Rocket Pool

Rocket Pool – Houston Upgrade

Smart Contract Security Review

Version: 2.4

June, 2024

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 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

The Houston upgrade is largely aimed at introducing a fully onchain DAO to govern the protocol, known as the Protocol DAO or pDAO. It is a truly onchain DAO that does not require snapshot voting or any other 3rd party tools to function.

The upgrade also introduces some other features allowing new integrations and platforms to be built on the protocol. Some of these include the ability to stake ETH on behalf of node (not just from the node itself) and a new RPL withdrawal address feature that can allow the node operator to supply the ETH for staking and another party to trustlessly provide the RPL for the insurance bond.

Security Assessment Summary

This review was conducted on the files hosted on the Rocket Pool repository and were assessed at commit f26996f.

The scope of the review was limited to the following diff, encompassing all changes and additions made to the contracts in relation to the following:

- The onchain Protocol DAO (pDAO): RPIP-33
- Stake ETH on behalf of node: RPIP-32
- New RPL withdrawal address: RPIP-31

An additional review, limited to pDAO specific functions, was conducted on Rocket Pool repository and was assessed at commit 7215562c.

Two follow up reviews were also conducted on Rocket Pool repository and were strictly limited to changes introduced in the commits e760442 and 7161d1c.

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 [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.

Coverage Limitations

Due to a time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 19 issues during this assessment. Categorised by their severity:

- Critical: 2 issues.
- High: 4 issues.
- Medium: 4 issues.
- Low: 3 issues.
- Informational: 6 issues.

Note: considering the large number of critical and high severity issues identified during this time-boxed engagement, Sigma Prime recommends further security testing on the code base in scope prior to any deployment.

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
RPH-01	ETH Locked In `RocketNodeDeposit` Contract	Critical	Resolved
RPH-02	Challenges Can Be Manipulated Through Arbitrary Path Traversal To Steal Proposal Bond	Critical	Resolved
RPH-03	Upgraded Contracts Threaten Proposal Bond Liquidity	High	Closed
RPH-04	Incorrect Offset When Setting `periodsPaid`	High	Resolved
RPH-05	Incorrect Computation Of `claimIntervalsPassed`	High	Resolved
RPH-06	Incorrect RPL Stake Calculation During Withdrawal	High	Resolved
RPH-07	Behavioural Inconsistencies In Protocol Settings Initialisation Code	Medium	Closed
RPH-08	Incorrect `stake.for.allowed` Value	Medium	Resolved
RPH-09	Challenged Leaf Indices Can Be Left Unverified Onchain	Medium	Resolved
RPH-10	Challengers Can Contest Non-Existent Indices To Steal Proposal Bond	Medium	Resolved
RPH-11	Implementation Discrepancies With RPIP Specifications	Low	Resolved
RPH-12	No Checks For Pending Withdrawal Addresses	Low	Closed
RPH-13	`*Old.sol` Contracts Do Not Match Current Version On Mainnet	Informational	Resolved
RPH-14	Snapshot Amendments May Lead To Leaf Verification Failures	Low	Resolved
RPH-15	Initial Votes Can Be Cast In Phase 2 Without Proof	Informational	Closed
RPH-16	Node's Votes Can Be Overridden Without Its Knowledge	Informational	Closed
RPH-17	Delegates Are Unable To Vote	Informational	Closed
RPH-18	Inadequate Merkle Height Verification	Informational	Closed
RPH-19	Miscellaneous General Comments	Informational	Closed

RPH-01	ETH Locked In `RocketNodeDeposit` Contract		
Asset	`RocketNodeDeposit.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Specific transactions using `depositWithCredit()` will result in ETH supplied as bond being stuck in `RocketNodeDeposit` contract indefinitely.

The issue arises when calling `depositWithCredit()` with `msg.value == 0`, but preloading the node with sufficient balance. The downstream function `_processNodeDeposit()` has not been updated to cater for situations where `msg.value` could be zero. Current implementation uses the `msg.value` assuming it will always be equivalent to a supplied bond amount, which may not be the case anymore.

In `depositWithCredit()`, if there is an existing node balance to use, it will be withdrawn from the vault into `RocketNodeDeposit` contract. However, in the following code from `_processNodeDeposit()`, if `msg.value` is zero (i.e. only using existing balance), the `_preLaunchValue` of 1 ETH will also be taken out of the vault and deposited into `RocketNodeDeposit` contract (to be used in funding the minipool later). The leftover value `remaining` will be zero, and as such, no deposits will be made back into the vault, resulting in the ETH balance being stuck in the `RocketNodeDeposit` contract.

Note, the deposit pool's `deposit.pool.node.balance` internal accounting variable will still be increased by `_bondAmount - _preLaunchValue` via `nodeDeposit()`:

```
if (msg.value < _preLaunchValue) {
    shortFall = _preLaunchValue- msg.value;
    rocketDepositPool.nodeCreditWithdrawal(shortFall);  // @audit this will still withdraw 1 ETH for prelaunch
}
uint256 remaining = msg.value + shortFall - _preLaunchValue;
// Deposit the left over value into the deposit pool
rocketDepositPool.nodeDeposit{value: remaining}(_bondAmount - _preLaunchValue);
```

As a final execution step, through `assignDeposits()` call, ETH balance will be withdrawn from the vault yet again, and deposited into the minipool to complete its funding. The ETH initially withdrawn by `depostiWithCredit()` will not be used and will remain locked in the `RocketNodeDeposit` contract.

Recommendations

Modify implementation of `_processNodeDeposit()` to cater for situations where `msg.value` could be zero, if the node has been preloaded with a sufficient balance. Also, do not assume that `msg.value` will be 8 or 16 ETH, as `balanceToUse` now also needs to be considered when calling `_deposit()` on line [**166**].

Revisit the need of calling `rocketVault.withdrawEther(balanceToUse);` in `depositWithCredit()`, considering that required ETH is already being withdrawn from the vault via `assignDeposits()` call.

Resolution

The issue has been addressed in commit e2557ce by modifying the implementation of `_processNodeDeposit()` to no longer rely on `msg.value`, but to use the contract balance instead.

RPH-02	Challenges Can Be Manipulated Through Arbitrary Path Traversal To Steal Proposal Bond		
Asset	`RocketDAOProtocolVerifier.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Due to errors in the `computeRootFromWitness()` function, users can craft specific challenges that will place valid proposals in a challenged state, where future challenges may not be contested, genuine proposals are blocked from voting and, subsequently, leading to the proposal bond being stolen.

The `RocketDAOProtocolVerifier.sol` contract is responsible for ensuring that proposal creations, votes and vetoes can only happen by users meeting relevant requirements. When a proposal is created, the proposer submits a Merkle pollard containing an aggregated voting power sum along with a proposal bond. Proposals can be challenged by any registered node prior to voting and winning challenges claim the proposal bond.

Proposals contain two Merkle trees combined by way of pollards. Both trees contain a power of two multiple of the number of RocketPool nodes. The first tree is the Network Voting Tree containing an aggregated voting sum of each node in the network. The second tree, which extends from the leaf nodes in the first tree, contains the delegate voting power assigned to each account by the node index in the first tree.

When a user submits a proposal, they provide a Merkle pollard for the Network Voting Tree with a maximum depth of 5. Challengers are then able to contest branches of this pollard by submitting an index they would like to challenge, along with a witness and leaf that hash to the root of the tree.

The function `computeRootFromWitness()` accepts an index, a leaf node and a witness, and produces a root hash. However, it does not validate that the witness length matches the size expected for a given index depth. This allows a malicious party to submit a valid witness for a given leaf, but choose any index that follows the same path in order to produce a matching hash.

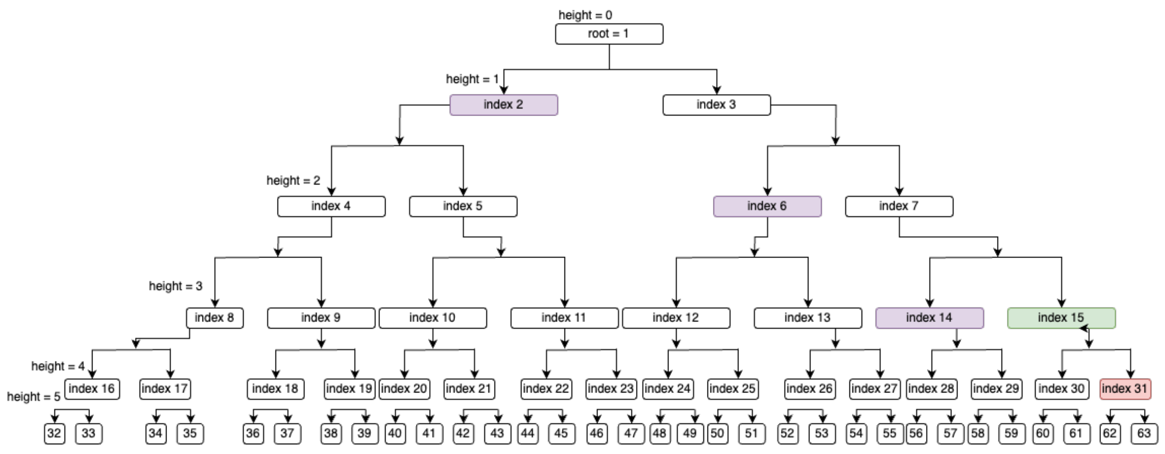

Figure 1: Example of a Merkle tree with 32 Rocket Pool nodes.

Let's consider an example above with 32 Rocket Pool nodes, this corresponds to a Network Voting Tree depth of 5. A challenger decides to challenge the proposal, providing the correct leaf node for index 15 along with the matching witness for index 14, 6 and 2. Under normal conditions if the challenger provided the index 15 as the actual challenge, we would hash values 15 and 14 to receive 7, this is hashed with index 6 and then the result is hashed with index 2 to calculate the root. If the calculated root matches the expected root, the challenge successfully updates the node to the challenge index and sets the hash to the value contained in the leaf.

Consider a modification to the above example, a malicious user decides to challenge index 31 (the right hand side layer down from index 15), provides the leaf for index 15, instead of 31, and provides the witness required for leaf index 15 (ie indexes 14, 6 and 2). Since the witness is hashed against the leaf, until all witness values are exhausted, the hash follows the same path from 15 to the root. This means the computed root for this malicious challenge index of 31 still matches the expected root hash value. This sets the root hash of the index 31 challenge to the value of the index 15. Since the hash no longer matches what the tree would expect for the index 31, the tree is now left in a challenged state that is not able to be responded to. This subsequently leads to the loss of the proposal bond.

Recommendations

The testing team recommends the following fixes to `computeRootFromWitness()`:

- Validate that `computeRootFromWitness()` contains the required amount of witnesses for the relevant index depth
- Check that the `_index` value in `computeRootFromWitness()` at the end of the for loop is the same as the expected root node before returning, revert if it is more. For example, it would need to ensure that for the first pollard, it finalises to an index 1 and revert if it is more.

Resolution

The issue has been addressed in commit 4857aff by adding a check that the proof length matches the depth of the challenge.

RPH-03	Upgraded Contracts Threaten Proposal Bond Liquidity		
Asset	`RocketDAOProtocolProposal.sol`, `RocketDAOProtocolVerifier.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

Proposals submitted and not finalised prior to a network upgrade, may result in locked funds of the proposers and challengers.

RocketPool uses a centralised contract to manage versions and implementation details for contracts within their ecosystem. Both the `RocketDAOProtocolProposal` and the `RocketDAOProtocolVerifier` contracts have access control mechanisms that ensure only the latest versions of those contracts will execute critical functions.

The following functions all have version based access control:

1. `onlyLatestContract("rocketDAOProtocolProposal", address(this))`:
   - `RocketDAOProtocolProposal.vote()`
   - `RocketDAOProtocolProposal.overrideVote()`
   - `RocketDAOProtocolProposal.finalize()`
   - `RocketDAOProtocolProposal.execute()`
   - `RocketDAOProtocolProposal.destroy()`
   - `RocketDAOProtocolProposal.vote()`
2. `onlyLatestContract("rocketDAOProtocolVerifier", address(msg.sender))`:
   - `RocketDAOProtocolProposal.destroy()`
3. `onlyLatestContract("rocketDAOProtocolVerifier", address(this))`:
   - `RocketDAOProtocolVerifier.submitProposalRoot()`
   - `RocketDAOProtocolVerifier.burnProposalBond()`
   - `RocketDAOProtocolVerifier.createChallenge()`
   - `RocketDAOProtocolVerifier.defeatProposal()`
   - `RocketDAOProtocolVerifier.claimBondChallenger()`
   - `RocketDAOProtocolVerifier.claimBondProposer()`
   - `RocketDAOProtocolVerifier.submitRoot()`
4. `onlyLatestContract("rocketDAOProtocolProposal", address(msg.sender))`:
   - `RocketDAOProtocolVerifier.burnProposalBond()`

It is possible to construe a scenario where the proposer is able to submit a proposal, a challenger can challenge the proposal, and an upgrade is done to the RocketDAOProtocolVerifier contract. This means subsequent calls to `RocketDAOProtocolVerifier.submitRoot()`, `RocketDAOProtocolVerifier.defeatProposal()`, `RocketDAOProtocolVerifier.claimBondChallenger()`, `RocketDAOProtocolVerifier.claimBondProposer()`, will all revert.

As a result, a proposal on an outdated contract may have a proposal bond that is unrecoverable since neither challenger or proposer can finalise the the status of the proposal by either executing it or defeating it, and there is no way to remove bonds after upgrade.

Upgrade scripts do not currently check if there are any pending DAO proposals, and doing so would open the opportunity for upgrades to be denied by creating throwaway proposals (though we grant that this would be a costly attack).

Recommendations

To avoid having to check whether there are existing DAO proposals, which may open up the opportunity for upgrade denial of service, the testing team advises on providing some functionality where users can retrieve their proposal bond after upgrades have been finalised.

Resolution

The finding has been acknowledged by the RocketPool team with the following comment:

> *"As we are in control of upgrades at this time, we can ensure any upgrades are backwards compatible with the bond system. Therefore, it does not pose a current threat."*

RPH-04	Incorrect Offset When Setting `periodsPaid`		
Asset	`RocketClaimDAO.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

Incorrect offset is referenced when setting `periodsPaid` variable in the storage.

```
// Update last paid timestamp and periods paid
setUint(bytes32(contractKey + lastPaymentOffset), lastPaymentTime + (periodsToPay * periodLength));
setUint(bytes32(contractKey + periodsPaid), periodsPaid + periodsToPay);    // @audit should be `+ periodsPaidOffset`
```

As a result, an `exists` variable will be overwritten instead, as `periodsPaid` will be zero to begin with and `existsOfsset` is also zero.

This could have significant implications to follow, as it will mark and effectively render the contract non-existent.

Recommendations

Modify code on line [**182**] to reference the correct offset, i.e. `bytes32(contractKey + periodsPaidOffset)`.

Resolution

The issue has been addressed in commit 3d1d634 as per the recommendations.

RPH-05	Incorrect Computation Of `claimIntervalsPassed`		
Asset	`RocketRewardsPool.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

Intervals since last claim period are incorrectly computed.

Currently, the interval start time is divided by claim interval duration, and subtracted from current block timestamp:

```
return block.timestamp - (getClaimIntervalTimeStart() / getClaimIntervalTime())
```

However, it should actually be the difference between current block's timestamp and claim interval start, divided by the interval duration.

Recommendations

Modify code on line [79] to correctly calculate intervals since last claim period, i.e.

```
return (block.timestamp - getClaimIntervalTimeStart()) / getClaimIntervalTime()
```

Resolution

The issue has been addressed in commit f8a658c as per the recommendations.

RPH-06	Incorrect RPL Stake Calculation During Withdrawal		
Asset	`RocketNodeStaking.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

Check on line [415] does not take into account amount of locked RPL balance, which should not be accounted for in the total withdrawable balance:

```
uint256 rplStake = getNodeRPLStake(msg.sender);
uint256 lockedStake = getNodeRPLLocked(msg.sender);
require(rplStake >= _amount, "Withdrawal amount exceeds node's staked RPL balance");
```

This could result in more RPL being withdrawn than allowed to.

Recommendations

Modify implementation to account for locked RPL stake, e.g.:

```
require(rplStake - lockedStake >= _amount);
```

Resolution

The issue has been addressed in commit 85bede4 as per the recommendations.

RPH-07	Behavioural Inconsistencies In Protocol Settings Initialisation Code		
Asset	`RocketDAOProtocolSettings*.sol`, `RocketUpgradeOneDotThree.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

Upgrades may not be able to complete successfully.

Each DAO type in the RocketPool network maintains a separate settings contract that determines various configurable options for the DAO proposal and actions contracts. Any new contracts for the RocketPool network are deployed and added to the network through an upgrade contract. Most of the contracts that make up the DAO Protocol Settings have constructors that require updating `RocketStorage` key and value pairs. This requires being added to the network's "allowed contracts", which typically happens during the upgrade itself.

Due to this behavioural inconsistency, the upgrade will require external contracts that run before the upgrade, which can predict the expected address, and then deploy the protocol settings after the upgrade. However, input into the upgrade contract is typically DAO controlled.

This could lead to a scenario where the upgrade is unable to proceed successfully.

The full list of affected contracts are:

- `RocketDAOProtocolSettingsAuction`
- `RocketDAOProtocolSettingsDeposit`
- `RocketDAOProtocolSettingsInflation`
- `RocketDAOProtocolSettingsMinipool`
- `RocketDAOProtocolSettingsNetwork`
- `RocketDAOProtocolSettingsNode`
- `RocketDAOProtocolSettingsProposal`
- `RocketDAOProtocolSettingsRewards`
- `RocketDAOProtocolSettingsSecurity`

On line [177-185] of the `RocketUpgradeOneDotThree` contract we can see how new contracts are added to `RocketStorage`. Furthermore, we can see the following code in the constructor of the `RocketDAOProtocolSettingsProposal` contract;

```
constructor(RocketStorageInterface _rocketStorageAddress) RocketDAOProtocolSettings(_rocketStorageAddress, "proposals") {
    version = 1;
    // Initialize settings on deployment
    if(!getBool(keccak256(abi.encodePacked(settingNameSpace, "deployed")))) {
        // Init settings
        setSettingUint("proposal.vote.phase1.time", 2 weeks);           // How long a proposal can be voted on in phase 1
```

The function `setSettingsUint()` will call `setUint()` which updates `RocketStorage`. However, this is only possible if the contract is added to the `RocketStorage` to begin with.

Recommendations

A possible workaround is using `create2` opcode to predict the address of the DAO protocol settings contracts, add those during the upgrade, then deploy. We believe this workaround to be complex, unnecessary and potentially undesired by the development team.

Alternatively, place the constructor logic in `initialise()` function, which is called once after the contracts are added to `RocketStorage`.

Resolution

The RocketPool team has acknowledged the issue with the following comment:

> *"We are aware of this situation and work around it by executing initialisation code in the upgrade contract."*

RPH-08	Incorrect `stake.for.allowed` Value		
Asset	`RocketNodeStaking.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

Incorrect node address is recorded as allowed to be staked for, which may result in unexpected and unintended results.

On line [**305**] the address to allow staking for is set as the caller of the function `msg.sender`, however, it should be a `_nodeAddress`, as the call may not necessarily come from the node itself, but it may also be a node's RPL withdrawal address:

```
setBool(keccak256(abi.encodePacked("node.stake.for.allowed", msg.sender, _caller)), _allowed);
```

Recommendations

Modify the implementation to record `node.stake.for.allowed` based on node address, e.g.:

```
setBool(keccak256(abi.encodePacked("node.stake.for.allowed", _nodeAddress, _caller)), _allowed);
```

Resolution

The issue has been addressed in commit d67fe0c as per the recommendations.

RPH-09	Challenged Leaf Indices Can Be Left Unverified Onchain		
Asset	`RocketDAOProtocolVerifier.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

Leaf node data is not verified onchain for Merkle depths of 5 or less during challenges on leaf nodes.

During proposal challenges, a proposer is required to respond to a challenger by providing nodes for the next Merkle pollard. Intended behaviour is to continue this round by round challenging until we get to leaf nodes corresponding to actual Rocket Pool nodes, at which point the data is then checked onchain. However, actual behaviour differs from this, and leaf node data is not verified onchain.

The following code block in the function `submitRoot()` is responsible for verifying leaf nodes and setting the challenge to responded:

```
if (indexDepth == treeDepth) {
    bytes32 actualHash = keccak256(abi.encodePacked(actual.sum));
    require(expected.hash == actualHash, "Invalid hash");
    setNode(_proposalID, _index, actual);
} else {
    require(expected.hash == actual.hash, "Invalid hash");
    if (indexDepth + depthPerRound >= treeDepth * 2) {
        uint256 n = getNextDepth(_index, nodeCount) - indexDepth;
        uint256 offset = (_index * (2 ** n)) - (2 ** (treeDepth * 2));
        require(verifyLeaves(getUint(bytes32(proposalKey + blockNumberOffset)), nodeCount, offset, _nodes), "Invalid leaves");
    }
}
```

As can be seen in this code block, leaves are only verified when `indexDepth + depthPerRound >= treeDepth * 2` is satisfied. In order to better understand our bounds and how index and tree depth relate to one another:

1. `indexDepth` cannot be greater than `treeDepth` as challenges cannot exceed the final leaf height. That means `indexDepth <= treeDepth`.
2. assuming the largest case, `indexDepth == treeDepth`, and codedepthPerRound = 5 this results in `2*indexDepth - indexDepth <= 5`.

The only situation that can lead to a truthful condition scenario is when `indexDepth <= 5`. Meaning it is only true when the index depth of the challenge is at most the final height of the first pollard (assuming the tree depth is one pollard). This restriction limits the likelihood of the attack to low as the real world case of the Network Voting Tree is that it will have far more than one pollard worth

Furthermore we can deduce from the above block that the responses to challenges (which occur when `setNode()` function is called) will only be noted when `indexDepth == treeDepth`. This condition can be simplified to `Math.log2(nodeCount, Math.Rounding.Up) == Math.log2(_index, Math.Rounding.Down);`. This will only occur if the index of the challenge is a leaf node.

These two scenarios combined produce the following unintended behavioural outcomes:

1. if the index of a challenge is a leaf node, the challenge will be marked as responded, whilst no verification is conducted
2. Merkle pollards of pollard length larger than 1 will not verify leaf indices onchain

Recommendations

The testing team recommends ensuring that leaf index challenges are also verified onchain.

Resolution

The issue has been addressed in commit 2cbf3e1 by moving the verification branch outside the parent condition branch so that verification occurs when the network tree is 5 layers or less deep.

RPH-10	Challengers Can Contest Non-Existent Indices To Steal Proposal Bond		
Asset	`RocketDAOProtocolVerifier.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

Due to errors in the `getPollardRootIndex()` function, users are able to create challenges for indices that exceed the Network Voting Tree length under specific conditions. This issue allows users to directly challenge indices they should not be able to contest, which could lead to lost proposal bonds for valid proposals.

When calculating the `pollardRootIndex` of an index for a Merkle pollard with depth less than 5, the pollard root index will return 1 for all indices of depth at least 5. If we consider a case with 8 nodes, an index depth of 31 will return a pollard root index of 1. However, this exceeds the max depth of our Network Voting Tree's Merkle pollard. The correct response would be a revert.

If the challenger can provide a valid witness for this non existent node, the challenge hash will be updated to the 31st index. This will result in a challenge that cannot be responded to.

Note, this issue is only applicable to trees with less than 32 indices, hence a low likelihood of exploitation as there is a significant number of Rocket Pool nodes on the network already.

Recommendations

The testing team recommends reverting if the index depth exceeds the `maxDepth` of the Network Voting Tree. This could be achieved as follows:

```
// Index is within the first pollard depth
if (_index < 2 ** depthPerRound) {
    if(_index > maxDepth) {
        revert("index exceeds network tree");
    }

    return 1;
}
```

Resolution

The issue has been addressed in commit 60684a7 as per the recommendations.

The RocketPool has also provided the following comment:

> *"This also highlighted another unintentional behaviour. A challenge could be made at any of the first 5 depths of either the network tree or a node tree. This has been corrected so that only indices that are in depths of multiples of 5 can be challenged. This was the intended behaviour previously but was misimplemented. This improved getPollardRootIndex() also makes it impossible to submit a challenge deeper than the maximum depth of the tree as was intended.".*

RPH-11	Implementation Discrepancies With RPIP Specifications		
Asset	`RocketMerkleDistributorMainnet.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The following implementation discrepancies with the RPIP specifications were observed:

- From RPIP-31 - *"As the controller of the RPL for a node, I MUST be able to trigger a claim of RPL rewards and restake a portion. If a node's RPL withdrawal address is set, the call MUST come from the current RPL withdrawal address"*.

  However, based on implementation of `claimAndStake()` on line [76] from `RocketMerkleDistributorMainnet`, it currently allows for the call to come from one of: the node's primary withdrawal address, node's address, or RPL withdrawal address:

  ```
  require(msg.sender == _nodeAddress || msg.sender == withdrawalAddress || msg.sender == rplWithdrawalAddress, "Can only claim
      ↪ from node or withdrawal addresses");
  ```

- From RPIP-33 - some fixed values from the parameter table do not match:
  - line [49] from `RocketDAOProtocolSettingsInflation` - `rpl.inflation.interval.rate` should be `> 1`
  - line [32] from `RocketDAOProtocolSettingsSecurity` - `_value` should be `< 0.75`
  - line [46] from `RocketDAOProtocolSettingsNetwork` - `network.submit.balances.frequency` should be `> 1 hours`

Recommendations

Modify identified implementations to align with RPIP specs, or clearly document implementation specifics and note any deviations from proposed RPIPs.

Resolution

The issue for RPIP-31 has been addressed in commit a747457.

For RPIP-33, the RocketPool team provided the following comment:

> *"The RPIP-33 values are correct in code and need to be updated in the RPIP."*

RPH-12	No Checks For Pending Withdrawal Addresses		
Asset	`RocketStorage.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

When setting a new withdrawal address, there are no checks to see if there are any pending withdrawal addresses in the queue.

This could result in the address that was just set being overwritten, if there are any old `pendingWithdrawalAddresses` that are accepted later on.

Recommendations

When setting a new withdrawal address, perform a check to see if there are any pending withdrawal addresses in the queue, and either reject the update, or clear the pending address queue after setting the withdrawal address.

Resolution

The RocketPool team has acknowledged the issue and responded with the following comment:

> *"Known issue. Cannot be fixed as RocketStorage is un-upgradable."*.

RPH-13	`*Old.sol` Contracts Do Not Match Current Version On Mainnet
Asset	`RocketDAOProtocolSettingsMinipoolOld.sol`, `RocketNetworkBalancesOld.sol`, `RocketRewardsPoolOld.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Code and recorded versions of some of the `*Old.sol` contracts do not match those currently deployed on mainnet:

- `RocketDAOProtocolSettingsMinipoolOld.sol` - should be `version = 2`, as currently observed on mainnet - see Block explorer
- `RocketNetworkBalancesOld.sol` - should be `version = 2`, as currently observed on mainnet - see Block explorer
- `RocketRewardsPoolOld.sol` - should be `version = 3`, as currently observed on mainnet - see Block explorer

Recommendations

Before upgrades, ensure the old contracts are recorded correctly and match those that are presently deployed on Mainnet.

Resolution

The issue has been addressed in commit da6febd as per the recommendations.

RPH-14	Snapshot Amendments May Lead To Leaf Verification Failures		
Asset	`RocketNetworkSnapshots.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The function `verifyLeaves()` may incorrectly return `false` if `rocketNetworkVoting.getVotingPower()` calculates a different `maxStakePercent` between proposal creation and leaf verification. If this occurs, a valid leaf may return `false` when a new root is submitted. This will cause leaf verification to also return `false`, preventing challenges from being contested, leading to a loss of proposal bond.

To reduce the risk of differences in `maxStakePercent`, RocketPool creates a snapshot of the `node.per.minipool.stake.maximum` value for a specific `block.number` to calculate `maximumStake` and produce the same expected value per staker for that specific proposal.

The `RocketNetworkSnapshots` contract uses a binary search algorithm to retrieve these snapshot key values for blocks within a defined upper limit. Amendments are currently possible by pushing updates to the snapshots for a specific key where the block matches a previous snapshot. This is illustrated in the following code block:

```
// Update or push new checkpoint
if (last._block == _block) {
    last._value = _value;
    _set(_key, pos - 1, last);
} else {
    _push(_key, Checkpoint224({_block: _block, _value: _value}));
}
```

Based on the above, amendments are possible as long as `last._block == _block`. It is worth noting that although network voting ETH staked snapshots might be updated regularly, the `node.per.minipool.stake.maximum` is updated infrequently. This increases the chance that an old snapshot is overwritten, what may affect a current proposal from successfully running `verifyLeaves()` function.

Recommendations

Consider setting used block to `block.number` in the `_insert()` function of the `RocketNetworkSnapshots` contract.

Alternatively, a revert could be triggered when existing blocks are used for new snapshots. It is important to consider these changes holistically according to the entirety of use cases supported by `RocketNetworkSnapshots` before implementing changes. Though these fixes aim to improve the security posture of the pDao, they may introduce issues in areas outside of the scope of this review.

If other parts of the system expect to be capable of amending previous block snapshots for certain keys, it is advised to create a separate snapshot functionality for the `node.per.minipool.stake.maximum`.

Resolution

The issue has been addressed in commit e4fc9c8 by hardcoding `block.number` into the `_insert()` function.

RPH-15	Initial Votes Can Be Cast In Phase 2 Without Proof
Asset	`RocketDAOProtocolProposal.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

Voters, who did not cast their vote in phase one, can still do so in phase two via `overrideVote()` function, which also does not require them to provide a Merkle proof.

By design, as per RPIP-33, in the first voting phase, voting delegates, and node operators who have not delegated, can cast their vote by providing a Merkle proof of their voting power (relative to the submitted proposal root). Once the first phase has passed, voting enters the second voting phase. Node operators who have delegated their vote, get the opportunity to override their delegate's vote, if they disagree.

However, there are no checks preventing voters who did not cast their vote in phase one, and who do not have delegates set, or whose delegates have not voted in phase 1, from calling `overrideVote()` to cast their vote.

As a result, voters who did not vote during phase one, can still cast their votes without needing to provide a Merkle proof in phase two.

Recommendations

Modify implementation of `overrideVote()` to prevent voters without delegates set, or with delegates who did not vote in phase one, from calling the function.

Resolution

This finding has been closed as false-positive with the following comment from the RocketPool team:

> *"This is intentional behaviour. For regular users that do not have any delegated voting power and have not delegated their vote, they can either vote in phase 1 with a proof or wait until phase 2 and vote without having to provide a proof. The outcome is identical. Phase 1 is primarily for delegates to vote with the sum of their delegated power and requires the Merkle tree to prove this sum. For nodes voting on their own, it is unnecessary to produce the Merkle tree as their own voting power is known on chain without any extra proof required. By default, all nodes are delegated to themselves. So their voting power and their delegated voting power are equal. Hence, whether voting in phase 1 or 2, the outcome is the same."*

RPH-16	Node's Votes Can Be Overridden Without Its Knowledge
Asset	`RocketDAOProtocolProposal.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

Any node's votes can be arbitrarily overridden during phase two without node's prior knowledge or approval.

A node can forcefully nominate any address as a delegate via `setDelegate()`. There is no mechanism in place for the nominated node to accept or reject this role. If a node (voter) nominates other existing node (delegate), it will be able to override that delegate node's original vote.

For example, consider the following scenario:

1. A forced delegate node votes for a certain proposal and has a voting power of 10
2. Total number of votes for the proposal is currently 20 (i.e. there were other 10 *FOR* votes casted by other nodes) and 5 *AGAINST*
3. Voter node that set the delegate from step 1 comes in and votes *AGAINST* the proposal. It's voting power is also 10
4. Delegate's votes will be overridden by reducing total amount of *FOR* votes by voter's voting power (10)
5. Voter's vote will then be casted on top of it in phase two (assuming they didn't vote in phase one)
6. In the end, total number of *FOR* votes is 10, and 15 *AGAINST*. The proposal is rejected with 10-15 votes.
7. The voter was able to overthrow the proposal, which otherwise would have passed with 20-15 votes.

Note, this issue may also have more implications where both nodes voting powers differ, particularly when nominating node's voting power is larger than this of the delegated node. In such situations, the vote deduction during override will exceed delegated node's vote count, and, as such, would result in deducting votes that were placed by other nodes.

Recommendations

Implement a 2-step mechanism to nominate, and then accept the delegate role. Only add an address as delegate if it has explicitly accepted it.

Add extra control mechanisms to `overrideVote()` to ensure that delegate's and node's voting powers match, and only number of votes casted by the delegate can be deducted, not the `msg.sender` total voting power.

Resolution

This finding has been closed as false-positive with the following comment from the RocketPool team:

"In phase 1, only nodes with delegated voting power can vote. So the "attacker" is unable to vote in phase 1 as they have delegated all their power to the "forced delegate node". In the network Merkle sum tree, their voting power is 0. In phase 1, the forced delegatee has 20 voting power (their own 10 and the 10 from the attacker). Only in phase 2 can the attacker override their delegatee's voting power. At this point the delegatee has applied 20 voting power to FOR. So overriding that with an AGAINST vote will result in 10 FOR votes and 10 AGAINST votes."

RPH-17	Delegates Are Unable To Vote
Asset	`RocketDAOProtocolProposal.sol`, `RocketDAOProtocolVerifier.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

When submitting a vote as a delegate, the `vote()` function will always revert with *"Invalid Node"* as `rocketDAOProtocolVerifier.verifyVote()` expects the call to always come from the node itself.

When submitting a vote during phase 1 via `vote()`, delegates are expected to be able to cast a vote on behalf of a node operator. However, the following call will always fail:

```
rocketDAOProtocolVerifier.verifyVote(msg.sender, _nodeIndex, _proposalID, _votingPower, _witness)
```

This is due to the following check in `verifyVote()`:

```
function verifyVote(address _voter, uint256 _nodeIndex, uint256 _proposalID, uint256 _votingPower, Types.Node[] calldata _witness)
      ↪ external view returns (bool) {

// ...(snip)...

// Verify voter
if(rocketNodeManager.getNodeAt(_nodeIndex) != _voter) {
    return false;
}
```

If `_voter` is a delegate and not an original node, the voter verification will always fail. This is also due to `setDelegate()` merely updating the storage variable `node.delegate`, which is then not checked for.

Recommendations

Modify implementation of `verifyVote()` to cater for situations where a call may come from a node's delegate. This could be done by checking if supplied `_voter` is node, or a node's delegate, if set.

Resolution

The finding has been closed as false-positive with the following comment from the RocketPool team:

> *"Delegatees do not vote separately on behalf of each of their delegators. They place a single vote with the sum of all their delegated voting power proved by the Merkle sum tree. Their leaf node in the network Merkle sum tree is a summation of all nodes who have delegated to them."*

RPH-18	Inadequate Merkle Height Verification
Asset	`RocketDAOProtocolVerifier.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

There is no explicit tree height verification for the requested index during verification of Merkle proofs.

The `verifyVote()` function is a critical code branch that validates if voters have the supplied voting power. A malicious user may provide a merkle proof with a shortened witness due to lack of height verification for the requested index.

The `verifyVote()` function has several user supplied parameters - `_votingPower`, `_witness` and `treeIndex`. The `treeIndex` and `_witness.length` are used as bounds for generating the Merkle root. If this root matches the expected, the vote is considered verified. The `computeRootFromWitness()` function does not make any checks to ensure the witness length matches what is expected for a given `treeIndex`.

Note, the issue does not appear to be directly exploitable, due to how the hashes are calculated for leaf nodes. With the leaf node hashes being different for intermediate nodes, a malicious user is only able to provide `_votingPower` for leaf nodes. As a result, modifications to the witness will not yield the expected merkle root.

Although indirect checks exist that protect against exploits, these checks are performed in different parts of the code and could be mistakenly omitted, particularly when introducing new functionality and potential future contract updates.

Recommendations

Consider implementing additional checks in the `computeRootFromWitness()` function to ensure that witness length matches the expected length for a specific `treeIndex`.

Resolution

The issue has been acknowledged and closed with the following comment from the development team:

> *"There is no immediate threat or exploit at this time."*

RPH-19	Miscellaneous General Comments
Asset	`contracts/*`
Status	**Closed:**
Rating	Informational

Description

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

1. **No checks if a node exists when retrieving its withdrawal address.**

   In `RocketStorage` contract, there are no checks to verify if a node exists when calling `getNodeWithdrawalAddress()`. As a result, enquired address `_nodeAddress` will be returned, even when a node doesn't exist.

   Note, this is called by other upper level functions, such as `RocketNodeManager.getNodePendingWithdrawalAddress()`.

2. **Incorrect event.**

   In `RocketNodeDeposit`, an incorrect event is emitted on line [113] - should be `Withdrawal` not `DepositFor`.

3. **Inconsistent withdrawal address logic.**

   - In `RocketNodeStaking`, the `withdrawRPL()` function logic dictates that if a node's RPL withdrawal address is unset, the call must come from the node's primary withdrawal address, or the node's address.

     However, this is inconsistent with behaviour of `withdrawEth()` where it has to be the withdrawal address, if set, or node address, not either. This could lead to confusion if not documented correctly.
   - In `RocketNodeStaking`, the `stakeRPLFor()` on line [272], consider if the ETH withdrawal address should also be accepted, if set.

4. **Additional checks.**

   - In `RocketNodeManager`, the `confirmRPLWithdrawalAddress()` function could benefit from `onlyRegisteredNode` modifier to ensure only calls on valid nodes are accepted.
   - In `RocketNodeStaking`, the `transferRPL()` function should check if `_from` is a valid node and if it has a sufficient balance to cover the `_amount`. A revert may occur later on otherwise during transfer, but this would save gas and return a user friendly error.
   - In `RocketNodeDeposit`, the `depositEthFor()` function could benefit from checks to ensure that `msg.sender != _nodeAddress`, i.e. the node is not depositing on behalf of itself - in such cases it should just use `deposit()` instead.
   - In `RocketNodeDeposit`, the `depositWithCredit()` function could benefit from an additional check and a clear error message when `_bondAmount < msg.value`.

5. **NatSpec improvements.**

   - In `RocketNodeDeposit`, NatSpec of `depositWithCredit()` is incomplete and appears to be a copy-paste of `deposit()`.
   - In `RocketNodeDeposit`, NatSpec of `_processNodeDeposit()` could be more detailed and explain various conditions and scenarios covered.

- In `RocketDAOProtocolVerifier`, NatSpec of `createChallenge()` is incomplete and missing details on input parameters `_node` and `_witness`.

6. **Gas savings.**
   - In `RocketClaimDAO`, consider adding `require(periodsToPay > 0)` after line [174], otherwise, calculations and updates that follow will yield no result as periods paid would have already been exhausted.
   - In `RocketDAOProtocolSettingsInflation`, checks on line [48] and line [49] can be moved higher up before calling `inflationMintTokens()` to avoid calling the function if an invalid value was provided.
   - For loops, use postfix operations when incrementing `i`. Prefix `++i` uses two less operations therefore saving gas.

7. **Lack of in-depth test coverage of DAO functionalities.**

   Insufficient test coverage of edge case scenarios, including complex Merkle trees and basic functionality (e.g. utilising delegates for voting) was observed.

   It is recommended to ensure comprehensive unit tests aiming to achieve 100% test coverage and supplementary fuzzing tests are developed.

Recommendations

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

Resolution

The comments above have been acknowledged by the RocketPool team, and relevant changes actioned in the following commits:

1. 30702a1
2. b236759
3. 6ef00e5

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are given along with this document. The `brownie` framework was used to perform these tests and the output is given below.

```
Running 17 tests for test/Dao.t.sol:DaoTest
[PASS] testCreateChallengeForNonExistingProposal() (gas: 93526)
[FAIL. Reason: revert: Invalid hash] testCreateChallengeInvalidLeafNode() (gas: 1073050)
[PASS] testCreateChallengeNormalProposal() (gas: 1064526)
[PASS] testCreateInvalidChallengeIndexNotCorrespondingToWitness() (gas: 1037894)
[FAIL. Reason: call did not revert as expected] testCreateInvalidChallengeIndexOutOfBounds() (gas: 1038235)
[FAIL. Reason: revert: Invalid hash] testCreateInvalidChallengeWithResponse() (gas: 1064758)
[FAIL. Reason: revert: Block too old] testDelayedProposalPropose() (gas: 187648)
[PASS] testFuzzFirstPollardRootIndex(uint256,uint256) (runs: 1000, 9518, 9546)
[PASS] testFuzzSecondPollardRootIndex(uint256,uint256) (runs: 1000, 11300, 11449)
[PASS] testInvalidBurning() (gas: 797576)
[PASS] testOverrideNodeVote_PoC() (gas: 1240622)
[PASS] testPropose() (gas: 794518)
[PASS] testSigPMerkleImplementation() (gas: 95958)
[PASS] testSpecificPollardRootIndex() (gas: 7497)
[PASS] testVote() (gas: 1432846)
[FAIL. Reason: revert: Invalid node] testVoteAsDelegate() (gas: 854027)
[PASS] testVoteInPhase2_PoC() (gas: 988413)
Test result: FAILED. 12 passed; 5 failed; 0 skipped; finished in 102.73ms

Running 2 tests for test/Merkle.t.sol:MerkleTest
[PASS] testGetDepthFromIndex(uint256,uint256) (runs: 1000, 8301, 8434)
[PASS] testGetMaxDepth(uint256) (runs: 1000, 5591, 5733)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 112.93ms

Running 5 tests for test/forking/ForkTest.sanity.t.sol:ForkSanityTest
[PASS] test_Deposit_HardcodedAddressMatchesStorage() (gas: 8295)
[PASS] test_RETH_HardcodedAddressMatchesStorage() (gas: 8268)
[PASS] test_RPL_HardcodedAddressMatchesStorage() (gas: 8245)
[PASS] test_Storage_IsInitialised() (gas: 7746)
[PASS] test_Storage_getGuardian_exists() (gas: 7740)
Test result: ok. 5 passed; 0 failed; 0 skipped; finished in 4.14s

Running 1 test for test/forking/RocketUpgradeOneDotThree.t.sol:UpgradeVersionTest
[FAIL. Reason: assertion failed] test_BugPOC_UpgradedContractVersionsHaveIncremented() (gas: 88132914)
Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 234.52s

Running 5 tests for test/forking/RocketUpgradeOneDotThree.t.sol:UpgradeTest
[FAIL. Reason: revert: Invalid or outdated network contract] test_BugPOC_DeployRocketDAOProtocolSettingsProposals() (gas: 95352)
[FAIL. Reason: revert: Invalid or outdated network contract] test_BugPOC_DeployRocketDAOProtocolSettingsSecurity() (gas: 95843)
[FAIL. Reason: revert: Invalid or outdated network contract] test_BugPOC_DeployUpgradeContracts() (gas: 41594766)
[PASS] test_Workaround_DeployNewSettingsContracts() (gas: 2503798)
[PASS] test_Workaround_HoustonUpgrade() (gas: 87964691)
Test result: FAILED. 2 passed; 3 failed; 0 skipped; finished in 234.52s

Running 1 test for test/forking/UpgradedForkTest.sanity.t.sol:NewContractsNotYetRegisteredTest
[PASS] test_HoustonContractsNewlyRegistered() (gas: 89362)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 234.53s

Running 1 test for test/forking/ForkTest.sanity.t.sol:NewContractsNotYetRegisteredTest
[PASS] test_HoustonContractsNotYetRegistered() (gas: 41315)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 234.54s

Running 16 tests for test/forking/UpgradedForkTest.sanity.t.sol:UpgradedProtocolVersionTest
[PASS] test_RocketClaimDAO_VersionAsExpected() (gas: 13515)
[PASS] test_RocketDAOProtocolProposals_VersionAsExpected() (gas: 13534)
[PASS] test_RocketDAOProtocolSettingsAuction_VersionAsExpected() (gas: 13590)
[PASS] test_RocketDAOProtocolSettingsDeposit_VersionAsExpected() (gas: 13569)
[PASS] test_RocketDAOProtocolSettingsInflation_VersionAsExpected() (gas: 13624)
[PASS] test_RocketDAOProtocolSettingsMinipool_VersionAsExpected() (gas: 13648)
[PASS] test_RocketDAOProtocolSettingsNetwork_VersionAsExpected() (gas: 13613)
```

```
[PASS] test_RocketDAOProtocolSettingsRewards_VersionAsExpected() (gas: 13613)
[PASS] test_RocketDAOProtocol_VersionAsExpected() (gas: 13568)
[PASS] test_RocketMinipoolManager_VersionAsExpected() (gas: 13658)
[PASS] test_RocketNetworkBalances_VersionAsExpected() (gas: 13492)
[PASS] test_RocketNetworkPrices_VersionAsExpected() (gas: 13525)
[PASS] test_RocketNodeDeposit_VersionAsExpected() (gas: 13547)
[PASS] test_RocketNodeManager_VersionAsExpected() (gas: 13635)
[PASS] test_RocketNodeStaking_VersionAsExpected() (gas: 13658)
[PASS] test_RocketRewardsPool_VersionAsExpected() (gas: 13591)
Test result: ok. 16 passed; 0 failed; 0 skipped; finished in 234.53s

Running 16 tests for test/forking/ForkTest.sanity.t.sol:ProtocolVersionTest
[FAIL. Reason: assertion failed] test_BugPOC_RocketDAOProtocolSettingsMinipool_VersionAsExpected() (gas: 32724)
[FAIL. Reason: assertion failed] test_BugPOC_RocketNetworkBalances_VersionAsExpected() (gas: 32599)
[FAIL. Reason: assertion failed] test_BugPOC_RocketRewardsPool_VersionAsExpected() (gas: 32751)
[PASS] test_RocketClaimDAO_VersionAsExpected() (gas: 13481)
[PASS] test_RocketDAOProtocolProposals_VersionAsExpected() (gas: 13502)
[PASS] test_RocketDAOProtocolSettingsAuction_VersionAsExpected() (gas: 13558)
[PASS] test_RocketDAOProtocolSettingsDeposit_VersionAsExpected() (gas: 13470)
[PASS] test_RocketDAOProtocolSettingsInflation_VersionAsExpected() (gas: 13547)
[PASS] test_RocketDAOProtocolSettingsNetwork_VersionAsExpected() (gas: 13536)
[PASS] test_RocketDAOProtocolSettingsRewards_VersionAsExpected() (gas: 13514)
[PASS] test_RocketDAOProtocol_VersionAsExpected() (gas: 13480)
[PASS] test_RocketMinipoolManager_VersionAsExpected() (gas: 13614)
[PASS] test_RocketNetworkPrices_VersionAsExpected() (gas: 13513)
[PASS] test_RocketNodeDeposit_VersionAsExpected() (gas: 13536)
[PASS] test_RocketNodeManager_VersionAsExpected() (gas: 13547)
[PASS] test_RocketNodeStaking_VersionAsExpected() (gas: 13537)
Test result: FAILED. 13 passed; 3 failed; 0 skipped; finished in 234.54s

Ran 9 test suites: 52 tests passed, 12 failed, 0 skipped (64 total tests)

Failing tests:
Encountered 5 failing tests in test/Dao.t.sol:DaoTest
[FAIL. Reason: revert: Invalid hash] testCreateChallengeInvalidLeafNode() (gas: 1073050)
[FAIL. Reason: call did not revert as expected] testCreateInvalidChallengeIndexOutOfBounds() (gas: 1038235)
[FAIL. Reason: revert: Invalid hash] testCreateInvalidChallengeWithResponse() (gas: 1064758)
[FAIL. Reason: revert: Block too old] testDelayedProposalPropose() (gas: 187648)
[FAIL. Reason: revert: Invalid node] testVoteAsDelegate() (gas: 854027)

Encountered 3 failing tests in test/forking/ForkTest.sanity.t.sol:ProtocolVersionTest
[FAIL. Reason: assertion failed] test_BugPOC_RocketDAOProtocolSettingsMinipool_VersionAsExpected() (gas: 32724)
[FAIL. Reason: assertion failed] test_BugPOC_RocketNetworkBalances_VersionAsExpected() (gas: 32599)
[FAIL. Reason: assertion failed] test_BugPOC_RocketRewardsPool_VersionAsExpected() (gas: 32751)

Encountered 3 failing tests in test/forking/RocketUpgradeOneDotThree.t.sol:UpgradeTest
[FAIL. Reason: revert: Invalid or outdated network contract] test_BugPOC_DeployRocketDAOProtocolSettingsProposals() (gas: 95352)
[FAIL. Reason: revert: Invalid or outdated network contract] test_BugPOC_DeployRocketDAOProtocolSettingsSecurity() (gas: 95843)
[FAIL. Reason: revert: Invalid or outdated network contract] test_BugPOC_DeployUpgradeContracts() (gas: 41594766)

Encountered 1 failing test in test/forking/RocketUpgradeOneDotThree.t.sol:UpgradeVersionTest
[FAIL. Reason: assertion failed] test_BugPOC_UpgradedContractVersionsHaveIncremented() (gas: 88132914)
```

Appendix B 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].