Rocket Pool

Rocket Pool Protocol

Eth2 Deposit Mechanism Review

Version: 1.1

November, 2021

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Rocket Pool smart contract and smart node modifications that are designed to enforce correct Ethereum deposits in the Ethereum 2.0 deposit contract.

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 modifications and their intended purpose. 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, smart node and developer library.

Overview

When making a deposit to the Ethereum 2.0 deposit contract, any deposit multiple of 1 Ether is permitted. When a user deposits Ether into the deposit contract, they submit, among other parameters, their withdrawal credentials. These credentials are used by Ethereum 2.0 clients to send withdrawn staked Ether back to users (once implemented).

A caveat to depositing Ether into the deposit contract, is that only the withdrawal credentials of the first deposit are used. Subsequent deposits' withdrawal credentials are ignored.

For this reason, Rocket Pool's smart contracts must not deposit Rocket Pool's funds under a validator's public key that has already deposited with malicious or unknown withdrawal credentials.

The modifications focused on this report are aimed to ensure that this does not occur. This is done by the following mechanism.

When a node operator creates a *minipool* to deposit Ether, the supplied Ether to the *minipool* is immediately sent to the deposit contract. The *minipool* must then wait a period (currently 12 hours) before Rocket Pool funds are added and the node operator is able to begin staking. During this period, known as the *"Scrub Period"*, ODAO members (a Decentralized Autonomous Organization) watch the beacon chain and verify deposits have the correct withdrawal credentials. Any *minipool* with invalid withdrawal credentials can then be voted to be scrubbed, whereby the *minipool* is removed from the Rocket Pool system, and the operator's RPL tokens are slashed.

Security Assessment Summary

This review targeted the following contracts, focusing exclusively on recent commits that target the deposit mechanism explained in the introduction of this report.

- `rocket-pool/rocketpool`: Smart contracts powering the Rocket Pool protocol.
  - Base commit 7716790
  - Target commit b16ba59
- `rocket-pool/rocketpool-go`: Golang developer library for interacting with the Rocket Pool protocol.
  - Target commit 7963684
- `rocket-pool/smartnode`: Rocket Pool node implementation.
  - Target commit 35e398b

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].

Findings Summary

The testing team identified a total of 9 issues during this assessment. Categorized by their severity:

- High: 1 issue.
- Low: 4 issues.
- Informational: 4 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the changes made to the Rocket Pool platform included in the scope of this assessment. 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 code base, including comments not directly related to the security posture of Rocket Pool (e.g 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
RPF-01	Replay `CREATE2` On Destroyed Minipools	High	**Resolved**
RPF-02	Potentially Insufficient Penalty for Malicious Operator Deposits	Low	**Resolved**
RPF-03	Potential Problems With Node Operator Fund Recovery From Dissolved Minipools	Low	**Closed**
RPF-04	Incorrect Use of Clock Timing	Low	**Resolved**
RPF-05	Unbounded Loop on `getPrelaunchMinipools()`	Low	**Resolved**
RPF-06	Safety Improvements For User Funds Allocated To Unbonded Minipools	Informational	**Resolved**
RPF-07	Resilience Of Price Reporting To Chain Reorganisation	Informational	**Closed**
RPF-08	Unnecessary Validator Activation Delay For FullDeposit Minipools	Informational	**Closed**
RPF-09	Miscellaneous Rocket Pool Issues	Informational	**Closed**

RPF-01	Replay CREATE2 On Destroyed Minipools		
Asset	rocketpool: RocketMinipoolManager.sol		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The opcode `CREATE2` creates a new contract at a deterministically generated address. The address is calculated from the contract bytecode, a salt and the sender address. Since the address is deterministic we can recreate it using the same parameters, however it will not overwrite or modify the bytecode if any exists at the address, i.e. `if (extcodesize(newAddress) > 0)`.

When a contract is destroyed via `selfdestruct`, the bytecode is deleted and the code size is set to zero. It is possible to redeploy a contract to the same address by calling `CREATE2` with the same parameters.

The impact of this result on `deployContract()` is that by using the same `_nodeAddress`, `salt` and `_depositType`, we are able to recreate a new minipool at the same address as one that has been destroyed. Note this does not require the node to use the same validator key or withdrawal credentials as those fields do not impact the address.

This vulnerability can be exploited by following the steps below:

1. An attacker makes a 1 ETH deposit in the Eth2 deposit contract using malicious withdrawal credentials;
2. The attacker makes a 16 ETH deposit in Rocket Pool with the same public key, which creates a minipool and forwards the 16 ETH to the Eth2 deposit contract;
3. The ODAO nodes will see invalid withdrawal credentials and call `voteScrub()` on this minipool;
4. The malicious node operator may then `close()` the minipool causing it to `selfdestruct` (this will require transferring 16 ETH to the minipool that will be immediately refunded);
5. They may now create a new minipool over the same address as the previous one by calling `RocketNodeDeposit.deposit()` with the same `salt` and 16 ETH attached but this time with a new public key so it is not scrubbed. Again, the 16 ETH will be forwarded to the deposit contract under the new public key;
6. 16 ETH of user deposits will then be assigned to the minipool. The attacker is then able to call `stake()` passing the previous malicious public key as a parameter. The transaction will succeed since it passes the following check:

```
require(rocketMinipoolManager.getMinipoolByPubkey(_validatorPubkey) == address(this), "
        Validator pubkey is not correct");
```

As a result, the attacker has deposited 33 ETH (1 ETH in the Eth2 deposit contract, 2 x 16 ETH for each deposit in Rocket Pool) and has 33 ETH balance with a validator that has malicious withdrawal credentials and there is 16 ETH taken from user deposits linked to the second valid public key and withdrawal credentials pointing to the minipool.

Recommendations

We recommend implementing one or more of the following solutions.

- Not *selfdestructing* the contract when it is being closed. Instead, finalise and lock the minipool at the end of `close()`. The remaining contract balance can still be transferred to `rocketTokenRETH`.
- Storing a mapping of each `salt` used by a `_nodeAddress` such as `mapping (address => mapping (uint256 => bool))`. It would then be possible to prevent a node operator from using the same `salt` twice and hence will not be able to recreate a destroyed minipool.
- Have an incrementing nonce, used to make the salt. The nonce should be unique for each validator.
- Include the validator public key in the salt.

Resolution

The issue was resolved by marking destroyed minipools in the `RocketpoolStorage` contract. When new minipools are created a check is performed to ensure that the new address is not currently in use by another minipool and that it is not the address of a destroyed minipool. The resolution can be seen in commit 912c07e.

RPF-02	Potentially Insufficient Penalty for Malicious Operator Deposits		
Asset	`rocketpool: RocketMinipoolDelegate.sol`		
Status	**Resolved:** See Recommendations		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

Although the current two–stage deposit sequence[1] protects against a malicious node operator stealing or locking network funds, the penalties for malicious behaviour may be insufficient in some scenarios. This may allow a malicious actor to flood the network with malicious minipools, draining the balance of ODAO member accounts and hindering their ability to execute `voteScrub()` and other vital tasks. Such an attack would require a very large amount of capital but may not be as costly to the attacker in terms of capital spent.

When a malicious node operator performs the withdrawal credential attack,[2] the ODAO can be reasonably expected to dissolve the minipool via sufficient transactions to `voteScrub()`. However, once beacon chain withdrawals are possible, the attacker should be reasonably able to recover the majority of this deposit back to their withdrawal address, losing a small proportion of their funds to fees.

As the exact details of the beacon chain withdrawal mechanism are yet to be specified, it is unclear what fees are involved and whether the operator would need to deposit an additional 15 ETH in order to top up their balance to 32 ETH and activate the validator, before being allowed to recover the deposit.

In the worst scenario for Rocket Pool, the attacker could quickly recover their partial deposit and pay minimal fees. Then, the attacker could flood the network with malicious minipools, each needing the ODAO members to pay gas fees to dissolve. This might drain funds from sufficient ODAO member accounts as to block consensus. Then, the attacker could be free to perform the full withdrawal exploit to steal network funds or otherwise profit from delayed RPL price reporting.

The viability of this attack hinges on ODAO or community members noticing and ensuring ODAO accounts are appropriately funded before secondary exploits can be executed. Risks are mitigated by an appropriate scrub period and that, in an emergency, anyone can transfer ETH to the relevant ODAO accounts to ensure they remain funded.

Risks associated with a flood of malicious minipools are also mitigated by the protocol DAO's ability to pause deposit assignment in an emergency. This makes it quite unlikely that such attacks could enable secondary exploits after rendering the ODAO temporarily inoperable. However, the need to enact an emergency pause may also incur a reputational cost.

Recommendations

Consider slashing some amount of staked RPL from node operators whose minipool has been subject to "scrubbing". This ensures there is a reasonable cost to malicious behaviour, irrespective of whether invalid staking

[1] In which two separate submissions are made to the staking deposit contract via `prestake()` and `stake()`, with an intermediate delay allowing the ODAO to verify the signature and withdrawal key.

[2] In which they submit an invalid `_validatorPubkey` in their initial deposit to `RocketNodeDeposit.deposit()`, such that the Eth2 staking deposit contract already contains a deposit associating that validator key but withdrawal credentials controlled by the attacker. See for detailed description: https://github.com/rocket-pool/rocketpool-research/blob/master/Reports/withdrawal-creds-exploit.md

deposits can be eventually recovered. It also increases the capital requirements for any associated large scale attacks.

Ensure the scrubbing delay (`RocketDAONodeTrustedSettingsMinipool.getScrubPeriod()`) is sufficient to allow ODAO members to fund their accounts.

Recommendations

This was resolved in commit 76d8506, where an RPL slashing penalty of 10% was introduced for scrubbed minipools.

RPF-03	Potential Problems With Node Operator Fund Recovery From Dissolved Minipools		
Asset	`rocketpool: RocketMinipoolDelegate.sol`		
Status	**Closed:** Acknowledged by the development team		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

In some circumstances, the current `close()` implementation can make it difficult or impossible for node operators to recover all their ETH from a "dissolved" minipool.

The `close()` function is intended to return node operator funds after the minipool was "dissolved" prior to becoming an active validator. Reasons for this can include the operator elapsing, invalid signatures or withdrawal credentials, or simply a change of heart by the node operator.

With recent changes to the deposit sequence, some node operator ETH is immediately submitted to the staking deposit contract.[3] As such, those funds must be withdrawn from the beacon chain consensus layer before they can be recovered by the operator of a dissolved minipool. Because the exact beacon chain withdrawal process has not yet been specified, some potential scenarios can be problematic for operators trying to recover their funds.

- If only a portion of node operator funds are sent to the staking deposit contract, the operator could reasonably want to immediately recover any accessible funds (rather than waiting for all their balance to be withdrawn to the minipool).

  This currently affects dissolved minipools with a `depositType` of `MinipoolDeposit.Full` (provided they had not received a user deposit), who would have all of their 32 ETH locked even though 16 ETH could be returned immediately. This would also affect other nodes should the `prelaunchAmount` be reduced so that it is less than the `nodeDepositBalance`.

- Fund withdrawal for validators with partial deposits may not be implemented, or may be implemented after withdrawal for active validators.

  The node operator may want to deposit additional ETH to the staking deposit contract, in order to increase the validator balance to 32 ETH and perform a withdrawal. With the current `close()` implementation, this additional ETH would be sent to the rETH token contract instead of being returned to the operator.

- The beacon chain withdrawal semantics could involve gas costs being deducted from the withdrawal amount.

  As such, the minipool would not receive the entire `prelaunchAmount` and the operator would need spend extra transaction fees to transfer ETH to the minipool and top up its balance so that it equals `nodeBalance` (as defined at line [438]). Only then could they successfully execute the `close()` and retrieve their funds.

While a problematic user experience for node operators, none of these scenarios render funds irrecoverably locked and they can be retroactively resolved by minipool contract upgrades. Those most often affected by these issues would be operators who are malicious, negligent, or incompetent. As such the severity of this issue is deemed *low*.

[3] Also known as "the beacon chain deposit contract" and, previously, "the Eth2.0 deposit contract".

Recommendations

Consider introducing the following changes for dissolved minipools:

1. Allow the node operator to recover any funds of theirs that are currently in the minipool without destroying the contract, to avoid unnecessarily locking funds until all are retrieved from the beacon chain.
2. Return excess funds to the node operator (as long as all user funds have been returned to the network).

   Alternatively, if this is a concern, it may be reasonable to send the excess to a vault controlled by the protocol DAO, to whom the operator can provide proof of ownership and petition for return of their funds.
3. Allow operators to close a minipool which has a balance near their owed amount (`nodeDepositBalance + nodeRefundBalance`), to account for possible deductions that cover withdrawal costs.

Other than item 1 enabling node operators to recover a portion of their funds, the other recommendations have no effect until the beacon chain withdrawal process is implemented. As such, any relevant changes can be introduced in later upgrades to `RocketMinipoolDelegate`, once the withdrawal process has been finalised. These are highlighted in the current report to ensure awareness.

A smaller `prelaunchAmount` will also help reduce the amount of operator funds locked for a long period.

RPF-04	Incorrect Use of Clock Timing		
Asset	`smartnode: rocketpool/watchtower/`		
Status	**Resolved:** Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The ETH1 API call `eth_syncing` may return a false positive in the cases where the node believes it has the latest head even though it may not. This often occurs for the first minute when booting a node. The impact is that the client may pass the check `service.WaitEthClientSynced()` which forces the Rocket Pool smartnode to wait for the ETH1 node to be synced.

As a result when the Rocket Pool smartnode is executing `dissolve-timed-out-minipools` or `submit-scrub-minipools`, the local clock time may be significantly ahead of the clock time of the latest seen head block (note the latest seen block is where we get our data and may be significantly behind the network head).

These discrepancies may cause either the node to believe some minipools have timed out or vote to either scrub them or dissolve them. This can be seen in the following example where `time.Since()` reads the local clock time. Since we have an outdated head, we may timeout the pool even though they have called `stake()` in a more recent block which was not received.

```
if statuses[mi].Status == types.Prelaunch && time.Since(statuses[mi].StatusTime) >= launchTimeout {
    timedOutMinipools = append(timedOutMinipools, mp)
}
```

The impact with relation to scrubbing is that if an operator relies on their local clock to check if a minipool has passed 1/2 the scrub timeout, we may vote to scrub this minipool as the ETH1 node has not seen the blocks which contain the correct deposit logs.

This issue is rated as a low likelihood since the probability of receiving a false positive from `eth_syncing` is small. In the case of scrubbing, it is highly unlikely that a sufficient number of nodes will have this issue at the same time to reach the required quorum to successfully scrub a minipool. In the case of dissolving, the transaction will revert if the node operator has called `stake()` and thus the cost will be the gas fees of the reverted transaction.

Recommendations

We recommend using the timestamp from the latest block rather than the local clock to account for any potential discrepancies between the two.

Resolution

Submitting scrub minipools has been updated to use the latest block time in commit f3a3718 and dissolving timed out minipools has been updated in commit 4ef9dc3.

RPF-05	Unbounded Loop on `getPrelaunchMinipools()`		
Asset	`smartnode: rocketpool/watchtower/submit-scrub-minipools.go`		
Status	**Resolved:** Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

There is an unbounded loop in the function `getPrelaunchMinipools()` when the parameter `limit` is set to zero. This is unbounded based off the number of minipools.

While the default deployment sets an initial limit of 14 on the total number of minipools allowed, this is intended to increase after an initial launch period and the limit may be removed entirely.

`rocketpool-go/minipool/minipool.go::GetPrelaunchMinipoolAddresses()` is called from the watchtower to fetch all the minipools with a `PreLaunch` status. `limit` to zero when called from the watchtower thereby creating an unbounded loop.

The unbounded loop will behave differently for different ETH1 implementations. In Geth, the default parameters are to timeout if the call takes longer than 5 seconds or if more than 50 million gas is used.

If this function were to fail then ODAO nodes would no longer be able to scrub malicious minipools and thus it would be possible to perform an attack in switching the withdrawal credentials for malicious ones.

Recommendations

We recommend setting a non-zero value for `limit` in `getPrelaunchMinipools()` the number of minipools fetched.

Alternatively, adjust the watchtower implementation to fall-back to paginating with a safe, non-zero limit should the unbounded call to `getPrelaunchMinipools()` fail.

Resolution

The recommendation was implemented in commit 9710d9a by setting the `limit` to 750.

RPF-06	Safety Improvements For User Funds Allocated To Unbonded Minipools
Asset	`rocketpool: RocketMinipoolDelegate.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Unlike conventional minipools that require a node operator deposit of 16 ETH (`MinipoolDeposit.Half`), all funds staked via an unbonded minipool belong to the network's rETH holders. As such, the two–stage `preStake()` sequence risks network ETH instead of the node operator's.

The current `prelaunchAmount` of `16 ETH` unnecessarily risks delivering network funds to a malicious ODAO member. A `prelaunchAmount` of `1 ETH` is sufficient to verify that the validator's withdrawal address is correctly registered.

Given the current settings requiring a large bond amount for ODAO membership and restricting the number of unbonded minipools, this issue is classed only as informational.

Recommendations

Given the upcoming launch, the testing team deems it perfectly reasonable to avoid addressing this issue until a later update, after the launch has completed successfully.

Evaluate whether the ODAO member bond and limits to the number of unbonded minipools (`"members.minipool.unbonded.max"`) sufficiently outweigh the damage a malicious ODAO member can inflict on the network through the creation of unbonded minipools with invalid signatures. This should also take into account reasonable delays involved with voting to kick the malicious member from the ODAO, and to what extent indirect profits (e.g. shorting RPL) may help counterbalance the penalties incurred in being kicked from the ODAO.

If the existing restrictions are sufficient, consider removing the two–stage deposit sequence (pre-stake then stake) for unbonded minipools, as both stages involve user funds anyway. This would result in gas savings to the node and ODAO for unbonded minipools, which can be passed onto the network.

Otherwise, consider adjusting the `prelaunchAmount` to `1 ETH` for unbonded minipools. This minimises the user ETH a malicious ODAO member can lock, while providing stronger safety assurances; that an unbonded minipool with a "Staking" status has had its withdrawal address vetted by a majority of ODAO members.

Also consider that a second deposit via `stake()` can also have an invalid signature. This two–stage deposit prevents the ODAO member from stealing network funds, they may still "burn" the ETH using an invalid signature. If later needing to increase the limits to the number of unbonded minipools, it may be worthwhile to implement a separate restriction on the number of unbonded minipools whose deposit signatures have not yet been vetted (i.e. those which have a status of `Prelaunch` or where the signature passed to `stake()` has yet to be independently validated).

Resolution

The unbonded minipool functionality has been removed in commit c98be45 for the short/medium term. It is planned to be reintroduced in a post-launch upgrade, and can be prioritised if there are insufficient node operators to meet demand.

RPF-07	Resilience Of Price Reporting To Chain Reorganisation
Asset	`rocketpool: RocketNetworkPrices.sol`
Status	**Closed:** Acknowledged by the development team
Rating	Informational

Description

Given the current method used by compliant ODAO members to select which block to report price information for, they may occasionally submit price information for blocks near or at the head of the canonical ethereum chain, which will occasionally be subject to a chain reorganisation ("reorg").

`getLatestReportableBlock()` is used by ODAO members to check the most recent block for which to report price information. Given this can can return `block.number` and the current watchtower implementation queries the head of the chain (latest block), small 1-2 block chain reorg events[4] may affect the validity of the data submitted on-chain.

Although this may be possible, given the randomised delay with which ODAO members poll `RocketNetworkPrices` to see if they should submit RPL price information,[5] there is a negligible risk that this simultaneously affects a significant portion of compliant ODAO members.

As such, the risk to the network is deemed negligible, and this issue is raised as *informational* to avoid occasional wasted gas due to ODAO members submitting invalid price information.

Recommendations

Consider modifying `getLatestReportableBlock()` to introduce a small delay so that the block number returned is always at least a few blocks removed from the chain head. Reorgs more than 2 blocks deep are extremely rare during normal network conditions.

Such a change could look like the following change to line [148]. Existing:

```
uint256 latestReportableBlock = block.number.div(updateFrequency).mul(updateFrequency);
```

Potential mitigation:

```
// some small block delay that is unlikely to be subject to a chain reorg
uint256 constant REORG_SAFETY = 6;
// ...
uint256 latestReportableBlock = block.number.sub(REORG_SAFETY).div(updateFrequency).mul(
        updateFrequency);
```

Alternatively, `(*submitRplPrice).getLatestReportableBlock()` in `submit-rpl-price.go` could be adjusted to make an `eth_call` to a slightly older block. This requires no changes to contract code but is more easily incorrect.

[4] These occur regularly and semi-regularly, see https://etherscan.io/blocks_forked

[5] Implemented in `smartnode: rocketpool/watchtower/watchtower.go`

RPF-08	Unnecessary Validator Activation Delay For FullDeposit Minipools
Asset	`rocketpool: RocketMinipoolDelegate.sol`
Status	**Closed:** Acknowledged by the development team
Rating	Informational

Description

Minipools created with an initial node deposit of 32 ETH (`MinipoolDeposit.Full` , a.k.a `fullDeposit`) do not need to delay staking by some `scrubPeriod` in order to mitigate exploits associated with bad withdrawal credentials or invalid signatures.

Because the node operator funds are sufficient to activate the beacon chain validator, 32 ETH can be immediately submitted to the staking deposit contract without putting any network funds at risk. This minimises activation delays, allowing the operator and network to more quickly start earning staking rewards.

The existing `scrubPeriod` may offset the primary reason for creating a `fullDeposit` minipool in the first place.

This issue is raised as a potential optimisation and constitutes no risk to the network. Indeed, implementing a separate deposit sequence for `fullDeposit` minipools introduces some complication which should be carefully considered and tested.

Recommendations

Given the upcoming launch and potential complications associated with these changes, the testing team deems it perfectly reasonable and likely preferable to avoid addressing this issue until a later update (after the launch has completed successfully).

Consider modifying the deposit sequence for full deposit minipools to allow 32 node operator ETH to be immediately submitted to the beacon chain deposit contract, to minimise validator activation delays.

In order for this to be safely implemented, it would be important to instead prevent node operators from withdrawing refunds from `fullDeposit` minipools until the `scrubPeriod` has elapsed. This protects network funds until the deposit signature and validator key has been appropriately validated. A “scrubbing” could rescind the `nodeRefundBalance` amount (or just remove the balance), but would need to ensure the operator could recover their funds once the beacon chain withdrawal has returned the validator balance. This may also complicate the state machine, in that one implementation could involve scrubbing a minipool with a `Staking` status.

RPF-09	Miscellaneous Rocket Pool Issues
Asset	`smartnode/*` & `rocketpool/*`
Status	**Closed:** Acknowledged by the development team
Rating	Informational

Description

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

1. For the two–stage staking deposit sequence involving minipool `prestake()` and `stake()` functions, the testing team recommends a change to the `prelaunchAmount` so that both deposits include some Node Operator funds.

   This means an invalid deposit signature will always directly burn node operator funds (e.g. with `prelaunchAmount = 8 ether`, 8 node operator ETH is immediately lost whenever they make an invalid deposit signature). Although the current `RocketMinipoolDelegate` contract makes any operator funds inaccessible until the network has been fully reimbursed, this change protects against a slight risk of buggy minipool contract upgrade.

2. Given signature verification functionality has now been implemented in the ODAO watchtower software, it may be reasonable to also validate the signatures passed to `RocketMinipoolDelegate.stake()`. An invalid signature could allow the node to immediately have their RPL slashed and stopped from earning RPL staking rewards. Perhaps a separate "Failed" status would be useful? As an invalid signature means the validator will not have enough funds to activate (without the node operator putting in an extra 16 ETH), it will not normally be reported as withdrawable by the ODAO. Although the node operator would have more funds locked indefinitely than the network, slashing the RPL would also allow the network to recover some of these losses.

   This involves some extra complication and is, at most, a "nice to have".

3. At `smartnode/rocketpool/watchtower/submit-rpl-price.go`, it is preferable to perform the `eth_call` query to `calculateTotalEffectiveRPLStake()` using the appropriate block number, as is done for `(*submitRplPrice).getRplPrice()`, rather than the default "latest".

   The contracts ensure that changes to effective RPL stake cannot occur until price reporting has reached consensus for that period, so this currently has no effect on the result's correctness.

4. In `smartnode/rocketpool/watchtower/submit-rpl-price.go`, the ODAO members can save gas costs by not submitting `submitPrices()` transactions for blocks that have already had their quorum reached. While these savings may then be passed onto the network in the form of reduced ODAO RPL rewards, some complications are introduced in balancing gas costs evenly across members. Unresponsive members may also be less easily identifiable.

5. In `RocketMinipoolDelegate.sol`, there doesn't seem to be a reason to use a separate `finalised` state variable (only "withdrawable" minipools are ever finalised so there is no ambiguity). This could instead be a `RocketMinipoolStatus` value, saving the storage costs and associated gas for node operators. However, this is not a regularly executed code-path, so the total savings are likely not worth an upgrade.

6. In `RocketMinipoolDelegate.sol`, there is no real reason for `stake()` to take a `_validatorPubkey` parameter, as this is effectively retrieved from storage anyway (after being set during `preStake()`).

7. If wanting to enforce that `RocketNodeStaking.calculateTotalEffectiveRPLStake()` is not misused and only ever executed off-chain, this can be done by requiring an impossible sending address.

```
require(msg.sender == address(0))
```

`address(0)` can be specified as the `from` address for an `eth_call`, but can never be used in a transaction. However, this is not really a problem, and contracts can safely use `calculateTotalEffectiveRPLStake()` if they pass safe values to the `offset` and `limit` parameters.

8. `smartnode/rocketpool/watchtower/submit-scrub-minipools.go` commented out code on line [269]

```
/*
data, err := t.bc.GetEth1DataForEth2Block("finalized")
if err != nil {
    return nil, err
}

eth1Block, err := t.ec.BlockByHash(context.Background(), data.BlockHash)
if err != nil {
    return nil, err
}
*/
```

9. `smartnode/rocketpool/watchtower/dissolve-timed-out-minipools.go` commented out code on line [163]

```
// Log
//t.log.Println("Checking minipools %d - %d of %d for timed out status...", msi + 1, mei, len(minipools))
```

10. At `contracts/contract/dao/protocol/settings/RocketDAOProtocolSettingsRewards.sol` line [26], the comment states "14 days by default" but the default is now 28 days.

11. At `contracts/contract/dao/node/settings/RocketDAONodeTrustedSettingsMembers.sol` line [16], the comment uses the spelling "Initialize", and appears to have been missed in recent code changes to standardise the spelling to "Initialise".

12. At `RocketMinipoolDelegate.sol` line [463], in the NatSpec comment for `voteScrub()`, the use of "it's" in this context is incorrect and should be "its".

```
// Can be called by trusted nodes to scrub this minipool if it's withdrawal credentials
        are not set correctly
```

Recommendations

Ensure that the comments above are understood and acknowledged, and consider removing unused, commented out code.

The testing team also acknowledge the upcoming launch, and note a preference to delay implementing recommendations that involve significant code changes, as opposed to rushing them through.

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.

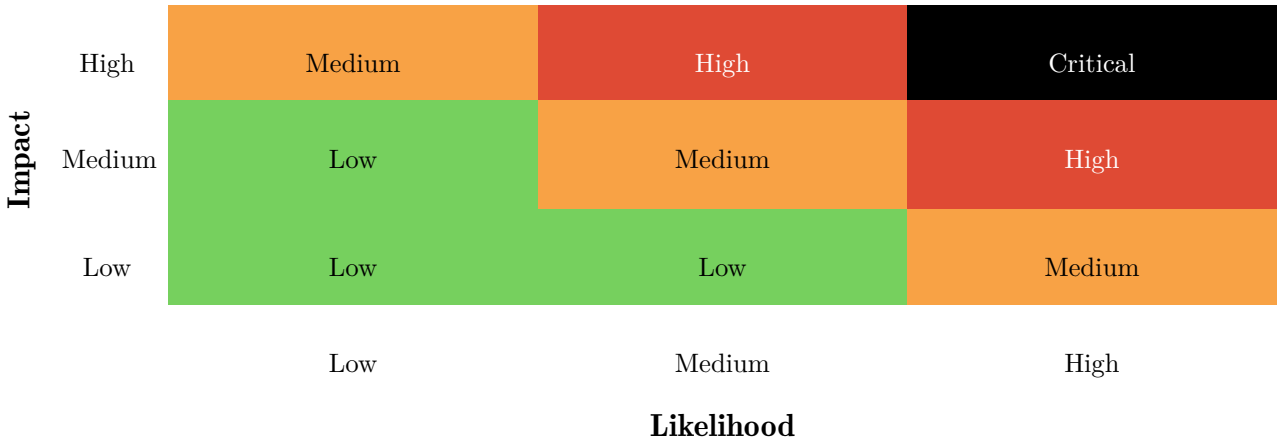

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

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].