Rocket Pool

Rocket Pool Protocol

Atlas Update Contract Review

Version: 1.0

Contents

Introduction **2**
- Disclaimer . . . 2
- Document Structure . . . 2
- Overview . . . 2

Security Assessment Summary **4**
- Findings Summary . . . 4

Detailed Findings **5**

Summary of Findings **6**
- Lightweight Minipool Initialisation Allows Theft of Operator Refund and State Corruption . . . 7
- Node Operators Can Claim RPL Stake Without Running A Node . . . 8
- Malicious Actors Can Prevent Non-Owner Distributions . . . 9
- Node Operators ETH Can Be Mistakenly Sent To The Deposit Pool . . . 10
- Distinction Of Partial/Full Withdrawals Is Not Guaranteed . . . 11
  - Poorly Performing Validators . . . 11
  - Highly Performing Validators or Long Delay Between Partial Withdrawals . . . 11
- Missing Guards Preventing Direct Initialisation of Minipool Base Contract . . . 13
- Minipool `preDeposit` Submits Entire Contract Balance . . . 14
- Non-Owners May Continually Distribute Balances . . . 15
- Recommendations . . . 15
- The RocketMinipoolProxy Could Be Made Lighter . . . 16
- Miscellaneous General Comments . . . 17

A Vulnerability Severity Classification **19**

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 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. This review is focused on a proposed major update to the protocol, entitled “Atlas”; in particular, focusing on affected smart contracts.

This “Atlas” update is the first major update to Rocket Pool following the Ethereum network’s upgrade to proof of stake (POS) consensus via a “merge” of the consensus and execution layers. This update is also in response to the current mainnet environment, in which the user demand for rETH outpaces growth in node operator’s ability to supply “Minipool” validators (with current collateralisation requirements).

Key features of this update include:

- **Introduction of "Lower ETH Bonded" minipools with an 8 ETH collateralisation requirement (*LEB8s*) —** Node operators can choose to create new minipools backed by either 8 or 16 ETH collateral.
- **A LEB8 migration mechanism —** Allowing node operators to convert existing 16 ETH collateral minipools into a LEB8, with the protocol allocating deposited funds for credit towards collateral for a new minipool.
- **Support for partial validator withdrawals —** As part of an upcoming Ethereum hard-fork, validators will regularly see any balance in excess of 32 ETH withdrawn to their nominated EVM address. This introduces support for early partial withdrawals to the minipool contracts.
- **Using minipool deposit queue ETH —** Use of ETH deposited by node operators to progress the deposit pool and more quickly activate those at the head of queue.
- **Include queue capacity in maximum deposit size —** Allowing single large user deposits to exceed the previous deposit pool maximum, if there are sufficient pending minipools to hold the excess.
- **Reduced minipool contract deployment costs —** New minipool contracts are created with a two-part proxy structure. The deployed minipool is a lightweight, non-upgradeable proxy contract that delegates to a hard-coded proxy containing upgrade logic which, in turn, delegates to the nominated `RocketMinipoolDelegate` implementation. This saves on deployment costs, with a trade-off of increased gas costs for transactions involving those minipools.
- **Removal of total effective RPL stake tracking —** The on-chain effective RPL stake is no longer required and this has subsequently been removed from all smart contracts.

Security Assessment Summary

This upgrade section of this review was conducted on the files hosted on the rocket-pool/rocketpool contract repository and were assessed at commit f95d430.

The review focused on changes introduced as part of the *Atlas* update. In particular, the changes introduced in branch `v1.2` when compared with the master branch (at commit 93f794b)

Subsequent retesting activities targeted commit bea151d, and focused solely on verifying the stated remediations.

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 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 10 issues during this assessment. Categorised by their severity:

- High: 2 issues.
- Medium: 2 issues.
- Low: 3 issues.
- Informational: 3 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
RPA-01	Lightweight Minipool Initialisation Allows Theft of Operator Refund and State Corruption	High	Resolved
RPA-02	Node Operators Can Claim RPL Stake Without Running A Node	High	Closed
RPA-03	Malicious Actors Can Prevent Non-Owner Distributions	Medium	Resolved
RPA-04	Node Operators ETH Can Be Mistakenly Sent To The Deposit Pool	Medium	Resolved
RPA-05	Distinction Of Partial/Full Withdrawals Is Not Guaranteed	Low	Closed
RPA-06	Missing Guards Preventing Direct Initialisation of Minipool Base Contract	Low	Resolved
RPA-07	Minipool `preDeposit` Submits Entire Contract Balance	Low	Resolved
RPA-08	Non-Owners May Continually Distribute Balances	Informational	Closed
RPA-09	The RocketMinipoolProxy Could Be Made Lighter	Informational	Closed
RPA-10	Miscellaneous General Comments	Informational	Resolved

RPA-01	Lightweight Minipool Initialisation Allows Theft of Operator Refund and State Corruption		
Asset	`contracts/contract/minipool/RocketMinipoolBase.sol`		
Status	**Resolved:** The recommended check has been introduced in commit `bea151d`.		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

There are insufficient checks in the `RocketMinipoolBase.initialise()` function, allowing it to be executed successfully by *anyone* on an already–initialised minipool contract. An unprivileged attacker can exploit this in any lightweight minipool contract to drain pending `nodeRefundBalance` ETH and corrupt the contract state, blocking further operation. If exploited by a malicious oDAO majority, this could be used to force a minipool contract upgrade that drains all lightweight minipools of all funds.

The testing team notes that widespread theft of funds is unlikely for attackers who do not control an oDAO majority, and state corruption is the likely chief impact on the wider protocol. In normal operation, `nodeRefundBalance` of an average minipool would be low or zero. If node operators are executing the functions to distribute withdrawals and partial withdrawals, their refund is immediately processed within the same transaction; it is only when network users execute these (e.g. due to an unresponsive node operator) that the `nodeRefundBalance` is credited. It is increasingly unlikely that a large number of minipools hold a significant `nodeRefundBalance` value at the same time.

Recommendations

Prevent `RocketMinipoolBase.initialise()` from executing successfully on contracts that are already initialised.

A check of the following form would be sufficient:

```
require(storageState == StorageState.Undefined, "Storage state not undefined");
```

RPA-02	Node Operators Can Claim RPL Stake Without Running A Node		
Asset	`contracts/contract/minipool/RocketMinipoolDelegate.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

A Node Operator can submit a full withdrawal of their node, receive the ETH from their withdrawal and continue to receive RPL from their staked RPL.

To achieve this state, a Node Operator submits a full withdrawal. They initiate the counter for a user to distribute the funds via, `beginUserDistribute()`. They then distribute funds via another user after the user distribute timeout has elapsed. Finally, they claim the withdrawn ETH via `refund()` without calling `finalise()`.

In this series of events the Node Operators has managed to obtain their owed ETH without running the `_finalise()` function.

The Node Operator's RPL stake remains in the system and they continue to receive RPL staking rewards. If the minipool has a large `nodeFee` that skews the average, they receive a larger share of the tip fee portion distributed them.

The actual reward calculation occurs off-chain so the actual staking rewards are not verified in this review, however it appears that staking rewards are included for non-finalised minipools.

Recommendations

There are a number of ways that this issue might be handled. The resolution should account for this scenario when calculating RPL staking rewards and smoothing pool rewards for Node Operators which refuse to finalise.

Some examples of possible mitigations are:

- Include `decrementNodeStakingMinipoolCount()` as part of a final distribution to allow network users to modify this counter if the operator refuses to.
- Require a Node Operator to mark the minipool as finalised after refunding over a specific amount of ETH.
- Moving the `ethMatched` modification to within `decrementNodeStakingMinipoolCount()`
- Non-finalised minipools could be handled in the off-chain calculations of staking rewards.

Resolution

The authors have acknowledged this issue. The rewards generation process is planned to be upgraded before the launch of the contract upgrade. The upgraded process will exclude validators that have exited the beacon chain.

A governance thread has been started to discuss the rewards generation process.

RPA-03	Malicious Actors Can Prevent Non-Owner Distributions		
Asset	`contracts/contract/minipool/RocketMinipoolDelegate.sol`		
Status	**Resolved:** The recommendation has been introduced in commit bea151d.		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The mechanism that allows any user to distribute capital for a minipool can be subverted by a malicious actor.

The `distributeBalance()` function is able to be called by users who have previously called `beginUserDistribute()`. This starts a timer after which a period of time specified by the `rocketDAOProtocolSettingsMinipool.isWithinUserDistributeWindow()` function, a user can distribute the funds.

This time frame is by default set such that if the function was called between 14 and 16 days ago, a user can distribute the balance.

As `beginUserDistribute()` is an external function with no modifiers anyone is able to call the function. The function also resets the timer. This means a malicious user can continually reset the timer, or front-run transactions to distribute the funds such that it is never possible for a user to distribute the funds.

The likelihood of this issue is low as funds can still always be distributed by the owner, its only the feature that non-owners are able to distribute funds that can be prevented.

Recommendations

The function `beginUserDistribute()` should be prevented from resetting the `userDistributeTime` if the elapsed time is below the upper bound of the `rocketDAOProtocolSettingsMinipool.isWithinUserDistributeWindow()` function.

RPA-04	Node Operators ETH Can Be Mistakenly Sent To The Deposit Pool		
Asset	`contracts/contract/node/RocketNodeDeposit.sol`		
Status	**Resolved:** The recommended check has been introduced in commit `bea151d`.		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

Node operators that have a sufficient deposit credit and additionally send ETH when depositing, lose their additional ETH.

On line [72], the conditional case that an operator has sufficient credit to create a minipool does not check `msg.value` to see if the node also is sending additional ETH.

It can be the case that a Node Operator forgets, or inadvertently sends a deposit value when it has sufficient credit for a deposit bond. The function accepts the ETH without returning the excess to the node operator.

Recommendations

Either return the excess ETH to the node operator or revert the transaction if `msg.value != 0` to prevent excess ETH being lost to the network.

RPA-05	Distinction Of Partial/Full Withdrawals Is Not Guaranteed		
Asset	`contracts/contract/minipool/RocketMinipoolDelegate.sol`		
Status	**Closed:** See the Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The Rocketpool contracts distinguish between a partial and full withdrawal by the amount of ETH sent to the contract. This design has caveats which should be highlighted. There are two extreme cases that are possible which can cause unexpected behaviour in the rocketpool contracts and should be considered.

Poorly Performing Validators

A validator that gets heavily penalized such that its balance becomes ≤ 8 ETH, could withdraw some of the remaining balance rather than returning it all to the rETH holders, as the Rocketpool contracts would consider this a partial withdrawal.

In long periods of non-finality validators are more heavily penalised. A validator that has been penalised for inactivity then slashed (or simply severely slashed) in rare cases can obtain a balance of less than 8 ETH. When withdrawn, this balance will appear to the rocket pool contracts as a partial withdrawal and the slashed validator could withdraw some of the remaining balance.

This can be done by calling `distributeBalance()` on the `RocketMinipoolDelegate` contract.

An example of this would be a node operator that supplied an original bond of 8 ETH and now has a total balance of 6 ETH. The operator would receive:

$$6 \text{ ETH} \times 0.25 + 6 \text{ ETH} \times 0.75 \times 15\%(\text{current node fee}) = 2.175 \text{ ETH}$$

The first term being its share of the rewards and the second term being the fee of the rETH holders rewards.

In this example the node operator has lost 5.825 ETH (72%) and the network has lost 20.175 ETH (84%).

This is illustrated as it should be accounted for in the economic modelling when deciding the risk associated with allowing node operators to supply 25% of the stake. In an adversarial scenario when an operator wants to attack the network, it should be known they can do this in a way that does not consume their entire 8 ETH stake.

Highly Performing Validators or Long Delay Between Partial Withdrawals

Validators could earn more than 8 ETH between partial withdrawal periods making them appear like full withdrawals.

Validators can make more ETH than the usual rewards by submitting valid slashings (a rough estimate is around 130 slashings to make 8 ETH). In addition to this, partial withdrawals are planned to be based on a queuing system which iterates sequentially over the entire validator set. There currently is no technical bounds on the validator set (besides the ETH in existence to be used for staking) and the period between partial withdrawals is proportional to the size

of the active validator set. The longer the period, the greater the average partial validator income will be. Also, any external user could send ETH to a minipool to make a distribution look like a full withdrawal.

Therefore in extreme consensus layer events, such as a single rocket pool validator slashing many other validators it could be the case that a partial withdrawal exceeds the 8 ETH bound. In this case, the owner may accidentally finalise the minipool state, despite it still participating in staking.

Recommendations

As the execution layer does not have good visibility over the consensus layer validator states this issue is difficult to resolve without adding additional minipool states or further information from the consensus layer.

As the likelihood of this attack is low, it is left to the authors to decide if it warrants further engineering to correct.

In the poorly performing case, its important to acknowledge the potential additional risk that it raises should an adversary attempt to attack the network at scale.

Resolution

This has been acknowledged. The Rocketpool team intend to build further monitoring tools in order to keep track and respond to extreme events such as those raised in this issue.

RPA-06	Missing Guards Preventing Direct Initialisation of Minipool Base Contract		
Asset	`contracts/contract/minipool/RocketMinipoolBase.sol`		
Status	**Resolved:** The recommended check has been introduced in commit bea151d.		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `RocketMinipoolBase.initialise()` function does not include important checks to protect it from being executed directly in the context of the RocketMinipoolBase implementation contract. Fortunately, the current `initialise()` code implicitly requires that the `rocketStorage` state variable be already set, as a precondition. As such, this issue is not exploitable in the current codebase.

The residual risk identified with this issue is associated with how easily it may become exploitable through small code changes that could be reasonably introduced by the maintainers.

Recommendations

Prevent `initialise()` from being executed directly in the context of the `RocketMinipoolBase` contract, without setting the `storageState` to `StorageState.Uninitialised` or `StorageState.Initialised` (which may enable other vulnerabilities). This could involve setting `storageState` to an invalid value in a constructor (provided the recommendation for RPA-01 is actioned to require a `storageState` of `Undefined`), or storing the address of the `RocketMinipoolBase` in code and prohibiting execution of `initialise()` when in the context of that address (i.e. that it can *only* be executed via a `delegatecall`).

The latter could look like:

```
contract RocketMinipoolBase {
  address immutable thisRocketMinipoolBase;
  constructor() {
    thisRocketMinipoolBase = address(this);
  }
  function initialise(address _nodeAddress) external {
    require(address(this) != thisRocketMinipoolBase);
    // ...
  }
}
```

RPA-07	Minipool `preDeposit` Submits Entire Contract Balance		
Asset	`contracts/contract/minipool/RocketMinipoolDelegate.sol`		
Status	**Resolved:** The recommended check has been introduced in commit `bea151d`.		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

When submitting an initial deposit to the beacon chain, the entirety of the Minipool contract's balance is sent. In some circumstances, this may not equal the `preLaunchValue` passed to `preDeposit()`.

In order to verify that the node operator has provided valid validator and withdrawal credentials, the Rocket Pool requires operators to submit a small *preDeposit* to the beacon chain deposit contract. Only when the oDAO has had sufficient time to verify the *preDeposit* does it become possible to pass network users' funds to the beacon chain.

At line [**326**] (shown below) the current entirety of the minipool balance is sent, instead of `msg.value` or `preLaunchValue`.

```
function preStake(bytes calldata _validatorPubkey, bytes calldata _validatorSignature, bytes32 _depositDataRoot) internal {
  // ...
  // Send staking deposit to casper
  casperDeposit.deposit{value : address(this).balance}(_validatorPubkey, withdrawalCredentials, _validatorSignature,
      ↪  _depositDataRoot);
  // ...
}
```

It may be currently possible for `address(this).balance` to differ from `prelaunchValue` in the following scenarios:

- When someone has directly transferred ETH to the minipool.
- If there are somehow older minipools (with a direct operator deposit) not yet in the queue. This should not occur.

Recommendations

Confirm whether this behaviour is intentional and consider documenting.

If not intentional, instead send a value of `prelaunchValue` at line [**326**].

Consider also that this may be an easily missed bug in the future, if it becomes possible to have a non-zero refund amount or other balance in the minipool at the time of `preDeposit()`.

RPA-08	Non-Owners May Continually Distribute Balances
Asset	`contracts/contract/minipool/RocketMinipoolDelegate.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

A timing mechanism is implemented that requires non-owners to wait a period of time before distributing balances on minipools that they do not own. In some cases, this time restriction is nullified.

The `userDistributeTime` variable that is used to indicate when a user has indicated that they intend to distribute a balance, is only ever reset on line [**367**], in the conditional case of a partial withdrawal.

The comments on the `userDistributeTime` function are:

```
///        If balance is greater or equal to 8 ETH, users who have called `beginUserDistribute` and waiting the required
///        amount of time can call to distribute capital.
```

This is not strictly true. As the counter gets reset every partial withdrawal, a non-owner must call `beginUserDistribute` again every time a partial withdrawal is executed. Also, there is no resetting of the counter on full withdrawals. Therefore non-owners are not restricted by this functionality to distribute multiple full withdrawals.

Recommendations

This may be the intended logic of the mechanism. It is raised only to ensure this behaviour is as intended.

Resolution

The logic is intended by the authors.

The waiting period is only there to prevent an arbitrageur from distributing user capital back to the deposit pool before the NO has a chance. Allowing the NO to arbitrage provides an incentive for NOs to exit minipools and should help with the rETH/ETH peg.

Resetting `userDistributeTime()` on skim is just a way to prevent bots from calling `beginUserDistribute()` some time long in advance of the full withdrawal.

RPA-09	The RocketMinipoolProxy Could Be Made Lighter
Asset	`contracts/contract/minipool/RocketMinipoolProxy.sol`
Status	**Closed:** The authors are aware.
Rating	Informational

Description

It is possible to minimise the proxy contract to minimise gas costs.

There is known minimal proxy implementations which are documented in [**?**] and [**?**] and may help in reducing the gas costs of the `RocketMinipoolProxy` contract.

The `rocketStorage` state variable could be instead assigned via a parameter to the `RocketMinipoolBase.initialise()` function.

Recommendations

This is raised only to make the authors aware that such implementations exist and could be used to minimise gas costs for proxy deployments.

RPA-10	Miscellaneous General Comments
Asset	`contracts/*`
Status	**Resolved:** Authors have acknowledged all issues and provided corrections
Rating	Informational

Description

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

1. **The `RocketMinipoolDelegate.beginUserDistribute()` function is missing the `onlyInitialised` modifier:**
   The modifier prevents state changes in the deployed implementation contract and should be added as a preferred practice.
   ✓ Resolved in commit [bea151d].
2. **Identified Typographical Errors:**
   - At `test/_helpers/minipool.js:185`, `createVancantMinipool()` should be `createVacantMinipool()`.
   - At `test/minipool/scenario-refund.js:44`, `assertBN.eq()` should be `assertBN.equal()`

   ✓ Resolved in commit [bea151d].
3. **Identified Optimisations:**
   (a) At `contracts/contract/node/RocketNodeDeposit.sol:110`, `getPreLaunchValue()` contains an unnecessary duplicate external call of `getContractAddress("rocketDAOProtocolSettingsMinipool")`. This was already retrieved at line [**104**] and, since the private function is not used elsewhere, its contents can instead be inlined below line [**105**].
   ✓ Resolved in commit [bea151d].
4. **Inconsistent capitalisation convention for ETH:**
   This abbreviation is inconsistently capitalised as *ETH* and *Eth* throughout the codebase. For example, `RocketNodeDeposit` contains functions `_increaseEthMatched()` and `increaseEthMatched()`, but `RocketNodeStaking` contains functions like `getNodeETHMatched()`, `getNodeETHProvided()`. Consider standardising in updated contracts, to mitigate the friction of typographical errors and third-party integrations.
   These can be reviewed via the following CLI commands: `rg --glob '*.sol' ETH` and `rg --glob '*.sol' Eth`.
5. **Unused variables:**
   - At `contracts/contract/minipool/RocketMinipoolDelegate.sol:276`, `launchAmount` is unused (shown below). This is unlikely to pose a problem but should replace the literal value.
     ```
     uint256 launchAmount = rocketDAOProtocolSettingsMinipool.getLaunchBalance();
     userDepositBalance = uint256(32 ether).sub(nodeDepositBalance);
     ```
   - At `contracts/contract/minipool/RocketMinipoolBondReducer.sol` lines [**43,52**], the `minipool` local variable is unused and can be removed.
     ```
     RocketMinipoolDelegate minipool = RocketMinipoolDelegate(_minipoolAddress);
     ```

   ✓ Resolved in commit [bea151d].

6. **Identified comment inaccuracies:**

   - The NatSpec comment for `dissolve()` at `contracts/contract/minipool/RocketMinipoolDelegate.sol:523` states

     ```
     /// Only accepts calls from the minipool owner (node), or from any address if timed out
     function dissolve() override external onlyInitialised {
     ```

     However, this is no longer accurate; all accounts must now wait for the timeout.

   ✓ Resolved in commit [bea151d].

7. **Miscellaneous recommendations and bugs:**

   (a) At `contracts/contract/minipool/RocketMinipoolDelegate.sol:5`, the `MerkleProof.sol` import is unused.
   ✓ Resolved in commit [bea151d].

   (b) At `contracts/contract/minipool/RocketMinipoolBondReducer.sol:8`, it is preferable to instead import the interface `contracts/interface/minipool/RocketMinipoolInterface.sol`, though compilation may optimise to the same result. A corresponding change should be made where the import is used at line [**89**].
   ✓ Resolved in commit [bea151d].

   (c) At `contracts/contract/upgrade/RocketUpgradeOneDotTwo.sol:202`, prefer moving the statement `executed = true;` up to line [**147**] to abide by the checks–effects–interactions best practice that protects against re-entrancy vulnerabilities.
   ✓ Resolved in commit [bea151d].

   (d) Consider introducing into `RocketMinipoolFactory` an explicit "sanity check" that the `RocketMinipoolBase` contract is registered before irrevocably "burning" it into the `RocketMinipoolProxy` instance.

   That is, check that the following `require` statement holds:

   ```
   address rocketMinipoolBase = rocketStorage.getAddress(keccak256(abi.encodePacked("contract.address",
       ↪  "rocketMinipoolBase")));
   require(rocketMinipoolBase != address(0));
   ```

   This is not associated with a vulnerability identified in the current code, as the deployment transaction would revert in a subsequent call to `initialise()` at `RocketMinipoolFactory.sol:63`.

   The testing team recommends including this in the `RocketMinipoolFactory` instead of the proxy constructor in order to reduce the size of the contract creation bytecode.
   ✓ Resolved in commit [bea151d].

   (e) Consider deregistering or removing `RocketMinipoolStatus` as a network contract, now that it serves no purpose. This helps minimise the number of contracts with access to `RocketStorage`, and thus its attack surface.

   (f) The logic in `contracts/contract/minipool/RocketMinipoolDelegate.sol` implementing a distribution cooldown (shown below), appears unnecessary and no longer serves a useful purpose.
   This, and maintenance of the `withdrawalBlock` state variable, appears safe to remove.

   ```
   443 // Rate limit this method to prevent front running
   require(block.number > withdrawalBlock + distributionCooldown, "Distribution of this minipool's balance is on
       ↪  cooldown");
   ```

   ✓ Resolved in commit [bea151d].

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.

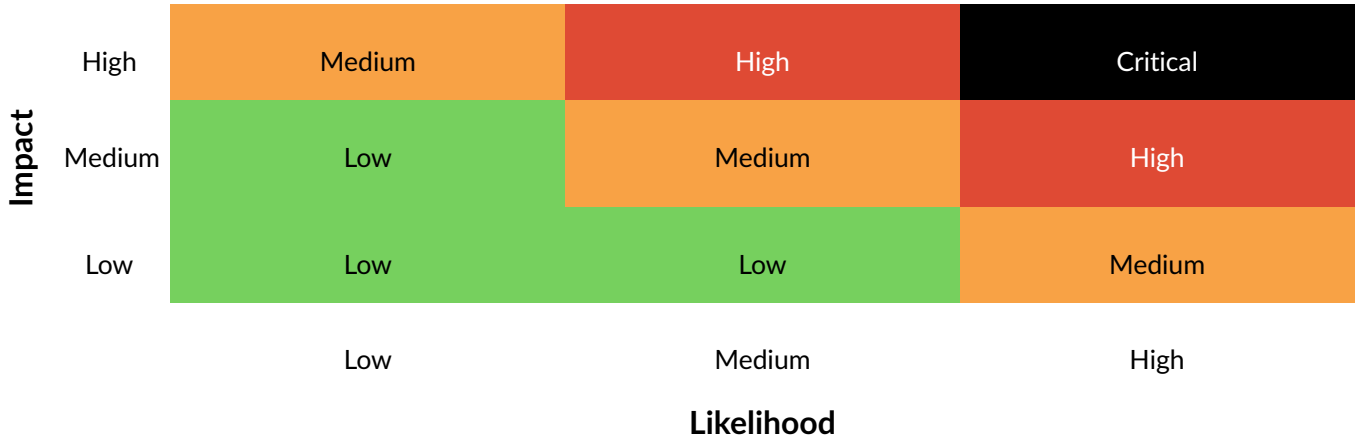

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.