Rocket Pool

Pre-Merge Upgrades

Version: 2.0

June, 2022

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Rocket Pool smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the upgrades and intended purposes of the Rocket Pool smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Rocket Pool smart contracts.

Overview

Rocket Pool is a decentralised staking network focused on the Ethereum consensus beacon chain. The long-standing Ethereum chain (the execution layer) and the beacon chain are in the process of undergoing a hard-fork (colloquially known as "the merge") to unify the two chains. This significant upgrade requires modifications to the Rocket Pool protocol to handle the new features and complications that the merge brings to Ethereum consensus stakers. This review is focused on the upgrades proposed by Rocket Pool to handle the new merge features and correct some small issues in the previous design.

The modifications to the Rocket Pool protocol fundamentally consist of the following:

- **Fee Distribution System -** After the merge, validators will start receiving fees from transactions when they propose blocks. Half of these should rightfully go to `rETH` holders and as such a fee distribution system has been introduced.
- **Improved Reward System -** Some small issues with the previous RPL reward system design has warranted an upgrade of the system and is included in this review.
- **Deposit Fee -** A 24 hour delay that was imposed to `rETH` tokens after being minted preventing them from being burned or transferred has been lifted in favour of adding a small deposit fee.

Security Assessment Summary

This review was conducted on the files hosted on the RocketPool repository and the review focused on the changes introduced between commits 24e7a6e and 99e984.

A subsequent round of review targeted commit f7657e6 and focused solely on verifying whether previously identified issues had been addressed.

Note: the OpenZeppelin libraries and dependencies were excluded from the scope of this assessment.

The manual code review section of the report, focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts. Specifically, their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focused on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [1, 2].

To support this review, the testing team used the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 12 issues during this assessment. Categorized by their severity:

- High: 1 issue.
- Medium: 2 issues.
- Low: 4 issues.
- Informational: 5 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Rocket Pool smart contracts. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as "informational".

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
RPM-01	Insufficient Input Validation In `executeRewardSnapshot()`	High	Resolved
RPM-02	Improper Access Control For `createProxy()`	Medium	Resolved
RPM-03	Inaccurate Priority Fee Distribution When Changing Minipool Count	Medium	Resolved
RPM-04	Improper Migration Access Control Allows Interference	Low	Resolved
RPM-05	Differing Thresholds for Node Penalties	Low	Resolved
RPM-06	Node Operator Can Block Priority Fee Distribution	Low	Closed
RPM-07	Unchecked Return Value For Low Level `create2`	Low	Resolved
RPM-08	Inconsistent constructor can lead to unusable contract	Informational	Resolved
RPM-09	Protocol DAO Treasury Centralisation Risk	Informational	Closed
RPM-10	Incorrect `getFeeDistributorInitialised()` In Some Cases	Informational	Resolved
RPM-11	Revert Error Message Not Propagated	Informational	Resolved
RPM-12	Miscellaneous general comments	Informational	Resolved

RPM-01	Insufficient Input Validation In `executeRewardSnapshot()`		
Asset	`RocketRewardsPool.sol`		
Status	**Resolved:** In commit ce56d42		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

A missing input validation step in `executeRewardSnapshot()` can allow execution of an old submission, resulting in corruption of the state and potentially misplaced funds.

`executeRewardSnapshot()` is intended for use in unusual circumstances where changes to the number of ODAO members or the threshold required for consensus[1] result in a previous set of submissions reaching the required threshold.

The current implementation of `executeRewardSnapshot()` does not check that the `RewardSubmission` argument passed to it is associated with the current round of submissions i.e. it does not validate that `_submission.rewardIndex == getRewardIndex()`.

This can allow previously executed submissions to be trivially replayed. Should the currently available funds be sufficient, the old submission is executed successfully, distributing funds incorrectly and incrementing the round number. This makes it impossible for execution of valid submissions for that round number and may be repeated to continually disrupt valid submissions (until eventually disabled via a code upgrade).

In the event of changes to the number of ODAO members or the consensus threshold, it can be possible for a colluding ODAO minority to make malicious submissions that can later be executed once the effective threshold value shrinks. Even if the malicious ODAO members were to be kicked, their old submissions are still present and may be possible to execute.

Should the meaning of the network indices change (used to refer to various network relayers), old submissions may also misdirect funds to incorrect network relayers. For example, if the index 1 previously referred to the Arbitrum network and was later changed to direct to a relayer for Loopring, the values involved may be incorrect and difficult for recipients to claim.

There do not appear to be any unit tests that exercise `executeRewardSnapshot()`, let alone how it handles malicious input.

Recommendations

Modify `executeRewardSnapshot()` to contain a check similar to line [120], like
`require(_submission.rewardIndex == getRewardIndex());`.

Update unit tests to cover `executeRewardSnapshot()` and, in particular, exercise scenarios involving attempts to execute a previously executed submission.

Resolution

The recommended require statement was added.

[1]The `RocketDAOProtocolSettingsNetwork.getNodeConsensusThreshold()` setting.

RPM-02	Improper Access Control For `createProxy()`		
Asset	`RocketNodeDistributorFactory.sol`		
Status	**Resolved:** In commit 445efcd		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

No access control restrictions are implemented for `createProxy()`, allowing any account to directly create a `RocketNodeDistributor` instance and bypass the intended initialisation. A malicious entity can exploit this to prevent existing node operators from receiving an appropriate share of their priority fee rewards.

`createProxy()` is intended for use by node operators to deploy a `RocketNodeDistributor` contract at a predictable address (one instance for each node operator). This instance is used (post merge) as a recipient for transaction *priority fees* (a.k.a. *tips*) that are awarded to the node operator when including transactions in blocks they produce.[2]

These tips are intended to be split between the node operator and the rETH holders (protocol users), with the node operator receiving a share corresponding to the staked ETH collateral they own (usually half) plus a fee deducted from the users' portion.[3]

When executed as designed, existing node operators execute `RocketNodeManager.initialiseFeeDistributor()` to calculate their current *node average fee* (the proportion to deduct from the rETH holders' share) and deploy their distributor contract.

Should someone have already deployed the distributor contract by executing `createProxy()` directly, a call to `RocketNodeManager.initialiseFeeDistributor()` will revert at line [166] due to `getFeeDistributorInitialised()` returning `true` (See RPM-10). It becomes impossible to correctly initialise `"node.average.fee.numerator"` without a code update, but it is treated as if it were correctly initialised to 0. Once incorrectly distributed, these tips become difficult to recover and, because anyone can execute `RocketNodeDistributor.distribute()`, a compliant node operator cannot prevent their rewards from being distributed.

Although easily exploited, an attacker would need to pay gas fees to deploy the `RocketNodeDistributor` instance and it would be difficult for them to directly profit, as undeserved tips are diluted across all rETH holders. Any profit would be best achieved through indirect means — like causing a scene and shorting RPL. As such, the likelihood rating is deemed *medium*.

Recommendations

Implement appropriate access controls for `createProxy()`, to only allow the protocol's current `rocketNodeManager` to execute.

Resolution

A check was added to `createProxy()`, to safely restrict execution to only the protocol's current `rocketNodeManager`.

[2]This is equivalent to the current EIP-1559 mechanism but the priority fees are delivered to the block producing validator instead of the miner.
[3]Where this fee is calculated as the mean `nodeFee` value across the minipools controlled by the operator.

RPM-03	Inaccurate Priority Fee Distribution When Changing Minipool Count		
Asset	`RocketNodeManager.sol` & `RocketMinipoolManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

When changing the number of active minipools, any existing priority fee rewards are for that node are distributed. When combined with changes to the node's average node fee, the order in which this occurs can cause the funds to be incorrectly distributed between the node operator and the rETH holders.

Priority fees are distributed based on the mean fee across the node operator's active minipools. Currently, a distribution is triggered at `RocketNodeManager.sol` lines [197,204], as part of updating the number of active minipools but *after* the average node fee has been updated. While this ensures the node has a properly initialised `RocketNodeDistributor` before being allowed to stake with more minipools, any funds already in the minipool should really be distributed according to the *previous* average fee value.

This has the largest impact in the situation where a node operator has a lot of value in their `RocketNodeDistributor` and their *last* minipool is being set to a *withdrawable* status. Here `decrementNodeStakingMinipoolCount()` is executed, reducing the number of active minipools to 0 (`getNodeStakingMinipoolCount(nodeAddress) == 0`). In that situation, the average node fee is reduced to 0 so the distribution would send the operator *none* of their deserved share of the user rewards.

Recommendations

In `RocketMinipoolManager`, execute the equivalent of `RocketNodeManager._distribute()` at lines [207 & 226], *before* any change to the node's minipool count and the average fee.

Consider replacing `RocketNodeManager._distribute()` with a function requiring that the fee distributor has been initialised and there is no pending priority fee balance to distribute
(i.e. `getFeeDistributorInitialised(_nodeAddress) && distributorAddress.balance == 0`).

Resolution

This has been successfully resolved in commit e0fa60e.

An equivalent of the previous `RocketNodeManager._distribute()` implementation is executed at the start of `incrementNodeStakingMinipoolCount()` and `decrementNodeStakingMinipoolCount()` prior to any other state changes.

The `RocketNodeManager` functions `increaseAverageNodeFeeNumerator()` and `decreaseAverageNodeFeeNumerator()` have been removed without issue, as they were used only by the `RocketMinipoolManager`.

RPM-04	Improper Migration Access Control Allows Interference		
Asset	`RocketUpgradeOneDotOne.sol`		
Status	**Resolved:** In commit 2240d20		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The `set()` function has insufficient access control, allowing others to interfere with the initialisation of the `RocketUpgradeOneDotOne` contract.

Part of the initialisation for `RocketUpgradeOneDotOne` is performed in a function separate to the constructor, named `set()`. This is used to spread the gas costs across multiple blocks.

It is possible for a malicious actor to front-run a transaction executing `set()` before the migration account does, stopping it from being correctly initialised. This posses little danger to the protocol, as this interference would be quite noticeable and any upgrade proposal involving a corrupted `RocketUpgradeOneDotOne` contract should be rejected by the ODAO. As such, the impact is limited to the reasonably large gas costs involved with deploying and initialising another `RocketUpgradeOneDotOne` instance.

Any attempts at redeployment using unchanged code could be repeatedly interfered with, but the appropriate fix is simple so such a scenario is unlikely.

Recommendations

Consider limiting access to `set()`, so only the guardian or a contract owner may successfully execute it.

Resolution

Appropriate access restrictions were added, to only allow execution by the account that deployed the contract (`set()` had been split into two functions to further spread gas costs across multiple blocks).

RPM-05	Differing Thresholds for Node Penalties		
Asset	`RocketNetworkPenalties.sol`		
Status	**Resolved:** In commit 508bb32		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

There are two differing thresholds used in `submitPenalty()` and `executeUpdatePenalty()`. The former uses `getNodePenaltyThreshold()` on line [49] and the latter uses `getNodeConsensusThreshold()` on line [68].

Depending on the configuration parameters of these thresholds (specifically the case that the consensus threshold is smaller than the node penalty threshold) any user can execute node penalties below the desired node penalty threshold limit.

Recommendations

Make sure these thresholds match each other (i.e. use the same threshold for both functions).

Implement unit tests to explore scenarios where these threshold values are different.

Resolution

The penalty threshold on line [68] has been modified to `getNodePenaltyThreshold()`, making the two functions consistent.

RPM-06	Node Operator Can Block Priority Fee Distribution		
Asset	`RocketNodeDistributorDelegate.sol`		
Status	**Closed:** See the Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

A malicious node operator can choose to block distribution of any priority fees, effectively locking them in their `RocketNodeDistributor` contract.

A node operator can set their withdrawal address to a contract address that reverts upon receiving ETH from the distributor contract, causing the call at line [**38**] in `distribute()` to fail and revert at line [**39**].

As the node operator is usually the one to perform the distribution and they would receive a majority of the fees, they are disincentivised to block the distribution.

Should such a scenario occur, the `RocketNodeDistributor` can be upgraded after-the-fact to allow the release of funds.

Recommendations

Consider recommending ODAO members penalise obvious offenders.

If this becomes a problem, consider upgrading `RocketNodeDistributorDelegate` to allow `distribute()` to succeed should the ETH transfer to the operator fail. A safer option would involve tracking node operator and rETH holder balances separately, in order to protect operators with a failing withdrawal from having their funds incorrectly attributed to rETH holders.[4]

Resolution

This is acknowledged by the development team. The Node operators are not incentivised to block fees, if this becomes an issue the development team will address it.

[4]A naive solution, in which the `require(success);` at line [**39**] were simply removed, could cause loss of node operator funds. Consider a scenario in which 1 ether is to be distributed evenly ($50\% : 50\%$) between the operator and rETH holders but the operator's transfer at line [**38**] fails (e.g. due to insufficient gas).

On first execution of `distribute()`, the rETH users will receive $\frac{1}{2}$ ether and the operator transfer will fail.

A subsequent execution of `distribute()` would see an attempt to split the remaining $\frac{1}{2}$ ether between the users and operator, even though the operator should receive all the remaining value.

RPM-07	Unchecked Return Value For Low Level `create2`		
Asset	`RocketNodeDistributorFactory.sol` & `RocketMinipoolFactory.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Some Rocket Pool contracts use the low-level `create2` opcode to deploy contracts. However, the contracts fail to adequately check whether this deployment was successful.

The EVM `create2` opcode returns 0 if the contract deployment fails.[5] This can be caused by insufficient gas, the contract initialisation code reverting, or a contract already existing at that address.

In `RocketMinipoolFactory`, `extcodesize` is used to check that the deployment was successful. This checks that a contract exists at the address, but would, however, return a false positive in the situation where a contract already existed at that address. A separate check in `RocketNodeDeposit.sol` `deposit()` (at line [[]45]) protects against deploying a minipool at the same address as a previous one.

In `RocketNodeDistributorFactory`, no similar check occurs. It is possible for `initialiseFeeDistributor()` and `registerNode()` in `RocketNodeManager` to execute successfully, even though contract creation failed. This also emits unnecessary `ProxyCreated(address(0))` logs that can cause trouble for chain analytics.

This may be more problematic in combination with other issues. However, in the case of RPM-02, it is helpful that `registerNode()` can complete even though someone already deployed a contract at that address. It is preferable to avoid relying on this.

Recommendations

As a best practice, ensure that whenever a low-level `create2` is used, the failure condition is checked by comparing the return value against `address(0)`.

In the case of `RocketMinipoolFactory`, this constitutes no risk to the protocol and is fine to introduce as part of a future update. For `RocketNodeDistributorFactory`, this should be resolved prior to deployment.

In general, prefer to use the high-level Solidity language features unless there is a good reason otherwise. If such a reason exists, document it via in–code comments. For example, the assembly at `RocketNodeDistributorFactory.sol` lines [`39-47`] could be replaced with the following:

```
RocketNodeDistributor dist = new RocketNodeDistributor{salt: ''}(_nodeAddress, rocketStorage)
```

This also has the advantage of enforcing type-safety at compile-time and including a check to revert should the deployment fail.

[5] This can be most easily checked experimentally, and can be confirmed in the yellow paper definitions for instructions `0xf0` and `0xf5`:

$\mu'_s[0] \equiv x$
where $x = 0$ if $z = 0$, i.e., the contract creation process failed, or $I_e = 1024$ (the maximum call depth limit is reached) or $\mu_s[0] > \sigma[I_a]_b$ (balance of the caller is too low to fulfil the value transfer); ([3] p37 — Appendix H.2 instruction `0xf0`)

This states that the head of the stack (containing the result value) is set to 0 on failure. More easily readable reference documentation could not be found that states the failure behaviour.

Resolution

This was resolved in commit 445ef4c by using the high-level Solidity feature, as described above.

`RocketMinipoolFactory` remains unchanged but constitutes no risk.

RPM-08	Inconsistent constructor can lead to unusable contract
Asset	`RocketDAOProtocolSettingsNetwork.sol`
Status	**Resolved:** In commit a7f3484
Rating	Informational

Description

Unlike other `RocketDAOProtocolSettings` contracts, the `constructor()` in `RocketDAOProtocolsSettingsNetwork` does not set any initial settings and does not set the `deployed` boolean.

This structure is not only inconsistent with other similar contracts, but if this contract is deployed with a fresh `RocketStorage`, variables cannot be set due to the `onlyDAOProtocolProposal` modifier checking for a set `deployed` setting.

Therefore a fresh install of all Rocket Pool contracts could leave this contract unusable.

Recommendations

Consider adding the same constructor pattern that is used in the other contracts of this class. Specifically, check if the `deployed` boolean is set; if not, set any required initial variables and then set the `deployed` boolean.

Resolution

The `deployed` boolean is now checked in the constructer and a similar pattern is used for initialising the contracts default settings.

RPM-09	Protocol DAO Treasury Centralisation Risk
Asset	`RocketDAOProtocolProposal.sol` & `RocketDAOProtocol.sol`
Status	**Closed:** See the Resolution
Rating	Informational

Description

Should the guardian account (controlled by the Rocket Pool development team) become compromised, it is possible for it to immediately spend all funds managed by the protocol DAO treasury (the `RocketClaimDAO` contract).

A compromised guardian account may also adjust settings that signal what percentage of RPL inflation rewards should be awarded to the protocol DAO treasury.

In the event of a malicious takeover, the ODAO members may choose to ignore any changes to the value returned by `getRewardsClaimerPerc("rocketClaimDAO")` and instead report via `RocketRewardsPool.submitRewardSnapshot()` a reward of 0 to be sent to the treasury.

Recommendations

Ensure the ODAO and Rocket Pool community are aware that the Rocket Pool *guardian* account has direct control over any RPL funds awarded to the Protocol DAO, as well as the ability to set the portion of RPL inflation rewards that are directed to the protocol DAO.

Consider developing an disaster response plan with the ODAO, so ODAO members know how respond in an emergency.

As it is likely important for some protocol DAO funds to be currently available for use (e.g. for funding maintenance and development of the protocol), disabling access entirely (via a code update) is acknowledged as likely unreasonable.

It may be relevant to limit the amount of funds saved to the treasury until a voting protocol DAO becomes active and the protocol DAO bootstrap mode can be disabled.

Alternatively, may be worth considering implementing a delay so that `bootstrapSpendTreasury()` must signal an intent to spend for some time before the spend can be executed. Optimally, this delay would allow the ODAO to prevent the spend by passing a contract upgrade, in the event of compromise.

Sigma Prime notes that this may not be reasonable, should there be valid reasons to spend treasury RPL at short notice.

Another alternative could be to divide the treasury, so that some portion of the funds are in “deep storage” and only become accessible once bootstrap mode is disabled.

Resolution

The guardian account is planned to be migrated to a 2 of 3 multisig to mitigate the centralisation risk before moving to a more DAO-centric solution.

RPM-10	Incorrect `getFeeDistributorInitialised()` In Some Cases
Asset	`RocketNodeManager.sol`
Status	**Resolved:** In commit 445ef4c
Rating	Informational

Description

The `getFeeDistributorInitialised()` function can incorrectly return `true` in some cases.

Conceptually, a value of `true` returned by `getFeeDistributorInitialised()` is intended to indicate that a `RocketNodeDistributor` contract is deployed for the node operator and `"node.average.fee.numerator"` is correctly set (i.e. that `_initialiseFeeDistributor()` has executed).

The current implementation checks to see whether a contract exists at the expected address, which does not always imply the intended meaning. This meaning does not hold if a contract can be deployed to that address via other means (as occurs in RPM-02). Should it be possible for some other contract to be deployed to that address, `extCodeSize` does not validate that the contract is a `RocketNodeDistributor`.

It is probabilistically infeasible to deploy a contract to the same address without using `RocketNodeDistributorFactory.createProxy()` (provided `keccak256` remains strong), so other scenarios can be safely ignored.

Recommendations

The testing team considers it sufficient to resolve RPM-02 and ensure only `_initialiseFeeDistributor()` can execute the `RocketNodeDistributorFactory.createProxy()` (i.e. that no other network contracts are able to call `createProxy()`).

Alternatively, an arguably safer implementation could involve setting a corresponding storage flag when `_initialiseFeeDistributor()` is executed.

Resolution

This was resolved by mitigating RPM-02, ensuring only the `RocketNodeManager` can execute `createProxy()`.

RPM-11	Revert Error Message Not Propagated
Asset	`RocketNodeDistributor.sol`
Status	**Resolved:** In commit 0cc6d06
Rating	Informational

Description

The `RocketNodeDistributor` contract acts as a proxy, with an underlying `RocketNodeDistributorDelegate` executing the actual logic. Should the `delegatecall` fail or revert, any returned error message is ignored and not propagated to the caller.

The logic implemented in the current `RocketNodeDistributorDelegate` does not return any meaningful error information.

```
25  assembly {
      calldatacopy(0x0, 0x0, calldatasize())
27    let result := delegatecall(gas(), _target, 0x0, calldatasize(), 0x0, 0)
      returndatacopy(0x0, 0x0, returndatasize())
29    switch result case 0 {revert(0, 0)} default {return (0, returndatasize())}
    }
```

As shown at line [29], a `revert(0,0)` with an empty message is executed should the `delegatecall` fail.[6]

However, once deployed, `RocketNodeDistributor` instances cannot be readily replaced or upgraded. As such, the current implementation would make it troublesome (or infeasible) for future upgrades to return specific error messages.

This is of more relevance as recent solidity versions provide the means to `catch` and handle errors differently based on the error message.

Recommendations

Consider modifying the `RocketNodeDistributor.fallback()` to propagate any revert message, as a means of future-proofing.

Resolution

The proxy contract now propagates error messages as recommended.

[6]Refer to https://docs.soliditylang.org/en/v0.7.6/yul.html#evm-dialect

RPM-12	Miscellaneous general comments
Asset	`rocketpool: contract/*`
Status	**Resolved:** In commit d3d1687
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Best Practice Recommendations:**
   - At `RocketUpgradeOneDotOne.sol:159`, the `executed = true` statement should be moved to line [**116**] to abide by the checks-effects-interactions pattern. There is no real risk of re-entrancy exploits in this context though.
   - Consider emitting a log in `RocketNodeDistributorDelegate.distribute()` to allow for more easy data analysis.
   - Add a `uint8 public constant version = 1` to `RocketNodeDistributorDelegate` for consistency with the other network contracts.
   - At `RocketNodeDistributorDelegate.sol:34`, consider replacing `1 ether` with a `calcbase` constant for consistency and readability.
   - Consider implementing a `getSubmitRewardsEnabled()` setting, akin to the existing `RocketDAOProtocolSettingsNetwork.getSubmitPricesEnabled()`.
   - Consider explicitly initialising `rocketStorage` and `nodeAddress` in `RocketNodeDistributorDelegate` to `address(0)` in the constructor for improved readability.
2. **Reused Storage Prefix:**
   In `RocketRewardsPool.sol` lines [**148-149**], the same storage prefix `"rewards.snapshot.submitted.node"` is used for two different types of data — one where the key is based on the entire submission and the other only using the `rewardIndex`.

   Although it is infeasible for these keys to collide, it is somewhat confusing and generally preferable to avoid. Consider renaming the prefix used at line [**149**] to something different.
3. **Redundant Line:** In `RocketNetworkPenalties.sol` the line [**41**] is redundant as the operation on the line above is identical.
4. **Minor optimisations:**
   - At `RocketNodeManager.sol:150`, `getFeeDistributorInitialised()` could have its function state mutability restricted to `view`.
   - At `RocketMinipool.sol:131`, `contractExists()` could have its function state mutability restricted to `view`.
   - At `RocketDAONodeTrustedSettingsRewards.sol:39`, replace `keccak256(abi.encodePacked("rewards.network.enabled", uint256(0)))` with a constant.
   - At `RocketNodeManager.sol:212-216`, the lines can be reordered so the `getUint()` does not execute when `denominator == 0`.

- In `RocketUpgradeOneDotOne.sol`, any addresses or ABI's that can be initialised in the constructor may be set as `immutable` to save on storage gas costs (with a corresponding tradeoff of an increase in bytecode size). As this is only intended to be executed once, these savings are not very important.
- At `RocketNodeDistributorDelegate.sol:22`, the rETH token address could be set as immutable (instead of the key) because it can't be readily upgraded.

5. **Recommended commenting and documentation improvements:**
   - In `RocketNodeDistributorFactory.sol`, consider a comment in `createProxy()` explaining why no salt is needed — that the `initCode` is already unique per node address.
   - In `RocketNodeManager.sol`, consider a comment in `_distribute()` to explain that line [228] will revert if the node operator does not yet have a `RocketNodeDistributor` instance deployed.
   - In `RocketNodeManager.sol`, consider a comment in `initialiseFeeDistributor()` that notes that the loop is safe for all current node operators on mainnet. This may be documented elsewhere, but is also nice to have in the code.
   - In `RocketDepositPool.sol:83`, the variable name `depositNetFee` is slightly confusing. It sounds like the value may include a fee somehow, rather than being the "deposit minus the fee". `depositNet` may be more legible.
6. **Unused variables:**
   The following local variables are unused:
   - `RocketUpgradeOneDotOne.sol:165` `namehash`
   - `RocketUpgradeOneDotOne.sol:187` `namehash`
7. **List of Typos:**
   - There have been unnecessary `string(abi.encodePacked("somestring"))` changes made at `RocketMinipool.sol:117` and `RocketBase.sol:90`.
     Given that these contracts are not readily upgradeable, the changes are assumed to be accidental.
   - `RocketNodeManager.sol:262` "waiting" should be "waited the".
   - `RocketRewardsPool.sol:124,135,144,151` "stake" should be "stack".
   - The comment at `RocketDAOProtocol.sol:74` appears to be identical to that at line [68] and is not relevant for its location in `bootstrapSpendTreasury()`.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The comments above have been acknowledged by the development team, and relevant changes actioned in d3d1687 where appropriate.

Appendix A Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

Impact			
High	Medium	High	Critical
Medium	Low	Medium	High
Low	Low	Low	Medium
	Low	Medium	High
		Likelihood	

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: `https://blog.sigmaprime.io/solidity-security.html`. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: `http://www.dasp.co/`. [Accessed 2018].

[3] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 2014. Berlin Version b2d0dbf – Updated May 2022, Available: `https://ethereum.github.io/yellowpaper/paper.pdf`.