {
        return;
    }
    nodeMap[nodeOperator].currentValidatorCount--;
}
```

NodeSet: Fixed in PR 266.

Auditor: Verified.

3.2.4 Discrete jumps in vault exchange rates can be sandwiched

Description: In the `WETHVault` and `RPLVault` contracts, any large discrete jumps in the vaults' exchange rates can create an opportunity for sandwich attacks. This style of attack occurs when a user profits by sandwiching a sudden change in the exchange rate with a deposit transaction and a redemption transaction, which essentially steals profit from honest users. Currently, neither vault has implemented a mechanism to prevent this attack, and moreover, there are at least two scenarios where discrete jumps can occur:

1. When the oracle submits an update transaction, all accrued beacon chain ETH rewards since the last update are added to the `WETHVault` exchange rate. These updates are expected to occur daily.
2. When `merkleClaim()` is executed, a discrete jump will occur in both vaults. Since merkle claims do not contribute to the `oracleError` logic and are therefore excluded from oracle updates, they only affect the system's accounting once claimed. These merkle claims are expected to happen in large monthly batches.

Note that scenario (1) is also present in Rocket Pool and was identified in a 2021 audit. Rocket Pool initially mitigated this issue by introducing a required delay on deposits and redemptions, but later switched to a deposit fee to improve user experience.

While the deposit fee approach generally mitigates the issue, the fee must scale with the potential arbitrage opportunity. Scenario (1) involves smaller arbitrage opportunities that could be addressed with a fee. However, scenario (2) presents a much larger risk due to the infrequent nature of the merkle claims.

Recommendation: After discussion with the team, there were two main ideas for how to mitigate this issue.

The first idea is to include the merkle claim values in the daily oracle updates. In other words, although the actual merkle claim occurs every 28 days, each daily oracle update would estimate how much the end-of-period claim value has increased since the last update. However, this approach has a potential drawback: the daily updates would only be estimates, which means unexpected discrete jumps could still occur. For an example of why the daily updates can't be perfectly accurate, notice that the Rocket Pool rewards calculation does not prorate rewards for node operators below 10% collateralization at the end of the reward period, even if they were above 10% earlier in the 28-day cycle.

The second idea is to “stream” the reward payments from one interval over the duration of the next reward interval. This approach would eliminate discrete jumps, but would also introduce potential fairness concerns. This is because new depositors would have access to rewards for a period during which they were not deposited, and withdrawers would forfeit their rewards for the last period.

Ultimately, the second idea has been implemented.

NodeSet: As described above, implemented backwards streaming of rewards in PR 286.

Auditor: Verified.

3.2.5 Frontrunning oracle updates will double-count ETH

Description: To help determine the exchange rate for xrETH <> WETH conversions, the Constellation system uses a beacon chain oracle contract that records the total yield currently held in the system’s minipools and on the beacon chain. Since this yield can be skimmed from minipools later and would otherwise be double-counted, the contracts utilize an `oracleError` storage variable to prevent this. Each oracle update resets this `oracleError`, since the oracle should exclude all ETH that is already within the Constellation contracts:

```
function _setTotalYieldAccrued(bytes calldata _sig, int256 _newTotalYieldAccrued, uint256
    _sigTimeStamp) internal {
    // ...
    OperatorDistributor(_directory.getOperatorDistributorAddress()).resetOracleError();
}
```

However, this logic creates a vulnerability. If ETH is skimmed from a minipool after the oracle signs an update but before the update is included on-chain, the ETH will be double-counted. Since it would not be difficult to intentionally front-run oracle updates with a skim of several minipools (using the `processMinipool()` function), a malicious actor could easily trigger double-counting of ETH within the system, artificially inflating the xrETH <> WETH exchange rate.

Recommendation: One potential fix is to have the oracle “commit” to the `oracleError`, and if there’s any difference from the actual `oracleError` when the transaction is executed on-chain, then the difference can be subtracted from the update. However, this may cause problems if the difference is larger than the update itself.

After discussion with the team, this was the approach taken for the time being. Even with this potential concern, the new implementation has fewer problems than the original implementation.

NodeSet: As described above, addressed in PR 282 and PR 283.

Auditor: Verified.

3.3 Medium Severity Findings

3.3.1 `getRequiredCollateral()` does not consider contract’s existing balance

Description: The `getRequiredCollateral()` function is implemented as follows:

```
function getRequiredCollateral() public view returns (uint256) {
    return getRequiredCollateralAfterDeposit(0);
}

function getRequiredCollateralAfterDeposit(uint256 deposit) public view returns (uint256) {
    uint256 fullBalance = totalAssets() + deposit;
    uint256 requiredBalance = liquidityReservePercent.mulDiv(fullBalance, 1e18, Math.Rounding.Up);
    return requiredBalance > balanceWeth ? requiredBalance : 0;
}
```

Notice that the return value from `getRequiredCollateralAfterDeposit()` is either zero if the contract's balance is sufficient to cover the required collateral, or the entire required collateral amount if it is not. This creates an asymmetry where the return value sometimes accounts for the existing `balanceWeth` and sometimes does not.

This asymmetry can lead to overestimations of the amount of ETH that needs to be sent to the vault. For example, notice how the `sendEthToDistributors()` function sends the entire `getRequiredCollateral()` value into the vault:

```
uint256 requiredCapital = vweth.getRequiredCollateral();
if (balanceEthAndWeth >= requiredCapital) {
    // ...
    SafeERC20.safeTransfer(weth, address(vweth), requiredCapital);
}
```

Recommendation: To prevent overestimations in the amount of collateral that needs to be sent, consider modifying the `getRequiredCollateralAfterDeposit()` function to account for the contract's existing balance:

```
function getRequiredCollateralAfterDeposit(uint256 deposit) public view returns (uint256) {
    uint256 fullBalance = totalAssets() + deposit;
    uint256 requiredBalance = liquidityReservePercent.mulDiv(fullBalance, 1e18, Math.Rounding.Up);
-   return requiredBalance > balanceWeth ? requiredBalance : 0;
+   return requiredBalance > balanceWeth ? requiredBalance - balanceWeth : 0;
}
```

NodeSet: Fixed in PR 282.

Auditor: Verified.

3.3.2 Constellation minipools are susceptible to donation attack

Description: In the `OperatorDistributor` contract, the `processMinipool()` function is used to skim ETH from a Constellation minipool back into the Constellation smart contracts. As part of this process, the code decides whether to call `distributeBalance(true)` or `distributeBalance(false)` on the minipool depending on whether the minipool's balance is sufficient for the minipool to be considered fully exited. The implementation is roughly as follows:

```
uint256 balanceAfterRefund = address(minipool).balance - minipool.getNodeRefundBalance();
if(balanceAfterRefund >= depositBalance) {
    // ...
    // withdrawal address calls distributeBalance(false)
    ar.onExitedMinipool(minipool);
    // ...
} else if (balanceAfterRefund < depositBalance) {
    // ...
    // withdrawal address calls distributeBalance(true)
    ar.onClaimSkimmedRewards(minipool);
    // ...
}
```

Note that the reason the `distributeBalance()` function takes a boolean argument is to allow the caller to indicate whether they expect the minipool to be treated as fully exited or not. This is important because an attacker can potentially manually transfer ETH into a minipool right before the node operator calls `distributeBalance()`, which would trick the system into thinking the validator has fully exited even if they haven't. In this scenario, the `distributeBalance()` call would treat the minipool as though it exited with a significant loss of ETH, leading to a slashing of the node operator, even though the validator is still active.

While this attack is unlikely due to the cost of "donating" ETH into the minipool, it could be profitable in theory. This is because the donated ETH and the RPL slashing ultimately benefit rETH holders, so an attacker who owns a large percentage of rETH could potentially profit from this manipulation.

Since the `processMinipool()` function decides whether to call `distributeBalance(true)` or `distributeBalance(false)` based solely on the minipool's balance, the original problem (which the boolean argument is meant to address) still persists. In other words, an attacker could manually transfer 8 ETH into a Constellation minipool and then call `processMinipool()` to trick Constellation into slashing itself on a validator that has not actually exited.

Recommendation: Since it will be rare for a Constellation minipool to exit with less than 32 ETH, consider not calling `distributeBalance(false)` if the minipool's balance is below 32 ETH. This approach would prevent the `processMinipool()` function from slashing itself in an attack scenario, giving Constellation admins time to investigate and, if necessary, exit the manipulated minipool to claim the donated funds.

NodeSet: Fixed in PR 285.

Auditor: Verified. The `processMinipool()` function will no longer automatically call `distributeBalance(false)` if the minipool's balance is low enough to be slashed, and instead a `SuspectedPenalizedMinipoolExit()` event will be emitted. Admins can later manually intervene by calling `distributeExitedMinipool()`.

3.4 Low Severity Findings

3.4.1 Signatures are invalidated even if `adminServerCheck == false`

Description: When the `adminServerCheck` storage variable is set to `true`, the `SuperNodeAccount` contract requires a signature to verify calls to `createMinipool()`. This is currently implemented as follows:

```
_validateSigUsed(_config.sig);
// ...
if (adminServerCheck) {
    // ...
    address recoveredAddress = ECDSA.recover(/* ... */), _config.sig);
    require(
        _directory.hasRole(Constants.ADMIN_SERVER_ROLE, recoveredAddress),
        'signer must have permission from admin server role'
    );
}
```

Notice that even if the `adminServerCheck` variable is set to `false`, the `_validateSigUsed()` function is still called. This function checks that the signature hasn't been used before and invalidates it in the process. This is unnecessary if `adminServerCheck == false`, since the signature is never used.

Recommendation: Move the `_validateSigUsed()` call to only happen if `adminServerCheck == true`:

```
- _validateSigUsed(_config.sig);
  // ...
  if (adminServerCheck) {
+   _validateSigUsed(_config.sig);
    // ...
}
```

NodeSet: Fixed in PR 268.

Auditor: Verified.

3.4.2 `sendEthToDistributors()` can unnecessarily change the `_gateOpen` variable

Description: In the `AssetRouter` contract, the `_gateOpen` variable controls whether the contract should accept ETH transfers:

```
receive() external payable {
    require(_gateOpen);
}

fallback() external payable {
    require(_gateOpen);
}
```

Within the same contract, the `sendEthToDistributors()` function opens and closes this gate before and after calling `weth.withdraw()`:

```
_gateOpen = true;
weth.withdraw(surplus);
_gateOpen = false;
```

However, this logic does not account for the possibility that `sendEthToDistributors()` might be called when `_gateOpen` is already set to `true`. For example, the `processMinipool()` function is implemented roughly as follows:

```
ar.openGate();
// ...
ar.sendEthToDistributors();
ar.closeGate();
```

In this case, it would make more sense for the gate to not be closed by the `sendEthToDistributors()` function, since the caller would expect to be managing it themselves.

Recommendation: Consider changing the `sendEthToDistributors()` function to not affect the `_gateOpen` variable if it is already `true`. For example, consider the following implementation:

```
bool alreadyOpen = _gateOpen;
if (!alreadyOpen) _gateOpen = true;
weth.withdraw(surplus);
if (!alreadyOpen) _gateOpen = false;
```

NodeSet: Indirectly fixed in PR 282 by removing the `AssetRouter` and `_gateOpen` storage variable.

Auditor: Verified.

3.4.3 Frontrunning `withdrawal_credentials` considerations

Description: A common issue in Ethereum liquid staking protocols is the "frontrunning `withdrawal_credentials`" attack. Essentially, if the beacon chain observes conflicting `withdrawal_credentials` across multiple deposits into a validator, it will default to the first `withdrawal_credentials` it received. To prevent user ETH from being deposited into validators with unexpected `withdrawal_credentials`, most liquid staking protocols must implement mechanisms to safeguard against this attack.

In Rocket Pool, this risk is mitigated by a mechanism known as the "scrub period." During this time, Rocket Pool can verify off-chain that the attack did not occur and take corrective action if it did. Since Constellation is built on top of Rocket Pool, any Constellation node operator attempting this attack would be caught by the scrub period. So, if this attack were to happen, the consequences would be as follows:

- The Constellation node operator would forfeit the ETH they provided as the `lockThreshold` in `createMinipool()`, which would be reclaimed by the Constellation smart contracts. This currently amounts to 1 ETH.

- The Constellation node operator would regain the `preLaunchValue` amount of ETH that Rocket Pool deposited into their malicious validator, also amounting to 1 ETH.
- The Rocket Pool contracts would vote to scrub the malicious minipool, resulting in 2.4 ETH worth of RPL being slashed from Constellation's balance.

In this scenario, the malicious node operator would recoup all of their ETH (excluding gas costs), while Constellation would lose 2.4 ETH worth of RPL. This suggests that a malicious node operator could spend a bit of gas to force a financial loss on Constellation.

Recommendation: Consider whether this risk is worth addressing. Given that node operators in NodeSet are whitelisted and trusted, this attack is unlikely to occur and might be safely ignored. However, if the risk seems worth addressing, it may be preferred to increase the `lockThreshold` required from the node operator. There are other potential mechanisms that can be explored to prevent the frontrunning attack, such as requiring an admin signature that commits to the expected deposit contract root when `createMinipool()` is called.

NodeSet: Acknowledged. In general, adversarial NO issues are not necessary to directly address, as NOs are in a semi-trusted role and maximum damage is limited.

Auditor: Acknowledged.

3.5 Informational Findings

3.5.1 `subNodeOperatorHasMinipool` mapping can be removed

Description: The `subNodeOperatorHasMinipool` mapping exists within the `SuperNodeAccount` contract and is used as follows:

```
modifier onlySubNodeOperator(address _minipool) {
    require(
        subNodeOperatorHasMinipool[keccak256(abi.encodePacked(msg.sender, _minipool))],
        'Can only be called by SubNodeOperator!'
    );
    _;
}

modifier onlyAdminOrAllowedSNO(address _minipool) {
    if(allowSubOpDelegateChanges) {
        require(
            _directory.hasRole(Constants.ADMIN_ROLE, msg.sender) ||
                subNodeOperatorHasMinipool[keccak256(abi.encodePacked(msg.sender, _minipool))],
            'Can only be called by admin or sub node operator'
        );
    } else {
        require(_directory.hasRole(Constants.ADMIN_ROLE, msg.sender), 'Minipool delegate changes
            only allowed by admin');
    }
    _;
}
```

Since information about minipools and their related node operators is already stored in the `minipoolData` mapping, the `subNodeOperatorHasMinipool` mapping is redundant and can be removed.

Recommendation: Consider simplifying the code by removing the `subNodeOperatorHasMinipool` mapping and replacing its usage with the `minipoolData` mapping. For example:

```
modifier onlySubNodeOperator(address _minipool) {
    require(
-       subNodeOperatorHasMinipool[keccak256(abi.encodePacked(msg.sender, _minipool))],
+       minipoolData[_minipool].subNodeOperator == msg.sender
        'Can only be called by SubNodeOperator!'
    );
    _;
}

modifier onlyAdminOrAllowedSNO(address _minipool) {
    if(allowSubOpDelegateChanges) {
        require(
            _directory.hasRole(Constants.ADMIN_ROLE, msg.sender) ||
-               subNodeOperatorHasMinipool[keccak256(abi.encodePacked(msg.sender, _minipool))],
+               minipoolData[_minipool].subNodeOperator == msg.sender
            'Can only be called by admin or sub node operator'
        );
    } else {
        require(_directory.hasRole(Constants.ADMIN_ROLE, msg.sender), 'Minipool delegate changes
    only allowed by admin');
    }
    _;
}
```

NodeSet: Fixed in PR 280.

Auditor: Verified.

3.5.2 `getNextMinipool()` iteration isn't always uniform

Description: Several contracts within Constellation call the `processNextMinipool()` function, which in turn calls `getNextMinipool()` to determine which minipool is next in the queue to be processed. Ideally, the `getNextMinipool()` function should return the minipool that has been waiting the longest to be processed. However, this isn't always the case.

For example, if a minipool is removed, the `_stopTrackingMinipool()` function will swap the last minipool index with the index of the removed minipool. As a result, the minipool that was swapped might be temporarily skipped or processed twice in a short period.

Another example is when a new minipool is created. This increases the `getNumMinipools()` count, potentially placing the new minipool next in the queue, even though it is the most recent addition.

Recommendation: Since this does not appear to be a major problem and is more of an inefficiency, consider simply documenting this behavior in the `OperatorDistributor` contract.

NodeSet: Comment added in PR 302.

Auditor: Verified.

3.5.3 Hash collision considerations

Description: When an admin signs a hash for use in the Constellation smart contracts, the hash typically includes `address(this)` and `block.chainid`. For example:

```
function createMinipool(CreateMinipoolConfig calldata _config) public payable {
    // ...
    address recoveredAddress = ECDSA.recover(
        ECDSA.toEthSignedMessageHash(
            keccak256(
                abi.encodePacked(
                    _config.expectedMinipoolAddress,
                    salt,
                    _config.sigGenesisTime,
                    address(this),
                    block.chainid
                )
            )
        ),
        _config.sig
    );
    // ...
}

function merkleClaim(
    uint256[] calldata _rewardIndex,
    uint256[] calldata _amountRPL,
    uint256[] calldata _amountETH,
    bytes32[][] calldata _merkleProof,
    MerkleRewardsConfig calldata _config
) public {
    // ...
    bytes32 messageHash = keccak256(
        abi.encodePacked(
            _config.avgEthTreasuryFee,
            _config.avgEthOperatorFee,
            _config.avgRplTreasuryFee,
            _config.sigGenesisTime,
            address(this),
            merkleClaimNonce,
            block.chainid
        )
    );
    // ...
}
```

While including `address(this)` and `block.chainid` helps differentiate hashes from different contracts, it's also important to ensure that hashes within the same contract cannot collide. In the example above, which is currently the only instance where two hash calculations exist within a single contract, there is no collision risk because the hash preimage sizes differ. However, in the future, it should be noted that even with `address(this)` and `block.chainid`, collisions can occur within a contract.

Recommendation: To make the hash calculations more robust, consider adopting EIP-712 for signing data. This would allow each function to use its own typehash, ensuring that hashes are distinct even within the same contract.

NodeSet: Indirectly addressed by removing signature for merkle claims in PR 286.

Auditor: Verified. The `ECDSA` library is no longer used twice in one contract.

3.5.4 WETH variables can be renamed

Description: In the `YieldDistributor` contract, most variable and function names reference WETH. For example:

```
function wethReceived(uint256 weth) external onlyProtocol {
    _wethReceived(weth, false);
}

function wethReceivedVoidClaim(uint256 weth) external onlyProtocol {
    _wethReceived(weth, true);
}

/**
 * @dev Handles the internal logic when WETH (Wrapped Ether) is received by the contract.
 * It updates the total yield accrued and checks if the current interval should be finalized.
 *
 * @param weth The amount of WETH received by the contract.
 */
function _wethReceived(uint256 weth, bool voidClaim) internal {
    // ...
}
```

However, this contract currently only manages ETH and does not manage WETH. The current naming may lead to confusion in the future.

Recommendation: Consider renaming the functions and variables to reference ETH instead of WETH.

NodeSet: Indirectly fixed in PR 278 with a refactor of the `YieldDistributor` contract.

Auditor: Verified.

3.5.5 `Treasury` and `CORE_PROTOCOL_ROLE` considerations

Description: Currently, the `Treasury` contract is not one of the contracts intended to be granted the `CORE_PROTOCOL_ROLE`:

```
_grantRole(Constants.CORE_PROTOCOL_ROLE, newProtocol.whitelist);
_grantRole(Constants.CORE_PROTOCOL_ROLE, newProtocol.wethVault);
_grantRole(Constants.CORE_PROTOCOL_ROLE, newProtocol.rplVault);
_grantRole(Constants.CORE_PROTOCOL_ROLE, newProtocol.operatorDistributor);
_grantRole(Constants.CORE_PROTOCOL_ROLE, newProtocol.oracle);
_grantRole(Constants.CORE_PROTOCOL_ROLE, newProtocol.priceFetcher);
_grantRole(Constants.CORE_PROTOCOL_ROLE, newProtocol.superNode);
```

It is important that this remains the case in the future, since anyone with the `TREASURY_ROLE` can make arbitrary calls from the `Treasury` contract. If the `Treasury` were granted the `CORE_PROTOCOL_ROLE`, this would mean the `Treasury` could call `onlyProtocol` restricted functions like `onNodeMinipoolDestroy()`, which would give the `TREASURY_ROLE` much more power than they're likely intended to have.

Recommendation: Ensure that the `Treasury` contract is never granted the `CORE_PROTOCOL_ROLE`, and consider documenting this concern in the comments of the `Treasury` and `Directory` contracts.

NodeSet: Addressed in PR 286.

Auditor: Verified. Relevant comments have been added to `Constants.sol` and `Directory.sol`, and the separation has been made more obvious with a file reorganization, with `Treasury.sol` now placed in a directory called `External/`.

3.5.6 `execute()` can't use contract's own balance

Description: The `execute()` and `executeAll()` functions within the `Treasury` contract are implemented as follows:

```
function execute(address payable _target, bytes calldata _functionData) external payable
    onlyTreasurer nonReentrant {
    _executeInternal(_target, _functionData, msg.value);
}

function executeAll(
    address payable[] calldata _targets,
    bytes[] calldata _functionData,
    uint256[] calldata _values
) external payable onlyTreasurer nonReentrant {
    require(_targets.length == _functionData.length, Constants.BAD_TREASURY_BATCH_CALL);
    require(_values.length == _functionData.length, Constants.BAD_TREASURY_BATCH_CALL);
    for (uint256 i = 0; i < _targets.length; i++) {
        _executeInternal(_targets[i], _functionData[i], _values[i]);
    }
}
```

In the `execute()` function, any ETH to be used during execution must be provided as `msg.value`, whereas `executeAll()` allows specifying `_values` to draw from the contract's existing balance.

Recommendation: To make the code more consistent and provide more flexibility, consider modifying the execute() function to accept a _value argument as well. Alternatively, since executeAll() can always be used in place of execute() by using a singleton array, consider removing the execute() function altogether.

NodeSet: Addressed in PR 286.

Auditor: Verified. The execute() function has been removed in favor of executeAll().

3.5.7 Signature malleability considerations

Description: In the Constellation contracts, there are several locations where signatures from protocol admins are used to verify that an action is authorized. For example, consider the following function from the Whitelist contract:

```
function _addOperator(
    address _operator,
    uint256 _sigGenesisTime,
    bytes memory _sig
) internal returns (Operator memory) {
    require(!sigsUsed[_sig], 'sig already used');
    require(block.timestamp - _sigGenesisTime < whitelistSigExpiry, 'wl sig expired');
    sigsUsed[_sig] = true;
    bytes32 messageHash = keccak256(abi.encodePacked(_operator, _sigGenesisTime, address(this),
        block.chainid));

    bytes32 ethSignedMessageHash = ECDSA.toEthSignedMessageHash(messageHash);

    address recoveredAddress = ECDSA.recover(ethSignedMessageHash, _sig);
    require(_directory.hasRole(Constants.ADMIN_SERVER_ROLE, recoveredAddress), 'signer must be
        admin server role');
    // ...
}
```

Notice that this logic prevents signature replay by invalidating the signature using the sigsUsed mapping. With this implementation, it's important to note that signatures in the ECDSA system can be vulnerable to signature malleability. This concern occurs when an attacker exploits the symmetry of the elliptic curve math to create an equivalent signature (i.e. a signature that correctly signs the same data from the same address) without ever knowing the signer's private key.

Since the implementation above only invalidates a single signature, it's theoretically possible for someone to bypass the replay protection by using an equivalent signature produced using malleability. This would allow an attacker to repeat an action that wasn't intended to be repeated. Fortunately, this is not a concern in the current codebase, because the OpenZeppelin ECDSA library includes a specific check to only accept one of the two possible signatures:

```
// EIP-2 still allows signature malleability for ecrecover(). Remove this possibility and make the
    signature
// unique. Appendix F in the Ethereum Yellow paper (https://ethereum.github.io/yellowpaper/paper.
    pdf), defines
```

```
// the valid range for s in (301): 0 < s < secp256k1n / 2 + 1, and for v in (302): v in {27, 28}.
    Most
// signatures from current libraries generate a unique signature with an s-value in the lower half
    order.
//
// If your library generates malleable signatures, such as s-values in the upper range, calculate a
     new s-value
// with 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141 - s1 and flip v from 27
    to 28 or
// vice versa. If your library also generates signatures with 0/1 for v instead 27/28, add 27 to v
    to accept
// these malleable signatures as well.
if (uint256(s) > 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0) {
    return (address(0), RecoverError.InvalidSignatureS, s);
}
```

Recommendation: While signature malleability does not appear to be a concern due to the safeguards in the OpenZeppelin dependency, consider adding extra measures to prevent malleability concerns entirely. For example, instead of invalidating signatures, consider introducing an incrementing nonce in every hash calculation. This would automatically invalidate all signatures once the nonce increases to a new value.

NodeSet: Addressed in PR 294 and PR 327.

Auditor: Verified.