Rocket Pool

Fee Distributor Fix Review

Version: 1.0

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of amendments to the Rocket Pool smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contracts, 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 functionality 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.

A recent Rocket Pool protocol upgrade introduced functionality to support the major Ethereum upgrade to a proof of stake (PoS) consensus mechanism (colloquially known as "the merge").[1] This was the focus of a previous review by Sigma Prime. Part of this upgrade involved introduction of a *Fee Distribution System*: After the merge, validators start receiving fees from transactions when they propose blocks. A representative portion of these fees should rightfully go to `rETH` holders and, as such, a fee distribution system had been introduced.

A problem was identified in this *Fee Distribution System*, which saw the *average node fee* initialised to an incorrect value for existing node operators. A such, any distributed funds were split in a ratio that does not accurately reflect the agreed–upon portion of rewards allocated to the operator as payment for services rendered.

This review is focused on a proposed fix to the fee distribution system, as well as an update to the protocol state to correct the *average node fee* value for each mainnet node operator who had performed the buggy initialisation.

[1] See https://ethereum.org/en/upgrades/merge/.

Security Assessment Summary

This review was conducted on the files hosted on the Rocket Pool contract repository and focused on:

- The bug fix to `contracts/contract/node/RocketNodeManager.sol` at commit b3cae4c (in the branch named "fix-initialise-distributor").
- The proposed state update contract `contracts/contract/hotfix/RocketHotfixNodeFee.sol` at commit fb6d261 (in the branch "v1.1-hotfix").

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

The manual code review section of the report is focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts. This includes 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 [**?**, **?**].

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

- 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 6 issues during this assessment. Categorised by their severity:

- Low: 1 issue.
- 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
RPFD-01	Potentially Unbounded Hotfix Function Execution	Low	Open
RPFD-02	Unsafe Arithmetic and Casting	Informational	Open
RPFD-03	Premature Hotfix Execute Possible	Informational	Open
RPFD-04	Outdated Javascript Dependencies	Informational	Open
RPFD-05	Testing Improvements and Contract Upgrade Methodology	Informational	Open
RPFD-06	Miscellaneous General Comments	Informational	Open

RPFD-01	Potentially Unbounded Hotfix Function Execution		
Asset	`contracts/contract/hotfix/RocketHotfixNodeFee.sol`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `execute()` function loops through all the errors. If the number of errors added is large, this may exceed the block gas limit and become impossible to action.

This does not pose a risk to the Rocket Pool protocol. If the problematic scenario occurred, the impact would be limited to additional gas fees involved in deploying a modified hotfix contract and the relevant ODAO voting.

Because `execute()` only needs to actioned once and is not part of the core protocol, there is no need to consider future functionality in the face of potential gas accounting changes introduced by Ethereum upgrades.

Recommendations

Consider modifying `execute()` to accept an index range as input, allowing execution over multiple transactions. Instead of using an `executed` boolean, this would likely require something like an integer `lastExecutedIndex` that to track what errors have been actioned.

Alternatively add a comment noting how this issue has been considered and accounted for.

If estimating the expected gas costs, consider that it can be possible for a node to have an incorrect `"node.average.fee.numerator"` of 0, which would incur an increased gas cost to modify.[2]

Resolution

The development team have evaluated the expected gas costs incurred when actioning `execute()` on mainnet, and have determined the gas costs to be within the block gas limit.

[2] This may not be the case for any node operator on mainnet but can occur. Consider a scenario where the minipool returned by `getMinipoolAt(0)` is no longer staking and a node operator only has one minipool. After executing the buggy `initialiseFeeDistributor()`, the node operator will have their numerator set to 0.

RPFD-02	Unsafe Arithmetic and Casting
Asset	`contracts/contract/hotfix/RocketHotfixNodeFee.sol`
Status	Open
Rating	Informational

Description

The `RocketHotfixNodeFee` contract's `execute()` function contains unsafe arithmetic and type conversions that do not perform bounds checking. For valid values used in practice this should never overflow and cause problems.[3] However, it remains an advisable practice to use bounds-checked arithmetic and type conversions except in very unusual circumstances.

It also avoids potentially complicated edge cases when verifying off-chain that the errors are correct and non-malicious.

The identified issue occurs at line [**64**] (below):

```
63  uint256 currentValue = getUint(key);
    uint256 newValue = uint256(int256(currentValue) + error.amount);
65  setUint(key, newValue);
```

Because Solidity 0.7.6 uses unchecked arithmetic, large input values can cause the result of `+` to overflow so the result of summing two positive values may be less than the input. Similarly, the `int256()` conversion may return a negative result for large `currentValue` input. This could theoretically cause confusing scenarios where a positive `error.amount` causes an effective *subtraction*.

Recommendations

The testing team recommends replacing the unsafe `int256` and `uint256` casts, as well as the unchecked `+` operator.

While this could be done via the Openzeppelin SafeMath library, it can be more simply replaced by making use of the existing `addUint` and `subUint` functions exposed by the `RocketStorage` contract. This also has added gas saving benefit of halving the number of external contract calls per iteration.

Such a fix could look like the following (note that `error.amount` is a signed `int256`):

```
if (error.amount < 0) {
  subUint(key, uint256(-error.amount));
} else {
  addUint(key, uint256(error.amount));
}
```

[3] The Minipool `nodeFee` is denominated as a portion of `1 ether` so should always be less than 10^{18}. As the `"node.average.fee.numerator"` stores the sum of all an operator's minipool fees, a valid value with potential to cause overflow problems would require the operator to have in the realm of at least 10^{58} minipools!

RPFD-03	Premature Hotfix Execute Possible
Asset	`contracts/contract/hotfix/RocketHotfixNodeFee.sol`
Status	Open
Rating	Informational

Description

It is possible for the guardian account to successfully invoke `execute()` before the `RocketHotfixNodeFee` contract is "locked", and the contract has not yet been accepted as a network contract. This is unlikely to occur and constitutes no risk to the network.

Consider the `execute()` function below:

```
54  // Once this contract has been voted in by oDAO, guardian can perform the adjustments
    function execute() external onlyGuardian {
56      require(!executed, "Already executed");
        executed = true;
58      Error memory error;
        // Loop over list of errors and adjust by error amounts
60      for (uint256 i = 0; i < errors.length; i++) {
            error = errors[i];
62          bytes32 key = keccak256(abi.encodePacked("node.average.fee.numerator", error.nodeAddress));
            uint256 currentValue = getUint(key);
64          uint256 newValue = uint256(int256(currentValue) + error.amount);
            setUint(key, newValue);
66      }
```

It is possible for execute to be called before the contract has been locked; there is no check involving the `locked` state variable.

When invoked without any errors added (i.e. `errors.length == 0`), `execute()` can complete successfully even though the `RocketHotfixNodeFee` instance has not been registered with `RocketStorage`. Normally `setUint()` at line [**65**] would revert when failing to pass the check in the `RocketStorage.onlyLatestNetworkContract` modifier. However, this does not occur when the loop is not entered due to no errors having been added.

Should this occur, the `RocketHotfixNodeFee` contract becomes unusable as `execute()` cannot be called again.

Recommendations

Consider introducing a check in the `execute()` function to explicitly require that `lock()` was previously invoked.

Educate voting ODAO members to not accept a proposal that contains an unlocked `RocketHotfixNodeFee` contract. Consider adding a check to the script in `https://github.com/rocket-pool/verify-v1.1-hotfix/`that warns if the hotfix address is not yet locked.

It is possible to restrict the `lock()` function so that it can only be invoked successfully when the `RocketHotfixNodeFee` contract is *not* a registered network contract. However, this involves additional code complexity and only provides minimal security protections against a negligent ODAO accepting an unlocked contract.

RPFD-04	Outdated Javascript Dependencies
Asset	`package.json` & `package-lock.json`
Status	**Open**
Rating	Informational

Description

The repository depends on several javascript NPM packages that are severely outdated and contain known vulnerabilities. Given the minimal attack surface involved, none of the known vulnerabilities were identified as relevant to the `rocket-pool/rocketpool` repository.

Of primary concern are NPM packages that provide Solidity contract dependencies. While the `@openzeppelin/contracts` package contains known vulnerabilities, these are similarly not applicable.

Recommendations

Consider updating NPM dependencies where possible, with the understanding that these updates should be evaluated to avoid modifying contract bytecode (in order to maintain an equivalence between the repository and the contracts deployed on mainnet). `npm audit fix` is a good starting point. While these vulnerabilities are not of concern to the project, resolving them helps reduce noise, making it easier to identify any new, potentially relevant vulnerabilities.

Consider implementing automated vulnerability notifications, so that the development team can adequately respond should some vulnerability become applicable.

RPFD-05	Testing Improvements and Contract Upgrade Methodology
Asset	`contracts/*`
Status	Open
Rating	Informational

Description

This section details findings and general recommendations regarding the current approach to testing and contract upgrades:

1. **`master` branch test coverage:**

   The targeted `fix-initialise-distributor` branch (built from `master`) contains a fix for a bug identified in `initialiseFeeDistributor()`. However, it does not contain any automated tests that can confirm whether the fix is effective and detect regressions. Indeed, this is not possible with the contracts currently in the `master` branch.

   The testing team notes the intention to maintain the repository's `master` branch such that it reflects the contracts currently used in mainnet. However, the functionality of `initialiseFeeDistributor()` can only be reasonably covered with tests that involve older versions of contracts.

   While it is not crucial that these tests be located in the `master` branch, it is important that they are implemented and maintained. For potential approaches, see below:

2. **Potential long term upgrade architectures:**

   While the mainnet contracts may match the current repository, the mechanism through this was reached is via repeated upgrades and migrations, inherently different to deploying a fresh set of contracts from `master`. The review team recommends introducing automated tests that provide some assurance that these approaches are equivalent.

   This scenario is somewhat akin to database schema upgrades, where it is often desirable to verify that the result of migrations is equivalent to the current schema definition.

   As more contract upgrades are introduced, it will likely become increasingly unwieldy to retroactively make sense of this sequence of upgrades. As such, it is worthwhile considering a more formalised structure and methodology while the number of mainnet upgrades remains low.

   Potential options include:

   - Separate repository to contain integration and upgrade tests. This could reference contracts in the main repository to avoid code duplication.
   - A long-lived `rocketpool-history` branch (or similar), that contains all versions of contracts deployed to mainnet.
     As it is infeasible to refer to more than one revision of contracts as "old", a directory structure should involve contract version numbers (or enumerated upgrades/migrations). This could look like `contracts/contract/vi/.../ContractNameVi` (where `i == ContractName.version`), or `contracts/contract/upgrade_i/.../ContractNameVi` (where the contracts of different versions may be located in the same folder, more closely reflecting the contracts introduced by a particular upgrade)

3. **mainnet fork testing:**

   Consider introducing integration testing that is directly executed on a mainnet fork. `ganache` can do this via the `--fork` and `--fork.network` CLI arguments.[4] This can be useful as a final test prior to deploying an upgrade on mainnet (though note that gas accounting may be slightly different).

4. **Lack of continuous integration testing:**

   There does not appear to be a continuous integration (CI) pipeline that executes tests when changes are made to `rocketpool` repository's master branch.

   This is quite useful for relatively stable code, as protecting against regressions becomes more critical. This also makes updating dependencies a less involved process.

   While it is possible that private CI testing is currently implemented, a publicly reporting CI has the advantage of informing viewers and third parties about exact software versions used and what platforms have been officially tested.

   This also removes the potential for human error in forgetting to test locally (and the associated mental load).

   Consider introducing a CI process that executes tests on commits to `master` and, optionally, pending PRs.

Recommendations

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

[4] Refer to https://blog.infura.io/post/fork-ethereum-replay-historical-transactions-with-ganache-7-archive-support and https://github.com/trufflesuite/ganache#documentation

RPFD-06	Miscellaneous General Comments
Asset	`contracts/*`
Status	Open
Rating	Informational

Description

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

1. **Increment Version on Updated Contracts:**

   Consider incrementing the `RocketNodeManager` contract's version field. While the fix is relatively minor, it does require deploying modified contract code.

2. **Remaining `*Old` Interfaces:**

   The repository's master branch has had previous versions of contracts removed. Previously retained with the naming scheme `ContractNameOld` for upgrade testing purposes (e.g. `RocketMinipoolManagerOld`), these were removed with the goal to have the master branch reflect the contracts currently used in mainnet.

   However, there remain dangling references to these in the form of interface definitions in `contracts/interface/old/` and unused imports in test code. These can also be safely removed.

3. **Outdated `package-lock.json`:**

   The `package-lock.json` is also outdated. When building with `npm install` and nodejs 16.x (the current LTS), the following update occurs.

   ```
   npm WARN old lockfile
   npm WARN old lockfile The package-lock.json file was created with an old version of npm,
   npm WARN old lockfile so supplemental metadata must be fetched from the registry.
   npm WARN old lockfile
   npm WARN old lockfile This is a one-time fix-up, please be patient...
   ```

   This is not a security issue, but commiting this update improves build times when cloning a fresh repository.

4. **Miscellaneous Recommendations:**

   (a) Evaluate whether it is worth adding functionality to remove or clear errors from an unlocked `RocketHotfixNodeFee`.

   This allows for correcting miscalculations without needing to deploy a fresh `RocketHotfixNodeFee` contract. However, it does introduce extra complexity and room for error (making it more difficult for ODAO voters to reason about the contract).

   The testing team acknowledges that, with the current `RocketHotfixNodeFee` implementation, it is possible to add an extra error for a node operator that is the *inverse* of the previous addition. For the purposes of `execute()`, this is basically equivalent to removing an incorrect error from the list. However, current off-chain verification scripts do not allow for more than one entry in the `errors` array per node operator.

   (b) Consider adding a `.nvmrc` [5] or `.tool-versions` [6] to specify the version of nodejs used.

   (c) The `.travis.yml` file is unused and references an out-of-support version 12 of nodejs.[7] Consider removing the file.

[5] Used by the nvm tool.
[6] Used by the asdf tool.
[7] Refer to `https://github.com/nodejs/release#release-schedule`.

Recommendations

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

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.

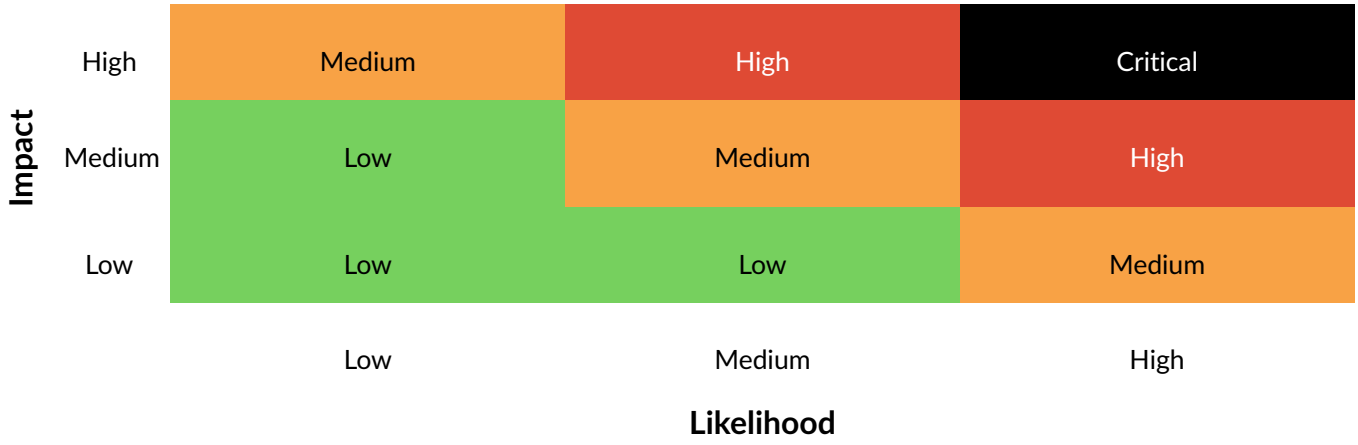

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.